Computers & Security, 8 (1989) 517–533

Refereed Article

SODA: A Secure Object-oriented Database System

T. F. Keefe[1], W. T. Tsai[1] and M. B. Thuraisingham[2]*

[1] *Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, U.S.A.*
[2] *Honeywell Inc., Corporate Systems, Development Division, Golden Valley, MN 55427, U.S.A.*

This paper describes a security model for object-oriented systems. The model supports a flexible data classification policy based on inheritance. The classification policy allows for a smooth transition between rigid classification rules and unlimited polyinstantiation. The security model treats the data model as well as the computational model of object-oriented systems allowing more flexibility. This model trades an increase in complexity for a more flexible security model.

Keywords: Secure database, Data classification, Object-oriented model, Security entities.

1. Introduction

Recently much research has been devoted to the design of multilevel secure relational DBMS [5, 6, 9, 14]. The relational data model is well defined and generally applicable to a wide range of data-modelling problems. For some problem domains involving multimedia DBMS and CAD/CAM, object-oriented systems present a more suitable data model and have become popular in these domains.

Object-oriented systems began as programming systems and are only now dealing with issues such as transactions and controlled sharing of data [7, 17]. Resolving these issues paves the way for more useful object-oriented DBMSs and generates a need for security.

Object-oriented DBMSs support a powerful computational model which can be used to express applications. The relational algebra does not deal with the subject of updating or creating new relations even though most relational DBMSs do provide this capability. The fact that the object-oriented model includes an explicit computational model allows the incorporation of the computational model into the security model.

Providing security for an MLS/DBMS-using query modification is discussed in ref. [13]. Query modification is suited to systems with a simply defined set of composable operators. Object-oriented systems provide a fairly large set of primitive operators. These operators can be combined procedurally to form functions with complex semantics. Query modification does not consider the problem of updates. Data modification and creation are common operations in object-oriented programs and an important issue in our model.

Security in object-oriented systems is discussed in ref. [16]. Objects, object schemas, classes, instance variables, methods, composite objects and messages are all given classifications. The paper develops a

*Present address: The MITRE Corporation, Burlington Road, Belford, MA 01730, U.S.A.

In an object-oriented system everything is represented as an object. An object is made up of private state information and a set of actions which represent the interface to the object. The state information is represented as a set of instance variables whose values are objects. The actions defined on objects are called methods. A method carries out its action by sending messages. A message consists of a method selector, which is the name of the method to be invoked, followed by a list of objects used as arguments. Sending a message to an object causes a method to be executed. Objects are passive entities which store information. A method is also passive. It represents a function which can be performed on an object. A message sent to an object creates a method activation. Method activations are active and perform computations.

Primitive objects represent their state directly; examples of these primitive objects are numeric values, strings and identifiers. Primitive methods represent actions carried out directly without sending messages; examples are adding numeric values and indexing arrays.

Each object has a type or class it belongs to. All objects in a class are equivalent computationally. Each may have a different state but the type of computation which can be performed on an object is uniform throughout the class. The class defines what methods are available in instances of the class and what instance variables are included in the instance objects. The class of an object is also an object. A class object creates new instances of its own type. A class object defines a type by specializing other types and defining additional behavior. The refined types are referred to as its sub-types. An object inherits methods and access to instance variables from its class object and each super-type of the class object all the way up the inheritance lattice to the root, OBJECT.

Methods are specified such that only data contained in the object receiving the message are modified directly. A method activation has no knowledge about the states of other objects and it cannot directly affect other objects.

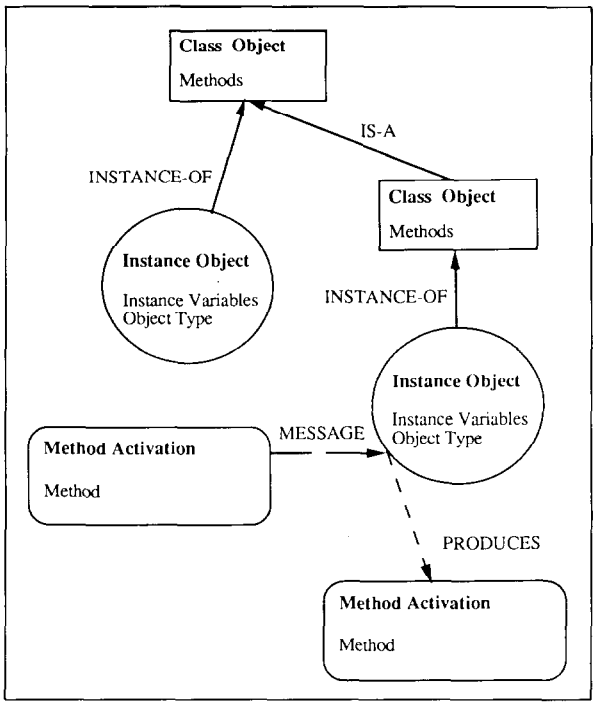

Fig. 1. Concepts of object-oriented model.

Figure 1 illustrates some of the concepts discussed above.

4. Security Model

This section introduces a security model with several unique features. First, it is proposed for use in object-oriented systems.

Secondly, the protected passive data in our model are instance variables and objects. Access to them is arbitrated by the TCB. Another common choice for protected objects is a message [3, 20]. In this approach each object is given a classification. To enforce the *-property [2], messages can only be sent to objects of an equal or higher level and responses can only be received from objects of a lower or equal level. This technique is well suited to distributed systems where the message passing is part of the system kernel. However, this approach has several problems. First, the need to enforce the *-property ensures that only objects of the same

set of application-independent properties which describe relations between entity classifications. It defines integrity rules for these entity classifications. It is difficult to determine which entities in the model act as protected objects and which act as subjects. This makes the security properties of the model difficult to evaluate.

Mandatory security is investigated in ref. [18]. Security is enforced with a combination of compile-time and run-time checks. The security model classifies variables as having a fixed or indeterminate sensitivity level. The indeterminate levels are meant to deal with indeterminate information flows and must be checked at run-time. The security model does not support automatic object classification rules needed in a DBMS.

The use of a sensitivity level range to control the sensitivity levels of data in a container is used in SEAVIEW [15] and TRUDATA [14]. TRUDATA uses it to limit vulnerability. SEAVIEW uses it to limit polyinstantiation where it is augmented with security constraints for tuple labelling. In our approach these constraints are the security constraints. Constraint satisfaction is enforced as part of mandatory security. Constraint satisfaction guides the action of the reference monitor.

We propose a security model for a multilevel secure object-oriented database (SODA) with the following advantages. It is posed in terms of an object-oriented model, and it enforces information containment and security label integrity. The model covers the computational model as well as data access and classification. This allows the interaction of the two concerns. The classification level of a computation is adjusted based on its clearance level, the data it has access to, and classification constraints enforced on data it wishes to create.

The model supports a data classification policy which fits naturally into the object-oriented model. The classification technique is based on the inheritance lattice which allows a natural way of expressing security constraints. The classification method allows the classification rigidity to be tuned on a class-by-class basis. This provides for wide variations in labelling requirements.

2. Multilevel Security

Multilevel secure computers protect objects classified at more than one level and allow sharing between users of different clearance levels. Objects are labelled with their sensitivity levels. Subjects are associated with clearance levels. A multilevel secure computer arbitrates all access of objects by subjects. The arbitration is carried out by the reference monitor according to a security policy.

MLS/DBMSs must deal with large numbers of objects, interrelated in complex ways which have semantic meaning. This causes several problems. The first is efficiency. Large numbers of objects can cause a large burden on the access monitor. Secondly, all of these objects must be classified in a complete and consistent way. The third problem is representing and manipulating objects containing data of multiple sensitivity levels. The interrelations of the data and their semantics lead to an inference problem. Inference occurs when information which can be retrieved from the database allows other data to be deduced. Inference provides a flow of data which is not arbitrated by the reference monitor.

3. Object-Oriented Systems

This section gives a brief background on object-oriented systems. There is a wide variation in what is meant by "object oriented." Most of our interpretation comes from SMALLTALK-80 [11]. Variations on this model are given in ref. [19]. The object-oriented model began as a programming system. Our definition of an object-oriented system also stems from our desire to incorporate database considerations such as schema evolution, transactions and controlled sharing of data. Our understanding of these issues comes from refs. [1], [7] and [17].

level can send and receive messages. Secondly, messages are sent to the class to carry out actions on the instances of the class. When the instances of the class have different sensitivity levels, the class objects must either be trusted to handle multilevel objects, or there must be a version of the class for each sensitivity level. These models also have difficulty with shared variables, *e.g.* variables in the class which are available to all of the instances. Unless the instance objects and the class share the same sensitivity level, class variables will either be unreadable or unwritable. These problems stem from the fact that a method when executing in this model has access to all of the object's data.

Data classification in our model is based on inheritance. Each object in a class shares the same classification constraint. Each may have a different sensitivity level but all satisfy the common constraint. The classification constraint of a class is inherited by its sub-types. The constraint may be redefined in the new class but only by the system security officer.

Attaching classification constraints to classes and allowing subclasses to inherit classification constraints simplifies data classification. It uses the existing structure of the system as the basis for classification. The inheritance of constraints ensures that they only need to be specified when there is a change.

Data classification specifies a range of sensitivity levels which can be assigned to an object and is strictly enforced by the model. This allows the classification rigidity to be adjusted on a fine scale.

Each method activation is independent. The current classification level of the method activation represents the sensitivity level of data which the activation has read. Once the activation finishes, this information disappears and control is returned to the caller acting with its original authority.

A fundamental concept of the model is that since everything in the object-oriented model is an object, there is good assurance that no portions of the system are unprotected. The view that everything is an object is interesting. It gives a simple understanding of the system but does not tell the whole story. To really understand the object-oriented model and our security model it is necessary to understand the practical limitations of this view. For example, all objects store their state in instance variables in the form of other objects. This view leads to an indefinite recursion. Each object stores its state in terms of other objects and no values are actually stored. The missing concept is the primitive object. These objects represent their state directly without using other objects and therefore terminate the recursion. Another example of a special case involves methods. Methods carry out their actions by sending messages. This is also a problem; each method sends messages but none performs a computation. Primitive methods are the answer to this problem; they perform actions directly and send no messages. Another concept of object-oriented systems is that each object is an instance of another. The solution here is the object Metaclass which is an instance of itself.

These discontinuities point to special cases in the object-oriented model and special cases in our security model. The three cases described above must be considered as special cases of the security model as well. First, all objects in the model are labelled except primitive objects which are not labelled at all. They represent basic data elements which have little meaning out of context. Secondly, all methods have an independent current classification level except for primitive methods. These methods inherit the current classification of the method activation which called them. Each object inherits classification constraints from its class. Metaclass is an instance of itself and must provide a built-in classification constraint.

4.1 Security Entities

This section identifies the security role played by each entity in the object-oriented model. The portions of the object-oriented model discussed are as follows: classes, primitive objects, objects,

instance variables, messages and method activations.

Classes
A class represents the type of its instances. The class defines the methods and instance variables its instances have. It also defines class variables which are available to all instances. Classification constraints are recorded in the class and apply to all instances.

Primitive Objects
These objects are the basic elements which all objects can be broken into. They are not assigned sensitivity levels. They represent basic data elements which have little meaning out of context.

Objects
Each object can have a sensitivity level. This sensitivity level restricts access to the whole object rather than just one instance variable. This classification method can be used to protect the objects regardless of where the reference is from.

Instance Variables
Each instance variable can have a sensitivity level. The value of the instance variable is the protected data. This classification can be used to protect primitive values and associations between objects.

Messages
A message is sent on behalf of, and represents a subject. It is sent to an object requesting execution of a selected method with the authority of the security subject which the message represents. Messages are conceptually labelled with two security classification levels. The first is the clearance level of the user. The second level is the current security classification of the originating method. These two levels act as an upper and lower bound on the authority of the new method activation.

Method Activations
Method activations are the only active entities in the model. Each method executes in a separate context described by an activation. The execution is carried out by sending messages to objects. Primitive methods are carried out directly by the method activation without sending messages.

4.2 Labelled Entities

This model supports two types of labelled entities, objects and instance variables. Each object can support only one type. Either the object is labelled or its instance variables are, but not both. Object labelling associates a sensitivity label with an object. This label is used to arbitrate access to the entire object regardless of the context in which it is used. This type of labelling is illustrated in Fig. 2.

An object with instance variables labelled has a sensitivity level for each instance variable of the object. The label controls access to the contents of the instance variable. This labelling protects the associations between instance variables. Since an object can be referenced by more than one object, variable labelling allows a classification to be associated with each context in which the data are seen. Variable labelling is illustrated in Fig. 3.

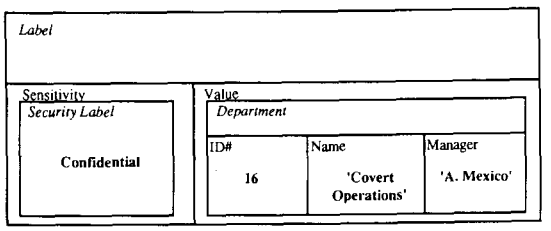

Fig. 2. Object labelling.

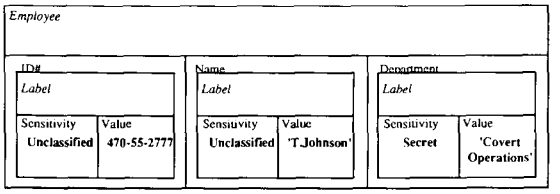

Fig. 3. Instance variable labelling.

4.3 Active Entities

In this model the active entity is a method activation. A method activation is similar to a function invocation. It is the result of a message being sent to an object. A method activation on its surface is similar to a process with no state. The objects accessed by the method activation are protected. Information encoded in the execution state of the program is not protected. Each method activation has its own state and is capable of encoding information. To control the information in the state of the activation, a method activation is assigned a current classification level. This level records the least upper bound of all information the method activation has read or has access to. This information can come directly from reading an object or indirectly from what the calling method has read. The clearance level of the method activation comes from the associated user. The clearance level serves as an upper bound on the current classification level.

TRUDATA [14] associates a range of authority with subjects. This range is changed by the user but is not modified dynamically.

4.4 Data Classification Policy

A classification constraint consists of two parts. The first part specifies the type of labelling, object or instance variable. The second part consists of sensitivity level ranges for each labelled object. In the case of object labelling there is one range for the object itself. For instance variable labelling there is one range for each instance variable. The range specifies allowable limits on the sensitivity level of the protected object.

Data classification is based on the inheritance lattice of the system. Classification is determined in two ways, type and specialization.

(1) The classification constraints for an object are obtained from its class.
(2) The classification constraint ranges of a new class are inherited from its super-type.

The inheritance lattice provides a natural way of categorizing objects into semantically meaningful groups. From (1), each class is assigned a classification constraint which applies to its instances.

The classification range of a subclass is inherited from its super-class, (2). The labelling type and ranges for corresponding labels are the same. In the case of variable labelling the sub-type may declare additional variables. These variables are unrestricted, *i.e.* they have a range of [SYSTEM LOW, SYSTEM HIGH]. This allows users to create new classes. They are not however allowed to modify constraints for the new class. The modification of constraints is done by the system security officer.

This classification mechanism can be used to hide associations between objects. Consider a class named "Flights" whose instances contain information describing the destination and cargo of flights. "Flights" is classified using variable labelling with "Destination" and "Cargo" both classified [SECRET, SECRET]. To correlate the destination and cargo for a flight, a user must have a SECRET clearance. This constraint classifies the path to the information, but not the information itself.

Classification can be based on the value of an instance variable. It requires the creation of a subtype for each classification group. For example, to classify information about flights with a destination of Iran as SECRET, a sub-type of the "Flights" class is created called "FlightsToIran." All instances of the class "Flights" are classified [UNCLASSIFIED, UNCLASSIFIED]. The instances of "FlightsToIran" are classified [SECRET, SECRET]. In this case it is undesirable to have a class which is visible to UNCLASSIFIED users called "FlightsToIran." To hide the class object, the class "FlightsToIran" itself can be classified SECRET by the System Security Officer.

Limits set by the classification constraints are enforced by the TCB. If a sensitivity level for a new object cannot be found which both satisfies the classification constraint and maintains information containment, the object will not be created. For

example, consider a method activation which has read SECRET data and tries to create an object which is constrained to be [UNCLASSIFIED, UNCLASSIFIED]. The object will not be created. The object created must be labelled SECRET to avoid writing down but this does not satisfy the constraint.

Sensitivity level ranges provide a conceptually simple way of classifying data. The technique provides a simple enforcement which can be carried out in the TCB. The task of assigning constraints and their verification is done by a separate trusted application, the sensitivity labeller.

4.5 Mandatory Access Restrictions

This section describes mandatory access restrictions. The restrictions define a set of allowable object accesses. There are three parts in the model. The first part describes which object accesses are allowed based on the sensitivity level of the object and the current classification level of the method making the request. The second part shows how these restrictions are modified to support polyinstantiation. The last section describes assignments and allowable changes to security classification levels of method activations.

4.5.1 Object Access

The discussion of object access is simplified by considering labelled object access. Figures 2 and 3 diagram the role of labelled objects for object and instance variable labelling. For the following discussion consider a method activation executing with a clearance level of L_{Sclear} and a current classification level of $L_{Scurrent}$ accessing a labelled object stored in a slot with a sensitivity constraint range defined by $[L_{bottom}, L_{top}]$. The method activation is allowed to

Rule 1.1

Read the value of a labelled object with sensitivity level L_O if $L_O \leqslant L_{Sclear}$. An unreadable object returns nil. (Nil is the value stored in an instance variable when an object is created. It means that no object is stored there.)

Rule 1.2

Create and store in the constrained slot a labelled object with sensitivity level $L_O = L_{Scurrent}$ if $L_{bottom} \leqslant L_{Sclear}$ and $L_{Scurrent} \leqslant L_{top}$; otherwise reject the update and inform the user.

Figure 4 represents the range of object sensitivity levels which a method activation can read. The dashed line represents the fact that the current classification level will be increased up to L_O (Rule 2.3). Figure 5 illustrates the conditions in which an object can be written. The condition is described in terms of the method activation's current classification and clearance level, and their relation to the classification constraint of the object. The dashed line illustrates the range in which the current classification will be raised (Rule 2.3). This is to satisfy the requirement that the data is classified above L_{bottom}. The clearance restriction disallows writing up. This allows polyinstantiation to be avoided when the classification range of the data is degenerate, i.e. $L_{bottom} = L_{top}$. The restriction on the current classification level maintains the *-property.

Rules 1.1 and 1.2 by themselves do not ensure the simple security property or the *-property [2] since the current classification levels of methods are

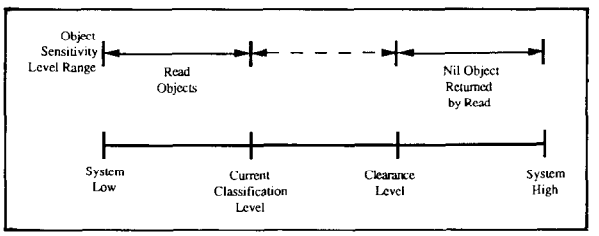

Fig. 4. Labelled object read rules.

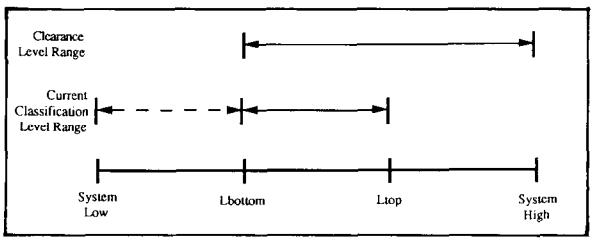

Fig. 5. Labelled object modification rules.

allowed to change and these changes have not yet been defined. The maintenance of these properties can be ensured only after examining the modification policy for method classification levels. The security properties enforced by the model are discussed in Section 5.

4.5.2 Polyinstantiation

The multiparty update conflict is noted by Denning *et al.* in [4, 5] and is the problem which polyinstantiation addresses. The conflict arises when low-level users unknowingly attempt to overwrite higher level data which are invisible to them. If the write is rejected a storage channel is created. If it is allowed, high level users have no way to protect their data.

Until now, we have ignored this problem. If the classification range for an instance variable is degenerate, *i.e.* $L_{bottom} = L_{top}$, this is adequate. In this case, the slot in which a labelled object is stored has one allowable sensitivity level. The model requires that to store a labelled object the user's classification level be equal to the level of the slot (Rule 1.2). Now if an update is rejected it is so because the user's classification does not satisfy Rule 1.2. This rejection cannot cause a covert channel.

When L_{top} strictly dominates L_{bottom} each slot has a range of sensitivity levels. In this case polyinstantiation is used to eliminate the conflict. Polyinstantiation allows each slot to contain an object of each sensitivity level between L_{bottom} and L_{top}.

Polyinstantiation is added to the model by allowing each instance variable to contain a collection of labelled objects. This collection is called a polyinstantiated set. Each element of the set is a labelled object. The sensitivity level of each element is unique in the set. When an element is added with the same sensitivity level it replaces the original member.

Rules 1.1 and 1.2 of the previous section can be revised to include polyinstantiation as follows:

Rule 1.1'
Read the value of all labelled objects in the polyinstantiated set with sensitivity level L_O if $L_O \leqslant L_{Sclear}$. Nil is returned when no objects are readable.

Rule 1.2'
Add a labelled object to the polyinstantiated set with sensitivity level $L_O = L_{Scurrent}$ if $L_{bottom} \leqslant L_{Sclear}$ and $L_{Scurrent} \leqslant L_{top}$; otherwise reject the update and inform the user.

Polyinstantiation introduces several problems. First, the interpretation of the data becomes more complex. Many simple values become sets of values when polyinstantiation is used. This complicates understanding of the data and makes applications harder to develop. The second problem deals with replacement. Normally, when a labelled object is stored in an instance variable it replaces the labelled object that is already there. With polyinstantiation this is not necessarily the case. It may just add another value to the set.

4.5.3 Method Activation Security Levels

A method activation executes with a security classification level $L_{Scurrent}$ determined by two quantities. The first is the clearance level L_{Sclear} of the user. The second quantity is the current security classification level $L_{Soriginator}$ of the method activation which started this method by sending a message. Below is a set of rules determining the current security classification of a method activation.

Rule 2.1
The login method begins execution with classification level $L_{Scurrent}$ = SYSTEM LOW, (the method begins when the system is started and has no originator).

Rule 2.2
A method activation begins with a classification level $L_{Scurrent} = L_{Soriginator}$.

Rule 2.3
If a labelled object with sensitivity level L_O such that $L_{Scurrent} \leqslant L_O$ is read or added to a polyinstan-

tiated set the current classification level of the method will be increased to the least upper bound of $L_{Scurrent}$ and L_O, *i.e.* $L_{Scurrent}$ *lub* L_O.

Rule 2.4
The object returned by a method activation is labelled with the method activation's $L_{Scurrent}$.

These rules ensure $L_{Scurrent}$ will always dominate the level of the information available to the activation. The current classification level will begin at the lowest possible level to allow the method activation the most flexibility possible.

One point that is not explicitly stated is that the current classification level pertains to a method activation. When the method returns, the information encoded in the state of the activation disappears. The caller then resumes with its original current classification level.

An example of how these rules work is shown in Fig. 6. Each line shows the authority range [current classification level, clearance level] for a method activation. The action which causes the change in authority level is described in italicized text between the lines.

The "Login Activation" begins with a current classification level of System Low (Rule 2.1). It sends a message to another object which starts execution of

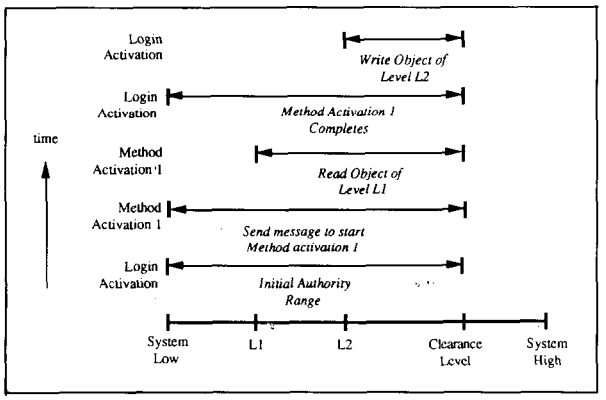

Fig. 6. Method activation rules.

"Method Activation 1" (Rule 2.2). "Method Activation 1" reads an object with a sensitivity level L1. In the process of doing this its current classification level is raised to L1 (Rule 2.3). When "Method Activation 1" returns, the "Login Activation" resumes with its original current classification level of System Low. Finally, it writes an object with constraint [L2, L2] and subsequently raises its current classification level to L2 (Rule 2.3).

5. Model Properties

This section discusses properties of the security model. We do not attempt formal proofs of these properties but use informal arguments to demonstrate them.

5.1 Simple Security Property

The simple security property states that a subject with a clearance level L_S is not allowed to read an object with a sensitivity level L_O such that $L_O > L_S$. In the notation used in this model, a subject with clearance level L_{Sclear} is not allowed to read an object with sensitivity level L_O if $L_O > L_{Sclear}$. This is ensured by Rule 1.1′.

5.2 *-Property

The *-property states that a subject with current security classification level L_S cannot write objects with sensitivity level L_O such that $L_S > L_O$. The proposed model does not allow information to be written down. Evidence is based on two facts. First, the current classification level dominates the sensitivity level of all information accessible to the method (Rule 2.3) and secondly the method activation cannot write or create objects such that $L_{Scurrent} > L_O$ (from Rule 1.2′, *i.e.* $L_O = L_{Scurrent}$).

The information accessible to a method activation can come from its instance variables, information about its calling context and information available about the existence of unreadable objects. The information accessible from instance variables is covered by Rule 2.3, $L_{Scurrent}$ dominates the sensitivity level of all objects which have been read by the method activation.

Information read by the calling method activation can be passed on by the mere fact that the method is executed. For example

SecretObject IFTRUE: [UnclassifiedObject AT "Answer" PUT: True]

This expression sends the conditional message IFTRUE: to SecretObject. The action taken by this message is to execute the block which is the argument to the message if SecretObject is True. The block consists of the expression in brackets. This expression will associate the value True with the index value "Answer" in the dictionary UnclassifiedObject. (See Appendix A for a brief description of the SMALLTALK syntax.)

The true block is only executed if the SecretObject is true. Once called, the true block implicitly knows the value of the SecretObject. Therefore the method activation must start execution at the SECRET level (Rule 2.2). $L_{Scurrent}$ dominates the level of its caller and thus the sensitivity level of all information it has access to. This also addresses the problem of information being transferred when the SecretObject is False. The program cannot store UNCLASSIFIED information when the vaue is True and it does not attempt to when the value is False. This program will not write down information about SecretObject.

Information about the existence of objects is given to a method activation when it can distinguish between null objects and objects it is not allowed to read. This transfer of information is disallowed by Rule 1.1′, (*i.e.* Nil is returned when no objects are readable).

5.3 Message Safety

Sending and receiving messages does not violate mandatory security. This will be discussed in two parts. Sending a message to begin execution of a method is discussed first, followed by a discussion of the object returned on completion of the method execution.

A message is sent by a method activation, M_1, to a passive object creating another method activation, M_2. M_1 executes with a clearance level of L_{Sclear} and a current classification level of $L_{S1current}$. From Rule 2.2, the method activation M_2 is started with the same current classification level and the same clearance level. Any information which is transferred to the method activation M_2 by beginning its execution is acceptable since both methods execute with the same current classification level.

Rules 1.1′, 1.2′ and 2.3 place the upper bound for $L_{S2current}$ to be L_{Sclear}. The object returned by M_2 is labelled, (Rule 2.4). Thus the upper bound on any object returned to M_1 by M_2 is also L_{Sclear}, by Rule 1.2′. This object can always be read by M_1 because of Rule 1.1′ and the fact that the same level for L_{Sclear} applies to both method activations. Security can only be violated if M_2 can return higher level information to M_1 and M_1 does not increase its current classification level to match that of M_2. If M_1 attempts to read the object returned, it will raise its classification level according to Rule 2.3 and security will not be violated. If M_1 does not read the object it will not receive the information and security will again not be violated.

5.4 Covert Channels

This section will discuss covert channels. We will consider storage channels and then timing channels.

5.4.1 Storage Channels

We will discuss two aspects of our model which seem to allow storage channels. The first aspect is the automatic change in current classification level predicated on the existence of invisible data. Rule 2.3 allows the change of a method's security level conditioned on the existence of an object with a higher classification level. This can allow a covert storage channel if another method activation can monitor the classification level of the method activation. A method activation will maintain some state information. This will include the temporary variables of the method activation, the program counter and a pointer to the calling method activa-

tion. This information is only available to the method activation itself and cannot be used to form a storage channel.

The second aspect is alerting the user when an update does not satisfy classification constraints. Rule 1.2′ allows the user to be notified when the update is not allowed. This model requires that a user satisfies the constraint without writing up or down. This rejection is different from when an update is disallowed because there are higher level data stored there. A rejected update in our model means that the constraint cannot be satisfied. It does not indicate whether higher level information is stored there.

5.4.2 Timing Channels

There is a covert timing channel in this model. It arises when a method starts a new method activation which raises its security level and then returns to the calling method which retains its original security level. The new activation can read high level data and communicate it by modulating its execution time. This channel exists only between two methods one of which calls the other. It can channel information owing to the differential in current classification levels of the two methods. The channel does not involve multiprocessing, because the calling method is suspended until the called method returns.

Timing channels are eliminated when all methods of determining time are removed. Even if there are no functions which return real time, it can be determined in other ways. Gasser [10] mentions several ways of measuring real time, including countering characters received by a terminal, counting the number of disk accesses or by having a user enter the time from a stop watch. Although this is not a complete list, several of these methods can be ruled out. Since there is no conspiring user, the techniques relying on this approach can be ruled out. The fact that there is no multiprocessing will constrain the methods as well. We are examining methods of dealing with this channel.

6. Implementation

Now we will consider the difficulty in implementation. We will outline the design of the implementation and discuss the amount of trust required for each. The design is described in terms of objects and the methods which they support. The trusted components of the design are described in Table 1.

There are two important issues to consider in mapping this model to an implementation. The model supports a large number of labelled objects, and there are a large number of context switches implied by the model. We will consider mapping the model to a secure operating system, supporting protected data segments, and secure processes.

First, let us consider the mapping of objects in the model to protection objects. Each protected object is object or variable labelled. Object labelling places a classification on the object memory of the object. Variable labelling places a classification on each of the instance variables.

Object-oriented systems such as SMALLTALK use an object table to simplify garbage collection. The table has an entry for each object in the system which points to the object. All references to an object go through the object table. The object table and the object itself are both given a classification level in this mapping. In the case of object labelling the object table entry is classified with the same level as the object. It should be noted that in this case, instance variables do not require polyinstantiation. Object labelling leads to polyinstantiated objects.

For variable labelling, the object table entry is classified at the lowest level (*i.e.* the greatest lower bound) of all the instance variable classifications. For each instance variable with a non-degenerate classification range a polyinstantiated set of sensitivity L_{bottom} is inserted which contains the instance variable's value, where the classification constraint for the instance variable is $[L_{bottom}, L_{top}]$. The frag-

TABLE 1
Trusted design components

Object	*Description*
OBJECTMEMORY	This represents the interface between the OBJECT and the OBJECTMEMORY. It must allocate objects to the proper sensitivity memory segment. If an attempt is made to read or write above the current classification level in accordance with Rule 1.1′ and 1.2′, the SUBJECT is requested to raise its current classification level to allow the read or write.
OBJECT	When a method begins execution, the SUBJECT is instructed to stack its current classification level. The SUBJECT will unstack its previous current classification level when the method completes.
OBJECT CLASS	OBJECT CLASS supports the classification of OBJECT. It provides protocol for recording the security constraints and providing access to them by the object memory.
POLYINSTANTIATED SET	This object must support sets of labelled objects. It supports methods to insert and retrieve objects determined by their sensitivity level. This object hides the polyinstantiated sets from the user, but relies on the object memory for security.
SUBJECT	This object represents the subject. It supports requests to change the current classification level, stack and unstack its current classification level.
SENSITIVITYLABELLER	This class provides the interface for setting the classification of OBJECT CLASS by the system security officer. This is a trusted application.

mented objects will require an extra level of indirection to access the multilevel instance variables. Figure 7 shows the mapping for the objects from Fig. 2.

The object table is a multilevel structure in several protected segments. Objects and polyinstantiated sets can be grouped together into single-level files.

The second issue is the large number of secure context switches. The conceptually simple approach of placing each method execution in a separate process is secure. The large number of processes would be inefficient in a traditional operating system. There are ways the number of processes can be reduced. Starting a new process only when a method actually reads higher level data and returning when there is a need to lower the current classification level would be a first step.

The most profound improvement comes in executing in one process and modifying the current classification level of the process as required. The

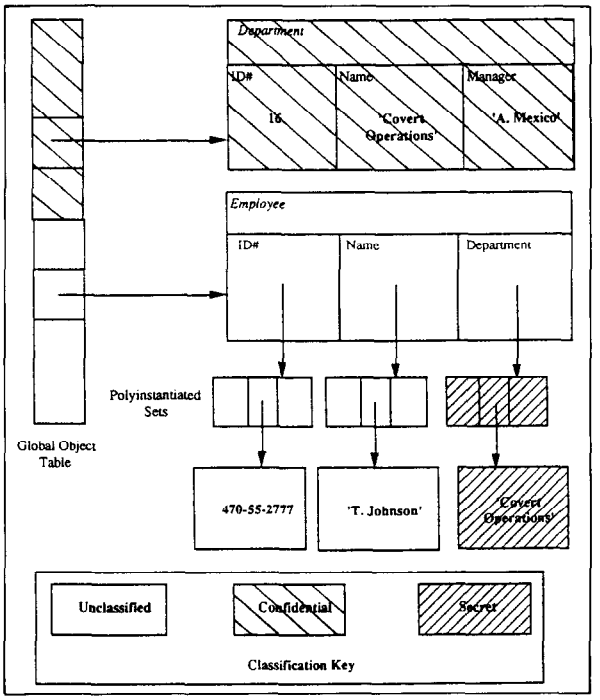

Fig. 7. Object allocation for labelled objects.

model described in ref. [2] provides a command to change the classification level of a subject. The command relies on a subject not maintaining state information outside the virtual memory segments which are protected.

This approach can be used in the implementation of this model. A method activation has a small amount of state information which is not protected in objects. Careful attention must be paid to removing this state information when the level is reduced, to avoid information being transferred to a lower level method activation.

7. Prototype

Floyd defines three classes of prototyping [8].

(1) Exploration.
(2) Experimentation.
(3) Evolution.

Exploratory prototyping is used when requirements are unknown or difficult to establish. Experimental prototyping is used to test a requirement's specification. An evolutionary prototype deals with requirements which evolve over time.

We have developed an exploratory prototype of the SODA security policy model. The prototype is written in SMALLTALK and was initially based on the model specified in [12]. The main purpose of the prototype was to test the feasibility and soundness of the model. The development and use of the prototype inspired refinements to the model which are described in this paper.

SMALLTALK was chosen as an implementation language because of the great similarity between the language concepts and the security model concepts. We attempted to implement the model by modifying the primitive methods which make up the TCB. In this way the restrictions of the security model are inherited by the entire SMALLTALK system. All accesses would then be arbitrated by these primitive methods.

We identified several primitive methods used for creating new objects, accessing instance variables and performing methods. These methods are inherited by all objects in the system and served as the basis for access control. These methods are described briefly below.

NEW
This method answers a new instance of the class which receives the message.

INSTVARAT: *index*
This method answers the contents of the instance variable denoted by *index*.

INSTVARAT: *index* PUT: *object*
This method stores *object* in the instance variable denoted by *index*.

PERFORM: *method* WITH: *argument*
This method starts an activation as if the message, "*method argument*" were sent.

Access control depends on a class called LabelledObjects. Instances have a value and a sensitivity level. These objects contain and protect other objects and instance variables. The method NEW inserts LabelledObjects when an object is created. The method INSTVARAT: above is modified to check these labels before retrieving the value.

Data classification for an object is inherited from its class. Object creation is also carried out by the object's class. In our prototype class objects include methods to support classification. They retrieve and modify the security constraints. The INSTVARAT:PUT: method queries the object's class to retrieve security constraints to determine if a value should be stored. The prototype includes an application called sensitivity labeller. This application provides a window interface for modifying security constraints. It is trusted but is not part of the kernel.

Each subject is modelled as a clearance level and a stack of current classification levels. The current

classification level is stacked before a method is executed and is unstacked when the method completes. This functionality is included in the PERFORM:WITH: method. Each subject has a window interface. Since SMALLTALK does not support multiple simultaneous users this gives us a way of simulating their interaction.

With polyinstantiation comes a new class PolyinstantiatedSet. Instances serve as repositories of labelled objects. The labelled objects in a PolyinstantiatedSet have unique sensitivity levels. The class supports methods for selecting visible objects, adding objects to the set, and methods to support heuristic selection strategies, such as "most sensitive member".

The prototype has been modified to follow the specification in this paper and now serves as an experimental prototype. An experimental prototype necessarily has a limited focus. This prototype limits its focus to the following issues

(1) Access restrictions.
(2) Security constraints.
(3) Polyinstantiation.
(4) Dynamic modification of subject security levels.
(5) Sensitivity labeller.

It does not address

(1) System architecture.
(2) Performance.

This focus determines the types of tests that can be carried out using this prototype. We intend to use the prototype to explore the impact of polyinstantiation on the user and the application developer.

The prototype clarified our understanding of the SODA model and helped us to refine it. The process of prototyping also gave support to the feasibility and soundness of the model. Our increased understanding came in part from a better understanding of the object-oriented model. This learning was especially useful because by building the prototype we learned precisely the parts of the object-oriented model which we needed to understand.

Dynamic modification of the current classification level was the biggest unknown in our model. In our original model [12] the current classification level was associated with a user and not the subject. In this model, the current classification level of a user cannot decrease. The prototype made apparent how restrictive this model was. The prototype helped us to explore the feasibility of associating the current classification with the method activation. This effectively allows the current classification as seen by the user to decrease. The prototype helped us to understand the risks and gather evidence of the approach's feasibility.

We originally thought it would be relatively easy to develop a secure database on top of SMALLTALK. We assumed that certain low level operations such as sending messages or accessing instance variables could be modified directly. We planned to modify these methods and embed the security model in the SMALLTALK system itself. This turned out to be difficult since these operations are carried out directly by the interpreter. Changing these functions requires modifications to the compiler or the interpreter. Regardless, the prototype was developed according to this model. The model can be exercised but the security it provides can be bypassed.

By trying to implement certain aspects of the model we found out immediately its complexity. While a simple function may have a complex implementation, if the complexity cannot be reduced, it is likely that the function is complex. One place where we have noticed a lot of complexity is in our implementation of PolyinstantiatedSets. The selection of elements to retrieve and replace is quite complex in the prototype. We believe more clarification of polyinstantiation in our model is needed.

Logical structuring of the model was improved from the prototype. The prototype makes apparent the inappropriate distribution of responsibilities. In the object-oriented model an object only has access to its own data. This encapsulation makes poor logical structure readily apparent. If responsibility is distributed unreasonably it will lead to a complicated implementation. An example of this is responsibility for initializing secure objects. This includes examining the classification and either enclosing the object in a labelled object or storing labelled objects in each instance variable of the object. We originally wanted the class to carry out this function since it had access to the classification information. It became clear in prototyping that this solution was not sound and the function became the responsibility of the instance.

8. Conclusion

We have proposed a security model for a multilevel secure object-oriented system, SODA. It is posed in terms of an object-oriented computation model. Each object is assumed to be a self-contained computing element whose only interaction with other objects is through sending and receiving messages.

The model allows a flexible labelling strategy based on the inheritance lattice. The strategy accounts for the instance-of and sub-type relation. The two labelling strategies allow classification of the objects themselves, or of their instance variables. This allows a sensitivity to be associated with the information contained in the object or the context in which the information is seen. Each object inherits a labelling constraint from its class. The constraint specifies a range of allowable sensitivity levels for the data being labelled. These types of constraints allow one to control the rigidness of the classification rules on a class by class basis.

Access control in the model accounts for the computational model of the application program. This simplifies the task of multilevel updates on objects with rigid classification rules. Polyinstantiation is only used when it is indicated by the data being stored and not because of restrictiveness of the security model.

One distinct advantage of our approach is that the object-oriented computation model provides a uniform treatment for all objects in the system. This simplifies the statement of a security model and provides a high level of assurance.

We have developed an exploratory prototype of the model to examine its feasibility. The prototype has been modified to adhere to the model described in this paper and we intend to use it to explore implementation issues, the ability of the model to support application development, and performance issues.

References

[1] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou and H. J. Kim, Data model issues for object-oriented applications, *ACM Trans. Office Inform. Syst.*, (1) (January 1987) 3–26.

[2] D. E. Bell and L. J. LaPadula, Secure computer systems: unified exposition and multics interpretations, *Tech. Rep. MTR-2997,* March 1976 (Mitre Corporation, Burlington Road, Belford, MA 01730, U.S.A).

[3] T. A. Casey, Jr., S. T. Vinter, D. G. Weber, R. Varadarajan and D. Rosenthal, A secure distributed operating system, *Proc. 1988 IEEE Symp. on Security and Privacy, Oakland, CA, April 1988,* pp. 27–38.

[4] D. E. Denning, S. K. Akl, M. Heckman, T. F. Lunt, M. Morgenstern, P. G. Neumann and R. R. Schell, Views for multilevel database security, *IEEE Trans. Software Eng.,* SE-13 (2) (February 1987) 129–140.

[5] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman and W. R. Shockley, A multilevel relational data model, *Proc. 1987 Symp. on Security and Privacy, April 1987,* pp. 220–234.

[6] B. B. Dillaway and J. T. Haigh, A practical design for multilevel security in secure database management systems, *Aerospace Security Conf., December 1986.*

[7] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derret, C. G. Hoch, W. Kent, P. Lyngback, B. Mahbod, M. A. Neimat, T. A. Ryan and M. C. Shan, IRIS: an object-oriented database management system, *ACM Trans. Office Inform. Syst.,* 5 (1) (January 1987) 48–69.

[8] C. Floyd, A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Züllighoven (eds.), *Approaches to Prototyping,* Springer, Berlin, 1984.

[9] C. Garvey and A. Wu, ASD_views, *Proc. 1988 IEEE Symp. on Security and Privacy, Oakland, CA, April 1988,* pp. 85–95.
[10] M. Gasser, *Building a Secure Computer,* Van Nostrand Reinhold, New York, 1988.
[11] A. Goldberg and D. Robson, *SMALLTALK–80: The Language and Its Implementation,* Addison-Wesley, Reading, MA, 1983.
[12] T. F. Keefe, W. T. Tsai and M. B. Thuraisingham, A multilevel security model for object-oriented systems, *Proc. 11th Nat. Computer Security Conf., Baltimore, MD, October 1988,* pp. 1–9.
[13] T. F. Keefe, M. B. Thuraisingham and W. T. Tsai, Strategies for secure query processing, *IEEE Computer,* 22 (3) (1989) 63–70.
[14] R. B. Knode and R. A. Hunt, Making databases secure with TRUDATA technology, *Proc. 4th Aerospace Computer Security Applications Conf., Orlando, FL, December 1988,* pp. 82–90.
[15] T. F. Lunt, R. R. Schell, W. R. Shockley, M. Heckman and D. Warren, A near-term design for the SeaView multilevel database system, *Proc. 1988 IEEE Symp. on Security and Privacy, Oakland, CA, April 1988,* pp. 234–244.
[16] T. F. Lunt, Secure distributed data views: identification of deficiencies and directions for future research, *A007: Final Rep.,* Vol. 4, SRI International, January 1989, pp. 65–74.
[17] D. Maier and J. Stein, Development and implementation of an object-oriented DBMS. B. Shriver and P. Wegner (eds.). In *Research Directions in Object-Oriented Programming,* Massachusetts Institute of Technology Press, Cambridge, MA, 1987, pp. 355–392.
[18] M. Mizuno and A. E. Oldehoeft, Information flow control in a distributed object-oriented system with statically bound object variables, *Proc. 10th NBS/NCSC National Computer Security Conf., Baltimore, MD, 1987,* pp. 56–67.
[19] M. Stefik and D. G. Bobrow, Object-oriented programming: themes and variations, *AI Mag.,* 6 (4) (Winter 1986) 40–62.
[20] R. S. Tosten, Data security in an object-oriented environment such as Smalltalk–80, *Proc. 1988 Int. Conf. on Computer Languages, Miami, FL, October 1988,* pp. 234–241.

Appendix A—SMALLTALK Syntax

This appendix provides a brief introduction to the SMALLTALK syntax. A method specification consists of a message pattern and a sequence of expressions separated by periods. The message pattern determines the message selector the method will be used for and assigns names to the formal parameters of the method. An example of a message pattern is shown below

SPEND: amount ON: reason

The message selector for this method is SPEND:ON:. The two formal parameters in this method are "amount" and "reason." The expressions which make up the body of the method consist of message expressions with an optional assignment. Message statements are described briefly below

UNARY MESSAGES

A unary message expression consists of the name of the receiver object followed by the selector of the method to be executed. The statement below sends the message consisting of a selector named SALARY and no parameters to the object "Emp01:"

Emp01 SALARY

KEYWORD MESSAGES

A message can be constructed from parts of the selector or keywords alternated with arguments. The following message sends the object "HouseHoldFinances" the selector SPEND:ON: along with objects representing the real number 30.45 and the string "food."

HouseHoldFinances SPEND: 30.45 ON: "food"

A message expression returns an object as a result which represents the value of the expression. This object can be assigned to an instance variable. This is done by preceding the message expression with the name of the variable and the assignment symbol ← as in the example below

TotalFinances ← TotalFinances + (HouseHoldFinances TOTALSPENTFOR: "food")

BLOCKS

A block is similar to a function in a traditional programming language. It takes a list of arguments and produces a result. A block is similar in form to a method. It is enclosed in square brackets and begins with a list of parameters. Separated from the para-

meters by a "|" is a list of expressions which form the body of the block. The block shown below is a function of one argument "objectToClassify" and returns a boolean result

[:objectToClassify|(objectToClassify salary) > 100000]

The block sends its argument the message "salary" and to the resulting object it sends the message with selector > and argument 100000. A block is an object and can be used as an argument to a method.

Dr. Bhavani Thuraisingham is a lead engineer at the MITRE Corporation researching in database security and integrity. Previously she was at Honeywell Inc., and before that at Control Data Corporation where she worked on database security, knowledge-based systems and networks. She was also an adjunct professor and member of the graduate faculty in the Department of Computer Science at the University of Minnesota.

Dr. Thuraisingham received her BS degree in mathematics and physics from the University of Sri-Lanka, MS degree in computer science from the University of Minnesota, M.Sc. degree in mathematical logic from the University of Bristol, U.K., and Ph.D. degree in recursive functions and computability theory from the University of Wales, Swansea, U.K. She has published more than 30 technical papers in the areas of database security, knowledge-based systems and computability theory. She is a member of the IEEE Computer Society, ACM and Sigma-Xi.

W. T. Tsai received his Ph.D. and M.S. degrees in computer science from the University of California, Berkeley, in 1982 and 1986 respectively, and an S.B. degree in computer science and engineering from MIT in 1979. He is currently an assistant professor in the Department of Computer Science at the University of Minnesota. His areas of interest are software engineering, artificial intelligence, computer security, and computer systems. He is a member of IEEE and AAAI.

T. F. Keefe is a graduate student in the Department of Computer Science at the University of Minnesota. His research interests include database security and software engineering. He received a B.S. degree in electrical engineering and an M.S. degree in computer science from the University of Minnesota in 1980 and 1985 respectively. He is a member of IEEE.