Security for Object-Oriented Database Systems

Jonathan K. Millen
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730

Teresa F. Lunt
SRI International
Computer Science Laboratory
333 Ravenswood Ave.
Menlo Park, CA 94025

Abstract

A multilevel object-oriented system can be implemented on a conventional mandatory security kernel. Each object is assigned a single security level that applies to all its contents (variables and methods). The informal security policy model includes properties such as compatibility of security level assignments with the class hierarchy. The representation of integrity constraints and classification constraints is illustrated.

1. Introduction

This paper proposes a design approach for a secure multilevel object-oriented database system. The object model seems to be an important direction of the future for database systems, as evidenced by the barrage of research and development in that area [e.g., KiLo89]. Our interest in the object model is also motivated by its natural applicability to knowledge-based systems, since it supports the notion of a class hierarchy, an essential ingredient of such systems [Brac83]. The class hierarchy provides a uniform way to represent and manipulate both knowledge and data. Unlike the relational model, it naturally captures the semantics of the information it contains. Furthermore, the object model's encapsulation of data and code into objects suggests a clean implementation of the system on a reference monitor.

We first discuss the essential features of a general object system model, and then extend the object model to incorporate mandatory label-based security. We go on to show how typical database security and integrity policies can be supported by this model, with special attention to inference problems and integrity constraints. Knowledge-based system concerns are addressed in [MiLu89], and an extension to handle composite objects is in [GaLu91].

Before proceeding, it is important to stress two things. First, our design for a multilevel object-oriented system relies on an underlying security kernel for its enforcement of mandatory security properties. This means that the layer providing the object-oriented interface need not be trusted. Second, the approach taken in this report is to assign a security level to each entire object. Other work in applying multilevel security to object-oriented systems requires trusted enforcement mechanisms in the object layer [Keef88, JaKo90]. Our single-level-object approach is attractive for its simplicity and its compatibility with a security kernel. We will also give evidence that the single-level-object approach is flexible enough to support realistic security policies.

There are many substantially different kinds of object-oriented systems. Our model adopts some of the main features of the Smalltalk system [GoRo83]. Smalltalk encapsulates programs and data into objects, organized into a class hierarchy that supports inheritance. The programming model is based on message passing between objects and message handlers within objects. Our version of these concepts is summarized below.

Objects

An object corresponds roughly to the notion of a logical record or relational tuple, with respect to its data content. An object stores named attributes in variables, which possess values. See Figure 1. The value of a variable is (unlike Smalltalk) not another object, but some data structure such as a string, number, etc. To support references to other objects, there is a data type called an object-id, discussed below.

![Object Diagram](image-url)

Figure 1. An object
The hierarchy

A hierarchy is, as usual, a directed rooted tree, with the root at the top. The relation from an object to the objects immediately below it is the instance relation. The inverse relation, going upward, is the class relation. A class object (sometimes just called a class) is nothing more than an object that is a class for some object; i.e., it has objects below it. A class object usually represents organizational entities at a higher conceptual level than logical records. If the database contains employee records, for example, there might be a class object called "Employee", containing the common attributes of all employees, and whose instances are individual employee records. More examples will be given later.

Smalltalk distinguishes between a subclass and an instance relation, and between class objects and instance objects. An instance object is an individual, meaning it can have no instances of its own. For this paper, we do not make use of this distinction. The distinction is significant primarily for reasons of implementation efficiency; it does not affect security policy, and it simplifies the statement of our security policy to ignore it. This is merely a terminological device; our model is applicable to systems that make the subclass-instance distinction.

Inheritance

Inheritance is the most important function of the class hierarchy. There is a close relationship between an object and its class: all the structure and any of the variable values of the the class are inherited by the object. In particular, all of the variables present in the class are also present in each of its instances. Furthermore, if an instance is missing a value for one of its inherited variables, a default value may be inherited from the class object.

Message handlers (i.e., methods) as well as variables are inherited by instances; they are discussed below.

An instance may have additional variables and methods not present in its class. New variables usually refine the concept defined by the class, creating a subclass. Below Employee, for example, there might be subclasses Technical and Support, with different additional attributes, and the actual employee records would be below one of those.

In some systems, the same object might be an instance of two different classes. An Employee, for example, might also be a Student. A common instance would inherit attributes from all its classes. Multiple inheritance can lead to dilemmas when attributes of the various superclasses conflict, but the problems that arise do not appear to be security problems. In this paper, we adopt the simplifying assumption that multiple inheritance is not present.

Methods and messages

An object has methods defined for it. Methods encapsulate the behavior of an object, in that an object can be acted upon only through executing the methods defined for the object. Methods are invoked by sending messages to an object. (See Figure 2.) A message consists of a command, which selects the appropriate method, and some arguments, if necessary.

Message: Command (Args)

Figure 2. Message processing

A method performs three sorts of activities: (1) it may read and write the variables of the object where it resides; (2) it may send messages to other objects, to invoke methods there; and (3) it may, when it terminates, return a value to the sender of the message that invoked it.

Since methods are inherited by instances, a message can be handled by an object if there is any object above it in the hierarchy (many levels above, perhaps) having a method for it.

References to objects

Every object has a unique identifier called an object-id. A reference to an object is an occurrence of its identifier in another object, in a variable. A reference to an object is needed when one wishes to send a message to it; it is the object's address.
In Smalltalk, the value of a variable is always an object. Saying that "the value of a variable is an object" is a figure of speech that requires caution, because careless use of it can complicate the statement of a security policy. For our purposes, it should be understood that the value of a variable is actually an object-id, when it is intended to refer to another object. When a variable has a value of some simple built-in type such as Integer or String, it can be stored within the object to which the variable belongs, and there is no need to use an object-id to refer to it.

Objects can be linked to one another by references in suitable variables. References can represent any of a multitude of relationships. For example, an Employee object may have a Job variable. Its value is often not just the name of a job, but rather a reference to another object containing information about the job, such as its duties and salary range.

Objects are also linked by the class-instance relation. In Smalltalk, this relation is implicit and maintained by the system. For the purposes of modeling security, we shall make this relation explicit, by assuming the existence of standard variables called "Class" and "Instances" in every object. The value of the Class variable is the identifier of the parent class of the object. The value of the Instances variable is a (possibly empty) list of identifiers of the object's instances. Figure 3 illustrates the class-instance relation.

It is very common for a variable to have a list or set of object identifiers as a value. Smalltalk would insist upon recognizing a set or list as another kind of object, so that one could say that a variable always had one object as its value. In a security context, where objects have security attributes, it complicates matters to view a set or list as another object. It is easier for us to say that a variable has a set of values. The ordering its elements is not important for security purposes.

2. Security policy

Here we present an informal model for a secure object-oriented system. It is a label-based reference-monitor model, in that it employs subjects, objects, security levels, and access restrictions. It describes and restricts the behavior of the application interface to an object-oriented system, with general axioms rather than specific transition rules. An important characteristic of our model is that each object has a single security level that applies to everything within it. (This is analogous to tuple-level labeling in the relational model.)

Entities

To specify a security policy for an object-oriented system, we must first describe the entities that make up an abstract model of the system.

The basic sets underlying the model are subjects, objects, a lattice of security levels, and the access modes read, write and execute. There is a labelling function that assigns security levels to subjects and objects. This much is supported by any mandatory security kernel.

A model of object-oriented security must also take into consideration the information flows that occur as a result of inheritance and the message-passing model of program execution. For this we need additional structure: (1) the class function indicating the class object of a given object, (2) a function assigning a home object to each subject, and (3) a function indicating the invoker of each subject—that is, the subject that sent the message being handled by the given subject.

In order to implement an object-oriented interface layer on a mandatory security kernel, the various functions are constrained by the *-property and other measures enforced by any such kernel. These derived constraints are presented below as six security properties in the context of the entities they apply to.

Objects

Because all objects are single-level, an object in the object system sense is also an object in the security modeling sense (although it may be implemented using more than one storage object as supported by the kernel). The security level of an object is fixed (except under special
circumstances noted below). Entities that belong to an object, such as its methods and variables, do not have security levels of their own; informally, they may be considered to be at the level of the object containing them.

The security level of an object is constrained by the following property:

Property 1 (Hierarchy Property). The level of an object must dominate that of its class object.

This property is needed to permit the object to inherit methods and variables from its parent. Any attempt to read an object by reading one of its variables, or execute it by sending a message to it, may implicitly read the object's parent class and perhaps the parent's parent class, and so on, until a default value or appropriate method is found.

Subjects

A subject is the active entity that executes methods to handle received messages, and also sends messages. Messages are addressed to objects. A subject is created when a message is received by an object, and that object becomes the home object for the new subject. The new subject exists for the sole purpose of handling the message, by executing an appropriate method. The subject may be destroyed when the method terminates. Although a subject is thought of as being located at its home object, its security level is not necessarily equal to that of the home object - it may be higher. A subject is given a security level when it is created, in accordance with Property 2. This is illustrated in Figure 4.

Property 2 (Subject Level Property). The security level of a subject dominates the level of the invoking subject, and it also dominates the level of its home object.

This property is needed so that the subject that is created to handle the message can read the message and also read variables and methods in the object where it is located. It is enough for the subject to be created at a level equal to the least upper bound of the invoker and object levels. However, it is sometimes useful to allow a subject to upgrade itself. Subsequent downgrading of a subject leads to a covert channel, and should not be allowed.

Note that messages themselves are not mentioned explicitly in the abstract security model. Their existence is implicit in the subject-invoker function. Informally, when thinking about messages as implementation entities, one can consider a message to be at the level of the invoking subject.

The underlying security kernel may or may not allow subject levels to change; the object system will simply comply what whatever is permitted. A given security kernel may or may not permit certain subjects to be trusted, giving them privileges to, for example, cause changes in object security levels or to downgrade themselves. Again, that is up to the underlying kernel. From a trusted system evaluation point of view, the implication of trusting a subject is that the method it is executing consists of trusted code, and must be protected accordingly.

Access policy

As part of the object system model rather than the security setup, we confine a subject's accesses to variables in its home object. If a subject wants to read or write variables some other object, all it can do is send a message to request retrieval or modification of some variable there. Because of inheritance, a subject may implicitly read variables or methods in objects above it in the hierarchy; that is taken care of by the Hierarchy Property.

Property 3 (Object Locality Property). A subject can execute methods or read or write variables only in its home object.

A subject attempting to execute a method or read or write a variable is constrained by the *-property of the security kernel. This property is elevated to the object system interface as follows:

![Figure 4. Subject Levels](image)
**Property 4 (®-Property).** A subject may write into its home object only if its security level is equal to that of the object.

Writing up and reading up are prevented by the combination of the Object Locality and Subject Level Properties.

**Other information-flow properties**

The foregoing four properties are easily formalizable in terms of the entities and functions already given. There are some additional properties that would require more effort to formalize, because they depend on the details of particular object system transition functions, and we do not wish to specify that level of detail here. However, there are two properties that would certainly be necessary for any secure object system, and they are stated informally and motivated below.

**Return values**

A subject is permitted to send messages to any objects it likes, including objects at a higher level. However, when a subject sends a message to a higher-level object, the invoking subject will not be able to receive a value in return, since that would constitute an illegal information flow. This observation is expressed in the following Return Value Property and is illustrated in Figure 5.

**Property 5 (Return Value Property).** An subject can send a return value to its invoking subject only if it is at the same security level as the invoking subject.

In the implementation, the invoking subject will have to receive something in place of a return value - perhaps a standard null value, or an acknowledgement from the system that a value from the addressee is unavailable. Note that the system, rather than the higher-level subject, should determine the time it takes to deliver the null value, otherwise a timing channel will exist. The underlying TCB is responsible for this protection.

**Object creation**

A subject typically causes an object to be created by sending a "Create instance" message to the desired class object. In order to avoid an obvious covert channel, the child object must be given a security level at least as high as the requesting subject.

**Property 6 (Object Creation Property).** The security level of a newly-created object dominates the level of the subject that requested the creation.

3. **Object system structure**

A layered object system

Suppose we are given an operating system kernel that enforces mandatory security and provides a general set of kernel calls similar to the Bell-LaPadula model rules. We believe that an object system designed in accordance with the policy in Section 2, and the architecture summarized below, can be implemented as an untrusted layer over the kernel.

The user application interface to the object layer consists of certain primitive operations, which are divided into system calls and system methods. The primitive operations will be discussed below. The primitives will do their own security checking according to the policy described in Section 2, but the underlying kernel will do its own independent enforcement, so the primitives need not be trusted for that purpose. Their function is to maintain the object system abstraction and to protect the object system user from kernel error messages. The layered system is illustrated in Figure 6.

The structural approach presented here has been tested in a feasibility demonstration prototype implemented on a Macintosh using Hypercard. The system is not secure, since there is no underlying TCB, but it has been useful in checking functionality.
Object system primitives

Begin by visualizing an "empty" object system: one with system software only. This might be called an object system "shell", used as a starting point for object-oriented applications. Another layer would be needed on top of this to support database management features.

At this point we have the list of system calls and system messages shown in Table 1. These are sufficient, in a minimal way, to act as the basis for more sophisticated functions for building an object-oriented system.

<table>
<thead>
<tr>
<th>TABLE 1. Object System Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYSTEM CALLS</strong></td>
</tr>
<tr>
<td>Send</td>
</tr>
<tr>
<td>Upgrade</td>
</tr>
<tr>
<td>Return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SYSTEM MESSAGES</strong></th>
<th><strong>ARGUMENTS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>level, new-variables</td>
</tr>
<tr>
<td>Delete</td>
<td>object-id</td>
</tr>
<tr>
<td>AddMessage</td>
<td>command, args, method</td>
</tr>
<tr>
<td>Getvar</td>
<td>variable</td>
</tr>
<tr>
<td>Setvar</td>
<td>variable, value</td>
</tr>
</tbody>
</table>

Note on syntax

When we show examples of methods, system calls will be written using the notation:

<system call> <argument>, <argument>

and messages will have their arguments in parentheses:

<command> (<argument>, <argument>).

Thus, to use the system call Send to send a message, we have to write:

Send <object-id>, <command> (<argument>, <argument>).

A message returns a value by depositing it in the special reserved program variable "value". Variables in the local object can be accessed with a syntax like:

Self.Var1 := Self.Var2

to read variable Var2 and store its value into Var1. Otherwise, a method looks like an ordinary program, with program variables, if-then-else statements, and so on.

4. The semantics of security

In this section, we discuss how the single-level-object approach can effectively and properly address multilevel database needs. The first step is to understand just how classification labels are applied to elements of a database.
Gary Smith has recently pointed out the ambiguity, due to the lack of a consistently applied semantic interpretation, of associating labels with the various structural elements in a database system [Smith88]. It is important to define precisely what it means to say that an object is classified. This amounts to specifying precisely what information or association an object's classification label is intended to protect.

Smith proposed the following three "dimensions" of classification:

1) the data itself may be classified;
2) the existence of the data may be classified; and
3) the reason, or rule, for classifying the data may be classified.

Our approach for handling the first two dimensions of classification will be discussed below in this section. The third will be taken up in a later section after the concept of an integrity constraint is presented.

Classifying data in associations

We shall begin here by considering how data are classified. As Smith pointed out, what is classified is really certain associations. That is, the value 17.3 by itself is not intrinsically classified, but rather, e.g., the occurrence of that number as the length of a missile.

As pointed out by [Lunt89], many so-called aggregation problems are not true aggregation problems and can readily be solved through appropriate design of the data structures in the database. The true aggregation problem is when some collection of facts has a classification strictly greater than that of the individual facts forming the aggregate. To qualify as an aggregation problem, it must be the case that the aggregate class strictly dominates the class of every subset of the aggregate; otherwise, the simple mandatory rules would suffice to protect the collection of information.

For many perceived aggregation problems, some of the essential facts are more appropriately classified at the higher, aggregate classification, as is typically the case when the aggregate represents an association among different entities. Such problems are readily solved by classifying the associations among entities separately and at a classification strictly greater than that of the entities themselves. Lunt's paper gives several examples in the context of the relational model; here we give some examples for the object model.

To begin with, suppose that employee names are unclassified and salaries are unclassified, but the association of a particular salary with a specific individual is classified secret. Putting the salaries in separate objects at the secret level would protect the association, but it would also, in effect, classify the salaries. Instead, consider the data design depicted in Figure 7. This scheme classifies the association, but makes the set of salaries available at an unclassified level for statistical purposes.

The protection against inference is not necessarily complete; see [Lunt89, Morg88] for discussions of additional inference threats that may not be addressed by this data design. For example, if there were only a few employees and salaries, and the ranking of the employees were known or could be guessed, the association could be reconstructed. In such cases, the preferred solution would be to classify one of the two entities in the relationship at the higher level, e.g., the salary objects.

Consider the design in Figure 8. In this design, the values for the employee variable Salary (which are object-ids corresponding to salary objects) can be visible, because it is common knowledge that employees have salaries. The salary dollar values themselves cannot be inferred. (In order for the salary object-ids to be entered into the unclassified employee objects, the empty Salary instances must have been created by an unclassified subject. The salary dollar values would be entered later by a secret subject.)

The approach taken here differs from the one proposed in [Keef88]. That model uses context constraints to handle inference problems in an object-oriented database. A context constraint specifies a classification set. Each object in a classification set allows itself to be read only when at least one other member of the set is still unread. The last unread object in the set must increase its sensitivity level to that specified in the context constraint before it is read. They suggest implementing the classification set as an ordered sequence of objects, in which an anchor object is responsible for alerting each first object that it is the first object in a context constraint. The object is also given the specification of the rest of the ordered sequence. Each object in the sequence then acts as an anchor object for the next object in the sequence, alerting it that it is included in the constraint and passing on the specification of the rest of the ordered sequence of objects.

A scheme of the above sort can work. However, the proposed mechanism is being used to enforce part of the mandatory security policy, so it cannot be implemented...
Figure 7. Design With Secret Associations

Figure 8. Design When Additional Threats Are Likely
using as untrusted layer over an ordinary reference monitor. For this reason, we do not pursue that approach.

Classifying object existence

Smith's second security dimension is object existence. The existence of an object should be hidden from low subjects if it was created by a high subject. Otherwise, there is a covert channel: a high subject can signal a lower one by creating and deleting new objects.

Existence of an object may be inferred by the appearance of its object-id in the value of a variable in some other object. Hence, we protect the existence of an object by ensuring that its object-id is not stored in a lower-level object.

Implications for object creation

An object-id is generated and stored as the result of the built-in Create method. When a Create message is sent to a class object, the object-id of the new instance is normally appended to the system-maintained "Instances" variable in the class object. However, if the Create message is sent by a higher-level subject, the new object-id should not be placed in the lower-level class object by the built-in Create method.

If the object-id of an instance created by a higher-level subject cannot be stored in the Instances variable of the class object, it must be stored somewhere else (in some high object), otherwise the object will be inaccessible. There is no "standard" place to put it, so this task is left to the invoking subject. Create should return the new object-id as a value to the invoking subject, which can then store it in a variable in some object at its own level.

For example, as shown in Figure 9, the secret skill "spying" does not appear as a skill of the unclassified employee Jane Doe. However, the skills S22 ("driving") and S99 ("spying") indicate that they are skills of E123 ("Jane Doe").

Application-specific variables such as Has-skills are maintained by untrusted user-supplied methods. If a method attempts to put the object-id of the secret skill S99 into the Has-skill variable of the unclassified object E123, it will fail. This is because, if that subject has read the object-id S99 from a secret object, it must itself be at least secret, so enforcement of the *-Property will prevent it from modifying E123.

The existence of a classified object is not always classified at the level of the object. The existence of a classified object may be unclassified, or perhaps classified at a lower level. In general, the existence of any object is classified at the level of the creating subject. An example of a secret object whose existence is unclassified was illustrated in Figure 8 - the salary object instance Z292.

As these examples illustrate, the mere fact that a new object is at a higher level than its class object does not determine whether its existence should be hidden from the lower level subjects. Whether to hide an object's existence is a policy decision that can be implemented by adjusting the level of the creating subject.

Object-ids and polyinstantiation

Because we use globally unique object-ids instead of user-chosen object names to identify objects, polyinstantiation does not occur in our model. However, ensuring object-id uniqueness is not easy. When a low-level subject creates an object, the untrusted object-layer process will not know about object-ids of objects created by higher-level subjects.
One solution is to use a pseudo-random number generator, so that object-ids are very unlikely to be duplicated. A user could supply a freely-chosen mnemonic prefix to the pseudo-random suffix in order to obtain more meaningful object-ids, and unclassified objects could, by convention, dispense with the pseudo-random suffix.

The database designer can always give each object a "Name" variable to hold a descriptive text string. This string could not be used directly to refer to objects, but it could be printed in reports or used as the basis of a search. The "Name" variable could appear in place of object-ids in the displays generated by the user interface or by applications. Selections based on the name display would then be unambiguous, because the interface or application would be able to translate them back to the object-id.

Although we can avoid real polyinstantiation, a similar effect may occur in the user-assigned object "names" or values of identifying instance variables. For example, consider the Employee objects shown in Figure 10.

![Figure 10. Apparent Polyinstantiation](image)

Object C113 corresponds to the employee whose Employee-id (assigned by the employer) is E2345 and whose Name is Gary Smith. Note that object C113 is classified secret. Now, consider what happens if an unclassified user wants to add another Employee instance with Name Rae Burns. This user does not see object C113 and thus does not know about employee Gary Smith with employee-id E2345. Thus this unclassified user may re-assign employee-id E2345, as shown in Figure 10. Notice that there is no polyinstantiation (object-ids are unique), but there is apparent polyinstantiation (the value of the variable Employee-id is not unique).

One way of avoiding apparent polyinstantiation is through the use of classification constraints (which will be discussed later). In the example above, a classification constraint could be used that requires all instances of object Employee to be unclassified.

Because all references to an object use the unique object-id rather than the value of a variable, there is no "referential ambiguity" [Gajn88] in our model.

5. Constraints

Constraints play an important role in databases, knowledge bases, and expert systems. In this section we discuss the general concept of constraints, including both integrity constraints and classification constraints. Next, we consider how constraints fit into our secure object model.

At this point, we address Smith's third security dimension: classification of the rules used to classify data. This reduces to the question of how to classify the objects in which classification constraints are stored.

**Integrity constraints** are user-specifiable rules that restrict the values that an object's variables may take. They may restrict the allowable set of values for a single variable, as in a type check, or they may constrain the relationship between values of variables. These relationships may be between the variables of a single object or may span objects.

**Classification constraints** are user-specifiable rules that constrain the allowable classifications of the objects. They may be used to constrain all objects of a given class to have the same classification (these have been called "type-dependent" constraints). They may assign a classification to an object that depends on the value of one of its variables or on the value of a related object's variable. Or they may assign a classification to an object that depends on the classification of a related object.

**Constraints as methods**

In an object-oriented system, a constraint check can exist as a method that is invoked automatically when any change is made to a variable in an object. Applicable constraint methods will be found both in an object and its class ancestors. Very often, there will be several constraints applicable to an object, defined in various ancestors. It is up to the system to look up the class hierarchy for all constraint methods, and generate the
messages to invoke them. To make this possible, constraint methods must be identified as such. For each object, there could be two lists of constraint messages, one for integrity constraints, and one for classification constraints, as shown in Figure 11. Methods defining constraints, like other methods, are part of the object.

![Figure 11. An Object With Named Constraints](image)

**Integrity constraint example**

Figure 7 showed the object Employee-Salary. This object has variables Employee and Salary. The following two user-specified integrity constraints could ensure that any value assigned to the variable Employee represents an employee that exists in the database, and also that any value assigned to the variable Salary corresponds to a salary that exists in the database:

```
must-be-employee:
  Send Self.Employee, Getvar(Class)
  Return (value = Employee)
must-be-salary:
  Send Self.Salary, Getvar(Class)
  Return (value = Salary)
```

This is an "advisory" constraint because it returns a boolean (the result of an equality test). A false result implies that the constraint has failed, and the object layer will post a warning. Since constraints are just user-supplied methods, they might be designed to take some specific action instead.

**Classification constraint example**

Figure 7 also shows the object Employee. Suppose we want to require that all employees be unclassified. (Note that the Hierarchy Property will allow employee instances to be classified higher than unclassified.) We can define a classification constraint to do this for us. Like integrity constraints, an object's classification constraints are methods for user-specified messages. Here is an advisory classification constraint for the object Employee:

```
unclass-employee:
  Return (Self.Level = unclassified)
```

(We are assuming that an object's level is accessible as though it were in a variable named "Level.")

**Constraint interactions**

The data designer must understand the interaction between the defined classification constraints and integrity constraints. To illustrate, consider the following example. Recall that the constraint "must-be-salary" on the variable Salary requires that the salary value must be the object-id of a Salary object that exists in the database.

But now consider what happens if some of the Salary objects are classified higher than secret. We might have a classification constraint that requires that salaries greater than $1,000,000, be classified top secret, for example. Then a secret subject that modifies the secret variable Salary for the secret object Employee-Salary cannot enter an object-id for a salary exceeding $1,000,000. This may be the behavior desired, but it is important that the data designer understand this.

**Classified classification constraints**

Here we discuss the third security dimension. We consider how to handle the data in our model when its classification rule is itself classified, and when it is unclassified.

In the situation shown in Figure 12, the classification constraint is unclassified. If it is okay for uncleared users to know this rule, then this is a suitable way to handle value-dependent classification constraints. However, it may be considered undesirable for uncleared personnel to know that there are salaries classified at the secret level, or that there are expected to be salaries higher than $500,000.

The fact that there are expected to be salaries higher than the unclassified $100,000 threshold is harder to hide, but, as we shall see below, not impossible.

The data design shown in Figure 13 is intended to hide the classification rule, but still allow it to be enforced. It subdivides the Salary class into three subclasses, for the three levels of Salary. Each subclass has its own
classification constraint, and is classified at the level of its instances. An uncleared user will not be able to extract any information from the higher-level subclass objects.

![Figure 12. Value-Dependent Classification Constraint](image)

If necessary to hide the existence of higher-level salaries, a higher-level user can create the corresponding-level subclasses, so that they will not appear in the list of instances of the salary object. An attempt to store a salary over $100,000 in a Staff-sal instance will still cause an error, but the user cannot deduce that there are higher salaries - perhaps all salaries are required to be below $100,000! The class structure visible to the uncleared user suggests that there may be classified instances of Salary, but there is no indication why they might exist, what their values might be, or at what level they are classified.

In Figure 13, each object Staff-sal, Z333, and Z444 has an integrity constraint (not a classification constraint) defined of the form \( A < \text{salary-amount} \leq B \), for appropriate values of \( A \) and \( B \).

Each subclass might also have a classification constraint defined for it that requires each instance to be no higher than the appropriate classification. That is, each instance of Staff-sal would be constrained to be unclassified, each instance of Z333 would be confidential, and each instance of Z444 would be secret.

Even if the various constraints described above are not implemented, however, there will not be any security compromise. The only problem will be that certain data might end up in unexpected locations and be inaccessible to authorized users. For example, if a secret salary object were created as an instance of Staff-sal by a secret subject (only a secret subject would actually know what the secret salary actually was), no reference to it would appear in the Instances variable in the Staff-sal object. As a result, unclassified subjects would still be unaware of it, and secret subjects would fail to find it in the expected place under Z444.

6. Conclusions

By taking an object-oriented approach, it is feasible to implement a multilevel database system. Starting with a conventional mandatory security kernel, an object system layer can be superimposed on it, and its features can be employed to create a data base and the associated computational apparatus. If each object has a single security level, the object system layer can be untrusted with respect to the kernel.
The object system layer presents a user interface consisting of several primitive operations, i.e., the system calls and system messages for use by methods. These primitives respect and enforce a set of security properties, which will protect the user from kernel violations as long as the user cooperates. If a user program bypasses the object layer, it simply loses the benefit of object-oriented services, while it is still constrained by the kernel's security enforcement.

The policy of a single security level for each entire object permits the data designer considerable power and flexibility to classify data associations as freely and independently as the consistency of the classification rules will allow. Classification policies for the existence of data and for the rules for classifying data can be supported. Several examples of realistic policy constraints of all these types were given.

Integrity constraints and classification constraints can be implemented in an object-oriented system as methods, and stored in the most general class objects for which they are defined. The purpose of classification constraints is not to prevent compromise; they are aimed at preventing overclassification and other operator mistakes that render data less easily available than it should be.

Acknowledgements

This research was supported in part by the Office of Research and Development of the National Computer Security Center, which funded MITRE through contract F19628-89-C-0001, and in part by the U. S. Air Force, Rome Air Development Center (RADC), which funded SRI through subcontract R009406 with IITRI (U. S. Government contract F30602-87-D-0094). The authors gratefully acknowledge the NCSC and RADC for making this cooperative work possible.

References


