8

Multilevel Data Model for the Trusted ONTOS Prototype

M. Schaefer[a], P. Martel[b], T. Kanawati[b], and V. Lyons[b]

[a]Arca Systems, 10320 Little Patuxent Pkwy, Suite 1005, Columbia, MD 21044, USA, 410-715-0500, 410-715-0713 Fax, marv@md.arca.com

[b]ONTOS, Inc., 900 Chelmsford, Lowell, MA 01851. USA, 508 323-8000, 508-323-8101 Fax, tanyk@ontos.com, wade@ontos.com

Abstract

This paper presents the evolving security model and philosophy of TOP, the Trusted ONTOS Prototype. TOP is being designed as a multilevel, high integrity, client/server object database management system as an extension of the existing commercial ONTOS product. TOP will provide features and assurances comparable to those defined for the B1 level of the US TCSEC.

Keywords: Object Database Management Security, Multilevel Database Security, Multilevel Integrity, Cover Stories.

1. INTRODUCTION

This paper introduces the Trusted ONTOS Prototype (TOP), which offers features comparable to those required for Class B1 of the DOD's Trusted Computer System Evaluation Criteria [3] and the associated Trusted Database Interpretation (TDI) [9] of the TCSEC.[1] We present an abstract overview of TOP's access control policy and introduce the strategy for mediating access between cleared users and objects in the multilevel Object Database.

In the late summer of 1993, a short research study was initiated by the National Security Agency, Rome Laboratory and ONTOS, Inc. to initiate development of an informal access control model for a trusted object-oriented database management system (ODBMS).

Although there have been numerous paper studies,[4,5,6,7,8,10,12,13] there are presently no worked examples of a trusted ODBMS, extant or under development. It is equally important to note that although the more traditional concepts and architecture of relational DBMS (RDBMS) tend to dominate the TDI, there are no *interpretations* of how specific TCSEC requirements are to

[1]The research described in this paper is funded by Rome Laboratory under Contract No. F30602-93-C-0123. We would like to express gratitude to our sponsor Joseph V. Giordano for his continuing and enthusiastic support of our research and also to thank Sandra A. Wade for leadership and vision. We also with to thank Win Cuthbert, Bill Wilson, Matt Morgenstern, Bill Herndon, Arnie Rosenthal and Don Marks for their valued critiques and suggestions throughout this project's history.

D. L. Spooner et al. (eds.), *Database Security IX*

be applied to an ODBMS. The development of TOP is intended to provide a proof-of-concept prototype that will provide a sound basis for future research and development needed to better understand a) the security related issues in the design and implementation and b) the evaluation, and especially the *assurance* requirements for a high-integrity, multilevel secure ODBMS that offers B1 features.

TOP takes a fresh look at the trusted DBMS problem. Previous, relational model-based approaches, have largely been based on a set of security architectures that lead to polyinstantiation as a means of preserving confidentiality. However, use of this strategy is often at the cost of database consistency and integrity. Given that object-oriented architectures invite the introduction of new security architectures, the opportunity is present to re-examine alternatives that could result in a more favorable tradeoff between these objectives.

2. TOP APPROACH

Our approach is based on a survey of relevant prior research in DBMS security, with a concentration in object-oriented studies. Mostly, these have consisted of formal and informal models and descriptions of hypothetical implementation strategies. Some of the literature identified specific constraints on the model and resultant functionality that follow from sometimes identified aspects of the access control model or the envisioned evaluation class. For example:

- B2 and higher evaluation classes concentrate on issues of TCB minimality, least privilege, and covert channels.[3, 13] These concerns tend to force the design in the direction taken by SeaView, LOCK DataViews, etc. This is because inferential attacks may disclose sensitive information if known multilevel integrity constraints are probed by a knowledgeable adversary.
- Least privilege considerations can interfere with the ability of a trusted DBMS to detect cases in which inadvertent polyinstantiation or breaches in referential integrity have transpired. This is particularly the case when a user logged in at a level lower than database high performs updates on the database. This is because the trusted DBMS runs as a subject at the user's logged in level and cannot see any of the database or metadata not dominated by that clearance level. Since the DBMS cannot, under these circumstances, obtain a complete consistent view of the database, it is generally incapable of managing all aspects of the data model itself.
- The above assurance considerations also have their effect on concurrency and transaction management. Contemporary user requirements call for DBMSs that support multiple concurrent users and preserve transactional integrity. This problem is complex and has received intensive research in the untrusted community, and it only becomes more complex and less certain when covert channel-free confidentiality requirements are imposed.

Much of the above complexity has resulted from an evolutionary approach to DBMS security based on prior results in O/S security and in initial attempts to retrofit relational DBMSs onto trusted operating system architectures. The resultant trusted DBMS policies have therefore largely been constrained by intrinsic limitations of O/S policy that, by fiat, could not be modified.

The TOP analysis and assurance methodology is (a) to hypothesize a complete multilevel data model including labeled database entities, (b) to establish through informal analysis that the model is consistent, (c) to superimpose access rules onto this model, (d) to analyze the adequacy of the model against multi-user database goals and objectives, (e) to establish the existence of an acceptable implementation strategy, and (f) to study the expected security properties of the resultant constrained abstract design. Clearly, considerable iteration is required. This approach should yield a usable combination of an access control and integrity model, a hypothetical security architecture, and an interpreted set of criteria against which to measure any faithful implementation.

3. SECURITY ARCHITECTURE AND POLICY MEDIATION PHILOSOPHY

The TOP security architecture is predicated on a client-server architecture. The TOP TCB component will be implemented such that it is interposed between the multilevel object database server and all clients. All of the server will be considered to be part of the TOP TCB. The TOP TCB will be implemented as a Trusted Subject on a platform that satisfies at least the B1 requirements of the

TCSEC. Each client will be an untrusted subject supporting precisely one user on a platform that satisfies at least the B1 requirements of the TCSEC. In addition it is required that the client and server platforms provide a Trusted Path mechanism that satisfies the feature and assurance requirements of TCSEC class B3 in order that the user and the TCB may independently invoke, as needed, a private means of communication that untrusted code may neither intercept nor impersonate. The Trusted Path is an integral part of support for TOP's multilevel integrity policy.

Clients will be supported on platforms that are logically distinct from the platform on which the server is implemented so that all references by subjects to objects necessarily pass through the TCB interface. The untrusted portions of TOP (a majority of the COTS ONTOS product[14]) will run on the client to support all forms of untrusted processing of TOP objects. The TOP TCB will implement functionality described abstractly in the model. The TOP Storage Manager will retrieve and store all persistent objects but will not participate in mediating access control policy.

This security architecture ensures that the needed Reference Monitor Concept properties are implemented: mediation will be complete since all access is forced to pass through the TCB. The TOP TCB is assumed to be provided with individual user identity and clearance information by the platform and LAN TCB components. The TOP TCB domain is designed not to support execution of any untrusted or user software and, along with physical isolation, is protected from tampering or other forms of unauthorized modification.

This logical architecture requires security support from the underlying platform and network. A single-user, untrusted client platform and a trusted LAN satisfying B1 or higher TCSEC/TNI requirements integrated with the B3 Trusted Path mechanism would satisfy TOP's assurance and enforcement requirements. Alternatively, a B1 or higher multilevel platform (providing the B3 Trusted Path mechanism) would also be satisfactory for implementing TOP.

Access is mediated for all objects based upon the clearance and login level of the user on behalf of whom a client domain is executing.[1] Consistent with the TCSEC and TDI requirements, the TOP TCB will permit a user to view a TOP object only if the sensitivity level of the client dominates the sensitivity level of the object; the TOP TCB will permit a user to update a TOP object or introduce a new TOP object into the persistent store only if the sensitivity level of the client equals the sensitivity level of the object. The Trusted Path mechanism will support directive actions by an adequately-cleared user to modify information at sensitivity levels dominated by the login level.

The TOP model allows naturally for low level object instances to contain default values for some of their visible properties while higher level object instances contain non-default values for these properties. This permits the implementation of multilevel views. TOP also permits inconsistencies including: cover stories and single-level updates (corrections) to object instances at different levels. Such Inconsistencies are a special subclass of integrity violation, and are called Polyinstantiation in the literature, but we choose to differentiate between deliberate and accidental cases. Deliberate cases include cover stories and corrections performed by subjects cleared to view all of the relevant information. Accidental cases (sometimes called *automatic* polyinstantiation) are those where, as a consequence of updates performed by untrusted subjects acting at different security levels, the TCB appears to create object instances that have the same object identity but that otherwise differ in value. The accidental case can result either from the user acting on incomplete information or from vestiges of the *-property. To distinguish all intentional cases from the accidental case, the latter case is referred to as polyinstantiation in this model.

Inconsistencies are often required to prevent a lower level subject from having knowledge of an existing instance created by a higher level subject when the lower level subject attempts to create the same instance in the database or to allow a higher level subject to create cover stories. The TOP multilevel data model is designed to automatically eliminate the possibility for most instances of polyinstantiation inconsistency.[2] Special provisions are integrated for the creation and maintenance of cover stories. TOP eliminates many causes of accidental polyinstantiation, while increasing the usability of cover stories[2] by providing: an explicit API to decouple higher level state from

[2]Accidental polyinstantiation occurs when a high level subject modifies data that was read from a lower level instantiation. The two possibilities are: (a) the low level data is a cover story, or (b) the modification should be made

lower levels (semantic vectors); entity integrity support; and, access to a trusted path mechanism to permit interaction between the appropriately-cleared user and the TCB.

3.1 TOP Discretionary Access Control Policy

These major obstacles need to be overcome prior to our being able to address issues of discretionary access control (DAC) policy: (a) identification of the abstractions to be protected by DAC; (b) inherited (c) identification of enforcement mechanisms appropriate to the present TDI guidance for DAC. (a) named objects: *databases, objects, l-instantiations, properties, types, procedures, referents, semantic vectors, registries* and *names* in a *registry.* These terms are defined in section 4.5.1 below. Because (a) pertains to selection of an integrated decision and enforcement strategy, the preëminent issues relate to the size and complexity of the TCB.

Value-based DAC mechanisms are often based on the use of *formularies* that are evaluated at run-time (*e. g.,* a strategist may view information relating to troop deployments only while they are within a specific set of operational theaters). Since formulary-based access mediation is determined dynamically, its use could lead to conflict with uninterpreted TCSEC and TDI requirements, particularly those that require that the TCB be capable of producing the required list of individuals or defined groups authorized for specific modes of access to objects with a database. TOP will operate in accord with the DAC policy of the assumed B1 platforms' TCB. However, because of the rich set of open issues on DAC for an ODBMS, TOP DAC policy will be postponed.

3.2 Secure State in TOP

TOP is in a secure state if, for all untrusted subjects s and all storage objects o in the persistent store: the security level of the domain of s (denoted $sl(\text{domain}(s))$ equals the logged-in security level of the user, u, on behalf of whom s executes; the clearance of u is at least that of s (written **dom**(clearance(u), $sl(s)$)); no object o in the domain of s is more restrictively classified than s (*i. e.,* only if **dom**($sl(s)$,$sl(o)$) and, *a fortiori,* than u is cleared to observe); s may place or modify objects o into the persistent store only if $sl(s) = sl(o)$. That is, an untrusted subject operates at or below the user's clearance level. Such a subject can observe only the information classified at or below its security level. Also, an untrusted subject can write only at its own level.

4. SEMIFORMAL TOP MODEL

In the model, we use standard set-theory and predicate calculus notation, employing *lub* (*glb*) to denote *least upper bound* (*greatest lower bound*), $\exists$ for existential quantification, $\forall$ for universal quantification, $\ni$ for 'such that', $\neg$ for negation, $\sim$ for complementation, $\wedge$ ($\vee$) for conjunction (disjunction), $\oplus$ for set concatenation, and $\otimes$ for Cartesian product. We use the symbol $\varnothing$ to denote the *empty value* and intend that it be synonymous with the word 'undefined'. $o(oid)$ denotes the object having *oid* as its OID. $o(oid, l)$ is the l-instantiation, which may or may not exist, of $o(oid)$. $sv(o(oid, l))$ is the semantic vector of $o(oid, l)$, and $v(o(oid, l))$ is the l-view of $o(oid)$.

A *subject* is defined as a process-domain pair with which there is associated a [range of] complete sensitivity level[s]. A subject is an active agent on the system. This definition is not incompatible with the definition in the TCSEC.[3]

A *tobject* is a passive container of data as defined in the TCSEC and is differentiated from '*object*' as used in the ONTOS object database context. The TCSEC concepts of *named object* and *storage object* are respectively denoted by *tnobject* and *tsobject.* Every tnobject is also a tsobject and, therefore, must have a sensitivity label.

---

at the lower level. For (a), TOP provides a programmable mechanism that does not require a trusted path (semantic vector). For (b), the high level subject can fail, or its user be prompted to act through the trusted path; in either case, no sensitive information would be revealed, though both are inconvenient. Note that it is possible to handle failures programmatically in the TOP programming model.

[3] Nonhierarchical compartments are not adressed in the present model, but will be in future.

A *sensitivity level* is a canonical representation of security clearances (assigned to human users and to subjects) and security classifications (assigned as labels to represent the sensitivity of information or containers of information). The sensitivity level of a subject *s* or a tobject *o* respectively is written *sl(s)*, *sl(o)*. For the time being, TOP only supports hierarchical security levels.

A *hierarchical sensitivity level* is an element of a totally ordered set of sensitivity levels. Traditionally, these are drawn from the set {UNCLASSIFIED, CONFIDENTIAL, SECRET, TOP SECRET}. A hierarchical sensitivity level is a component of a *complete* sensitivity level.

A *dominance* relation is defined for any two complete sensitivity levels l_1,, l_2.. We say that l_1 *dominates* l_2, and write **dom**(l_1, l_2) if $l_1 \geq l_2$. We say that l_1 *strictly dominates* l_2, and write **sdom**(l_1, l_2) if both **dom**(l_1, l_2) and $l_1 \neq l_2$. Note that **dom** forms a partial order, *i. e.*: **dom**(l, l), **dom**(l_1, l_2) and **dom**(l_2, l_1) implies $l_1 = l_2$, and **dom**(l_1, l_2) and **dom**(l_2, l_3) implies **dom**(l_1, l_3) For convenience, we define l_1 to be *higher* (*lower*) than l_2 if **sdom**(l_1,l_2) (**sdom**(l_2,l_1)).

Terms specific to TOP multilevel objects are introduced and discussed below.

4.1 Tnobjects and Tsobjects

In accord with the TCSEC, every persistent object is a tnobject (and discretionary access to object instantiations is thereby required to be controlled by the TCB). Every object is also a tsobject (and hence each instantiation must be labeled with a sensitivity label). *l*-instantiations are tnobjects and tsobjects, as are *properties, types, procedures, referents, semantic vectors, databases, registries* and *name* in a *registry. References* are tsobjects, but not tnobjects.

4.2 Types and classes

Types are objects that define the structure and behavior of (other) objects; i.e.: denotable storable representations of classes. Each object is associated with one direct type; each type typically is associated with many objects called its instances. Types define inheritance, properties, procedures, and structure. Inheritance allows types to be defined as specializations of other types. If a type *A* is defined by inheritance from a type *B*, any procedures, properties, or inheritance defined in type *B* are implicitly defined in type *A*.

Classes are abstract programming constructs that describe the structure and behavior of objects. In languages like C++, classes are second class entities that can neither be addressed nor stored. In some other languages like Smalltalk, classes are implemented through objects, and are first class citizens of the programming model.

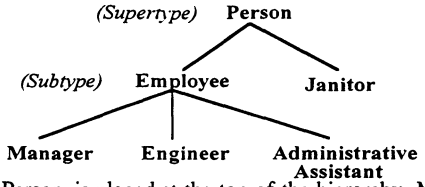
In practice, we use *type* and *class* synonymously.

4.2.1 Types and visibility levels

A *type* consists of properties and procedures. Each of these has a minimum visibility level assigned to it, and is accessible only to subjects that execute at dominating sensitivity levels. Procedures may be invoked to execute at any sensitivity level that dominates their visibility level. Generally, the type is classified to the *glb*(visibility levels(procedures, properties)) associated with the type. Also, TOP allows the assignment of arbitrary visibility levels to types, and does not require that a class graph be isomorphic across visibility levels[4].

In an object database, Types are organized into a hierarchy such as the one below:

[4]Branches in the type tree imply distinctions; that is: information. Therefore, it may be necessary to hide type distinctions from low level subjects. Note that our approach allows for invisible supertypes, and remains otherwise similar to the approach that requires visibility to be non-increasing along inheritance paths (that is: the class graphs remain strongly similar, and are isomorphic at the highest visibility).



The most general class, Person, is placed at the top of the hierarchy. More specialized cases of the Person class are placed below it in the hierarchy. A class hierarchy has an inheritance property associated with it. A supertype, such as Person, passes along its property and procedure definitions to its subtypes, in this case Employee and Janitor. In other words, the Employee and Janitor classes inherit the properties and procedures of their supertype Person. Inheriting behavior (procedures) and representation (properties) enables code sharing (and hence reusability) among software modules. A relational database provides no comparable facility.

Supertype and inheritance relations are derived *at the level of the subject* by traversing the multilevel type hierarchy and selecting the "nearest" supertype visible from the level of the subject.

In the multilevel model, in order for a type or procedure to be inherited by a subtype, the type or procedure must be visible in the supertype. The *visibility level of a type* is dominated by the greatest lower bound of the visibility levels of the type's explicitly defined properties and procedures (if any), and is established by the System Security Officer. In most contemporary multilevel ODBMS models[5,6,7,8,10,12,13], this requirement is interpreted as meaning that visibility levels of subtypes are monotonically nondecreasing on every path from the base class in the class hierarchy tree.[5] Hence, in the above example, the visibility level of Engineer dominates the visibility level of Employee and, *a posteriori*, of Person.

4.2.2 Value Specification

A *Value specification* is an object that describes values; *e. g.*: basic type, amount of indirection, aggregate contents, return values from procedures, etc.

Unlike *types*, *value specifications* can be used to describe integers, and other primitive types (not included under classes in C++), as well as classes. They also describe pointers and references, vectors, and other qualifications of other types without requiring that the programmer define new classes for them. The ONTOS class `OC_ValueSpec` [14] implements this concept.

4.2.3 Objects

Objects are data structures consisting of a set of variables called instance variables or property values and procedures. Objects represent the unit of storage in an object database -- all data in the database is stored as property values of objects. Each object has an identity unique from any other object's. This identity is represented by a unique ID value, the OID. If the object exists (*i. e.,* the OID *oid* corresponds to an object in the persistent store), then the object may be denoted by *o*(*oid*). This object is also called the *complete object*. Every object is a direct instance of a class (or type) *C*, which defines its structure and behavior.

4.2.4 Objects and Object Identifiers

Object identity is that quality of the object that makes it unique. An object identifier is different from Object Identity. It is a physical representation of Object Identity. The OID will *not* be visible to untrusted subjects.

In practice, an OID is a finite string of unique value, not necessarily embedded within the object. Such a string serves as the persistent address of the object. Database clients receive OID strings from the server and use them to denote objects in the database. Typically, OID strings are structured, and contain information about object locations in the persistent store, object age relative to other objects (ordering), etc.

---

[5]The popular nondecreasing monotonicity requirement has also been challenged by Herndon.[6]

TOP introduces OID obfuscation to avoid exposing OID information, by creating transient mappings (transaction duration) which map OIDs to randomized tokens for use by the untrusted clients. Thus, clients can address objects, but are not allowed to observe form or order. Furthermore, the tokens observed by clients lose their meaning outside the transaction in which they are used, and therefore are useless for identity based inference schemes.

4.2.5 Properties

Properties are objects that represent the structural elements (called property values) of the objects of a given type. Every object of a given type has a property value for each property defined by the type. In this sense, types can be thought of as two-dimensional matrices each column in the table corresponding to a property and each row corresponding to an object. Each cell, or intersection of a column and row, corresponds to a property value. Each property is associated with the type that defines it, a property name that uniquely identifies it within the context of the type, and a value specification that describes legal values for the property (in terms of the C++ language's typing system).

Property minimum visibility levels. Each property defined in a type is therein assigned a *minimum visibility level,* a sensitivity level l at which the property becomes visible to all dominating sensitivity levels, l'.

4.2.6 Procedures

Procedures are objects that represent some of the legal operations associated with the objects of a given type. Other operations (such as free functions) may exist and not be modeled in the database. As with properties, procedures are assigned a *minimum level of visibility.* A procedure, p, may be executed as part of the domain of the subject, s, providing **dom** $(sl(s),vl(p))$. That procedure is instantiated to execute at $sl(s)$.

4.2.7 Views, Instantiations, and Semantic Vectors, and References

The *l-view* of an object o is a representation of o's state, as it would be observed by at level l. The view is a computed quantity, not a database object.

(a) The l-instantiation of an object is a data structure that represents the contributions to the object's state at that level. If it exists, the instantiation of an object denoted by the OID *oid* and defined at level l is denoted $o(oid, l)$ and is called the ***l*-instantiation** of the object. (b) The state of an object, as observed at level l, is identical to a lower level state if no l-instantiation exists at that level; otherwise, it is computed from the l-instantiation using the *semantic vector* and, potentially, lower level state. The l-instantiation has a slot for every property defined by the direct class of the object and all its ancestors. Note that $o(oid) = \cup \{o(oid, l)\}$. In a multilevel ODBMS, object identity remains unique, but the OID may be associated with distinct l-instantiations of the object that have been *defined* and entered into the *persistent store* at specific sensitivity levels.

The *semantic vector* at level I, $sv(o(oid), l)$, is a string associated with an l-instantiation, with one symbol for each property value represented by the instantiation. For each property value the associated symbol describes the computation which assigns a value to it: the value is read from below (scooping), or assigned an explicit value. For $t = T(o)$, and each property $t.p$ such that **dom**$(l, vl(t.p))$, $sv(o(oid, l))$ is represented by an element in the semantic vector that is defined and is a member of {scooped, initialized_scooped, static, immutable }.

A *reference* represents a *pointer*, and is essentially another OID representation, suitable for mapping back and forth between persistent and transient address spaces.

The **complete *object*** is the full set of l-instantiations that share a common OID.

The ***l*-complete object** is the set of l'-instantiations of the complete object such that **dom**(l, l').

4.2.8 Property and Property Value Visibility

If a depends on b, then b must be visible whenever a is visible: *i. e.,* **dom**$(sl(a),sl(b))$.

Every property *t.p* of a type *t* is assigned an explicit visibility label, *vl*(*t.p*). *vl*(*t.p*) determines the minimum security level at which instantiations of *t.p* become visible (*i. e.*, may be included in an *l*-instantiation of an object of type *t* or a subtype of *t*). With each attribute explicitly defined in the type *t.p* there is associated a *default value dv*(*t.p*). Either *dv*(*t.p*) is explicitly defined or *dv*(*t.p*) ≡ ∅. If *o* is of type *t*, then *T*(*o*) ≡ *t*.

Hence, for each property *p* in the *l*-instantiation *o*(*oid, l*), we require that **dom**(*l, vl*(*t.p*)), where *p* corresponds to *t.p* and **dom**(*l, sl*(*oid*)). Note that this property value, written *o*(*oid, l*).*p*, is not defined if **not dom**(*l, vl*(*t.p*))

4.2.9 Creation of l*-instantiations*

Creation of the first instantiation of an object is nearly identical to the creation of an object in Untrusted ONTOS. The instantiation is initially created as a temporary object on the client. At the time the subject on a client calls on the server to commit the object, the TCB creates a persistent OID and creates the object's initial *l*-instantiation. The sensitivity level *l* is the level of the untrusted subject on the client. If the commit request is issued from the Trusted Path (*i. e.*, the user on behalf of whom the client is acting issues the request through the Trusted Path), then *l* is the sensitivity level specified by the user, subject to the constraint that the clearance of the user dominates *l*.

A semantic vector *sv*(*o*(*oid, l*)) will be instantiated at sensitivity level *l* at the time of the commit. It will assume the default values for a semantic vector unless the client issues a request to the contrary. *sv*(*o*(*oid, l*)) will be accessible to authorized, cleared, users.

The *l*-instantiation explicitly contains *only* the immutable or static values of each property in accord with *sv*(*o*(*oid, l*)), since all values can be derived from the *l*-complete object. Creation of *l*-instantiations of objects classified 'lower' than any previously defined *l′*-instantiations of the existing persistent object *o*(*oid*) can only be achieved via the Trusted Path, since there is no way for the *l*-instantiation's OID to be referenced from sensitivity level *l*. The semantic vector for the object's instances may need to be reviewed and/or redefined as a consequence of the operation. *E. g.*, if the *l*-instantiation is being created below some *l′*-instantiation that had been using static values for certain properties, it may be appropriate to change the semantic vector for the *l′*-instantiation to use scooping on these properties. A semantic vector *sv*(*o*(*oid, l*)) is associated with each existing *l*-instantiation *o*(*oid, l*).

4.2.10 Semantic Vector Semantics

Initially, *sv*(*o*(*oid,l*)).*p* is either null, 'immutable', 'scooped' or 'static'. It is 'immutable' if the property is the object's Entity Identifier, and 'scooped' if the property is read dynamically from 'below.' If initially *sv*(*o*(*oid, l*)).*p* = static, then initially *o*(*oid, l*).*p* ≡ *dv*(*t.p*). Suppose *sv*(*o*(*oid, l*)).*p* = scooped, and *l′* = *lub*{*l″*∋ *o*(*oid, l″*) exists and **sdom**(*l, l″*) and **dom**(*l″, vl*(*t.p*))}. Then initially, *o*(*oid, l*).*p* ≡ *o*(*oid, l′*).*p*.

If a value is placed in a property from which there is nothing to scoop from 'below', the semantic vector value *sv*(*o*(*oid, l*)).*p* is initialized with the value initialized_scooped.

If *sv*(*o*(*oid, l*)).*p* = initialized_scooped, the value will be given an initial value by the user, but will take on the value scooped and reflect scooped semantics from the time when a new *l′*-instantiation is created, where **sdom**(*l,l′*). From that time on, the value of *o*(*oid, l*) will be from *o*(*oid, l′*).

4.2.11 The l*-view*

When a user logs in, a subject is created on the client at a specific sensitivity level (clearance) that is dominated by the user's security clearance.

Recall that, *o*(*oid, l*) designates the *l*-instantiation of the object *o*(*oid*), which may or may not exist. Similarly, the semantic vector for *o*(*oid*), if it exists, is denoted by *sv*(*o*(*oid*)). It is not planned that the user will be able to request the *l*-instantiation directly, although provision is made

to request the *l*-view visible at the client's sensitivity level. The *l*-view of *o*(*oid*) is designated by *v*(*o*(*oid, l*)). Clearly, the user can derive *o*(*oid, l*) given *v*(*o*(*oid, l*)) and *sv*(*o*(*oid, l*)).

An *l*-view is defined at sensitivity level *l* of the [multilevel] object denoted by *o*(*oid*) only when there is a defined *l'*-instantiation *o*(*oid, l'*), where **dom**(*l, l'*). If *l* dominates a property *p*'s minimum visibility level, then *p*'s property value at *l* is defined respectively to be: (a) the property value of *p* in the *l'*-instantiation, if *o*(*oid, l'*).p exists, where *l'* = **max**{*l''* ∋ **dom**(*l, l''*) **and** *o*(*oid, l''*) exists}; (b) the default value defined for *p* otherwise. For all levels *l* dominating an existing object's *l'*–instantiation, the *l*-view *v*(*o*(*oid, l*)) exists and is defined.

It has not yet been decided whether TOP will support providing a client logged in at *l* with the *l'*-view of an object where **sdom**(*l, l'*). It is unclear whether this feature would be needed, but it could probably be added in the form of a TCB-call that includes the sensitivity level of the requested view. The view will either be identified to an existing *l'*-instantiation or it will be derived from the existing object. So there are three cases: (a) *l'*-instantiation; (b) highest existing *l''*-view, where **sdom**(*l', l''*) and new properties are assigned default values; or (c) ∅. Whatever is retrieved will be treated in the client as if it were a single-level instantiation. *I. e.*, the *l'*-view, if committed by untrusted code operating at *l*, would become the new *l*-instantiation, even if there already existed an *l*-instantiation of the object.

If multilevel updates are desired -- or if polyinstantiation would result from the update -- then the Trusted Path would have to display the server's image of the *l*-complete object with all values at their level along with proposed update, and the *user* would then specify the desired update.

4.2.12 References

A property *p* in an *l*-instantiation *o*(*oid, l*) of an object may reference some other object *o*(*oid'*) providing that there exists a non-null *l*-view of *o*(*oid'*) at the time the reference is made. The sensitivity level of the reference *o*(*oid, l*).*p* is, of course, *l*, since that is the sensitivity level of the client making the reference. Subsequently, should a subject, *s*, view *o*(*oid, l*) from a strictly-dominating level, say *l'*, then if *o*(*oid, l*).*p* is scooped and if *s* follows the reference, *s* will be presented with the *l'*-view of *o*(*oid'*) rather than the *l*-view to which *o*(*oid, l*).*p* was initially bound.

Suppose a *U*-instantiation exists as the object's only instantiation, and it contains a ∅ reference value. Suppose a user creates a *C*-instantiation of that object and `puts` in a *C*-reference to an existing *C*-object. On being notified that polyinstantiation is being created, the user can see the ∅ reference. The user may try using the Trusted Path to place the reference in the *U*-instantiation and scoop up. However, the TCB will not permit this to be done unless a *U*-instantiation of the referent exists.

4.2.13 Cover Stories and Polyinstantiation

Whenever an untrusted subject executes, the ★-Property dictates that all updates default to the subject's sensitivity level. Suppose a subject requests an *l*-view of some object *o*(*oid*). As long as there is some *l*-instantiation *o*(*oid, l*) ≠ ∅ where **dom**(*l, l'*) and *l* = *glb*{ *l''* ∋ ∃*o*(*oid, l''*)} the *l*-view is defined, even if *o*(*oid, l*) = ∅. If the user modifies this *l*-view and issues a `put`, an *l*-instantiation is created if one did not exist before, and should the user subsequently commit the update, the *l*-instantiation is accordingly committed. When an *l*-view is committed, it and its semantic vector are examined in concert with other existing *l*-instantiations of the object. For a given property, it is possible that the value in the *l*-instantiation differs from the value of the property of the nearest scoopable *l*-instantiation. This will result in the semantic vector for this property's *l*-instantiation being updated to static. The result is classed as a *polyinstantiation*. It is also possible

for a subject to explicitly update a semantic vector to static for some property of an l-instantiation. This case is also classed as a polyinstantiation.

More precisely, polyinstantiation occurs whenever there are two different l-instantiations of the same object that [potentially] differ on the value of a specific property that is visible at both levels. Of course, it is sufficient to have $sv(o(oid,l)).p = static$ since property values could not otherwise differ within an object.

Polyinstantiation is always due to deliberate action on the part of the subject acting at a strictly dominating level. In the event where it is intentional that an l-view and an l'–view differ on some property, the 'lower' instantiation is called a *cover story.*

TOP considers the modification of the default (all scooping) setting for the semantic vector on all potentially dominating property values to be an explicit enabling for creation of a [dominated] cover story. No other action is required

4.2.14 Object Deletion

Object *deletion* updates the state of the database. Therefore, like *write*, we cannot allow its observation below the level at which the deleting subject is executing. Furthermore, if there are instantiations above the deletion level, then the deletion is potentially a cover story. Therefore, the effect cannot be automatically cascaded upwards.

The TOP motivation is as follows: to the untrusted user, object deletion, while operating at a particular level, should be indistinguishable from a complete object deletion.

TOP's policy for object deletion is as follows: the level at which the object is deleted is marked by a *tombstone* (a token indicating deletion). If there are no other instantiations for the object, the complete object is deleted (safely). If other instantiations exist below the deletion level, they continue to remain visible at their respective levels. If other instantiations exist above the deletion level, they also continue to be visible at their respective levels, and any values they scooped from the deleted instantiation would be written upwards to maintain the coherence of such views. Uninstantiated levels of the object appear deleted if their views end up being constructed from a tombstone.

During maintenance, and cover story/polyinstantiation reconciliation, it is possible to "revive" an object (remove the tombstone, usually replacing it with a live l-instantiation). Because of the need to maintain the appearance of complete object deletion, we need to insure that any references that used to appear obsolete (pointing to a deleted object) continue to appear obsolete; otherwise, untrusted subjects may infer the existence of higher instantiations. To insure this, we annotate references, and the complete object at each level, with *incarnation numbers* (invisible to the untrusted client). Thus, an obsolete reference continues to appear obsolete, while a fresh reference resolves, though both point to the same object.

4.2.15 Entity Integrity, Polyinstantiation and Object Naming

A *name* can be modeled as an identifier for a stylized property of a directory, and the property contains a reference to an object or $\varnothing$ and may have different values at different levels in the same directory according to TOP's rules of polyinstantiation. Alternatively, a name could be modeled as an object consisting of two properties: a *string* and an OID. Each name is "contained within" (referenced by) one directory. The names in a given directory are constrained such that different names have different strings (property values) and different names have different OID property values. Graphically, this is explained by an array where the OID o_{li} corresponding to the <sensitivity-level, *name*> pair is derived in accord with the following constraints:

$$o_{li} = \begin{cases} \varnothing & \text{name not in use} \\ oid & \text{name refers to some object} \\ \text{scooped} & \text{non null reference value exists at } l' \wedge \mathbf{sdom}(l', l') \end{cases}$$

Note that the object directory in a TOP database is just another aggregate, and is easily modeled as a multilevel dictionary (see below).

4.3 Entity Integrity

In relational database management, the *entity integrity* requirement ensures that at any given time, a relation contains at most one tuple with a specific value of the primary key. Entity integrity has not been relevant to traditional single-level object oriented database management because of its use of *object identity* instead of the content-based addressing represented by the relational model's primary keys. It is possible, and sometimes desirable, for several object instantiations to have identical values.

However, in the multilevel case, an ambiguity problem arises that is not apparent in the single-level case. Suppose a client requests the server to commit a *new* l-instantiation $o(token, l)$, where *token* is a client-specific temporary OID that must be mapped to an OID, and suppose there exists at least one l'-instantiation $o(oid, l')$ where $\mathbf{sdom}(l', l)$. The DBA may prefer that 'obviously' related but differently-classified object instantiations be automatically identified as l-instantiations of the same object and be mapped by the TCB to the *same* OID. If DBA-established criteria for being 'obviously' related are satisfied, then $o(token, l)$ will be mapped into $o(oid, l)$.

ONTOS has always offered the capability for a user to associate a name with an object. While it is not required that every object have a name, ONTOS has also provided the user with a means to determine the name of an object. In order for this capability to be provided in a multilevel context, an analogue to a *primary key* must be specified by the DBA as a set PK of properties in the named object's type such that

- each primary key property is given the same visibility level: $\forall\ p_k', p_k'' \in PK$: $\{vl(p_k') = vl(p_k'')\}$
- the properties comprising the primary key are constrained to be constant in each l-instantiation of the object: $\forall\ l, p_k \in PK\ sv(o(oid, l)).\ p_k \equiv$ **immutable**

The type will define the explicit set of primary key properties, if any, for the object. All the properties identified in the type's primary key must have the same visibility level[6]. The semantic vector will identify instantiations of this *identically-classified set* with the value 'immutable'. That is, for $t.PK$ = the set of properties comprising the primary key of type t: TOP's approach to Entity Integrity (EI) allows cover story creation through normal processes.

Note that the uniqueness constraint imposed for EI considerations cannot be used to its full extent. Most importantly, the constraint cannot be inherited, as that would require TOP to unify within one object instantiations of potentially incompatible types, or would generate failures that would reveal information to uncleared subjects. Therefore, the entity identifiers are annotated with type information, thus weakening the uniqueness constraint.

4.4 Aggregates

Aggregates are an object grouping that provides a convenient means of storing and manipulating either ordered or unordered groups of objects. Aggregates, also known as *collections* or *containers*, are not generally found in relational databases. In most cases, relational databases deal only with single value data types: characters, integers, currency, etc. Object databases, on the other hand, almost always support multivalued data types such as the following aggregates: Lists, sets, arrays, and dictionaries. Using aggregates, it is straightforward to support one-to-one, one-to-many, and many-to-many relationships between objects. Each is discussed briefly below.

4.5 Aggregates in Single-Level Object Databases

This section describes the use of aggregates in traditional, untrusted, object database management. The following section discusses aggregates in the multilevel context of TOP.

A *list* is an ordered unkeyed Aggregate that represents linked lists, sequences, queues, or stacks. It stores members serially; each member (element) has a position in the List. Insertion into

---

[6]Note that such uniqueness constraints are not inheritable, as that would introduce a severe conflict between type coherence and security.

the List at a particular position increments the position of all members following that position. Removal of a member does the opposite.

A *set* is an unbounded, unordered Aggregate. Set members can be inserted, removed, and tested for membership. Unlike the other Aggregates, Sets do not support multiple entries for the same element; all *elements of a set are unique.*

An *array* is an Association Aggregate whose keys must be the continuous range of integers between the specified lower and upper bounds, either of which may be positive, negative, or zero. All of the elements of an Array are allocated and initialized to NULL. The cardinality of an array is the number of distinct values in the range from the *lower bound to the upper bound.*

Like an array, a *dictionary* is also an association aggregate. Keys or tags for a dictionary, instead of being a continuous range of integers, can be user-defined. A dictionary instance's tag can associate one object to one or more other objects supporting associative lookup. Dictionaries are unrestricted in size and may be ordered or unordered.

4.6 Persistent Storage of Aggregates

As a motivation to this discussion, recall that in addition to controlling the visibility of information, TOP's access control policy addresses integrity and usability. Minimally, this means that TOP's access controls are designed towards the goal of not making database use impossible or overly inefficient, nor leading cleared users to erroneous conclusions nor to loss of data integrity.

The notion of polyinstantiation at the representation level (low level data structures) of aggregates must be rejected because of the strong coupling between the components of a container. For example, we cannot scoop the cardinality count, and polyinstantiate hash buckets, and expect that all views would appear as sensible data structures. Also, we rejected a single-level aggregate with polyinstantiation and access control at member granularity because doing so would violate the ★-property (*e. g.*, cardinality is common to all levels).

We consider a simple l-instantiation model consisting of a "full-bodied" aggregate, independent of other instantiations. We extend the concept of scooping to encapsulate membership information, and consider some suitable abstractions. Basically, we introduce a 3-valued notion:

1. At level l' where **dom**(l', l), the aggregate is *always* identical to the one at level l (scooping).
2. At level l', where **dom**(l', l), the aggregate membership information is a modification of the level l information.
3. At level l', where **dom**(l', l), the aggregate is independent of lower levels (traditional static).

4.6.1 l-*view and* l-*instantiation*

In the following text, an l-view is a full-bodied aggregate object, which is constructed using an l-instantiation. Recall that an l-instantiation is an object which, when coupled with lower level instantiations (or views) and its semantic vector provides sufficient information for constructing an l-view. Clearly, the l-view of a T-typed object is an instance of class T, whereas the l-instantiation of such an object is not constrained in this manner.

4.6.2 Visibility of Aggregate Membership Information

In addition to the usual visibility rules, we need to consider the visibility of membership information. That is, although an object X and an aggregate A may both be visible at some level l, the fact that X is (or is not) contained in A may be classified at a higher level.

For example, suppose the record on James Bond may be visible at U, while at level U we can see a set of MI6 employees. However, the fact that Bond works for MI6 (*i. e.*, is a member of the employee set) should be visible at or above level S only. Conversely, if we also keep track of SPECTRE membership in another set, and have Bond infiltrating it, we may desire that his record appear in that set at level U, but be absent at level S or above. The fact that Bond is *infiltrating* SPECTRE may be classified T. Thus, the Bond record appears in both sets (annotated to avoid the appearance of treason, of course).

4.7 Simple Model: Aggregates as Atomic Objects

In this model, we consider aggregates as atomic objects. Under this model, an *l*-instantiation is a full-bodied aggregate; that is, if the semantic vector at *l* indicates that the aggregate is static, then the *l*-view for that *l*-instantiation requires no information from lower levels of visibility to provide membership information. A scooped *l*-view is computed from the "highest" dominated *m*-instantiation when *m* exists. As with other atomic types (*e. g.*: integer), the value of the object is fully stored at one cell, not collected from different cells at possibly different levels of visibility. Instantiation at a level implies a complete separation from the lower level.

On the positive side, this implementation provides

- View consistency: if we assume that updates do not violate the invariants of the application(s) using the aggregate, then it is clear that any *l*-view, scooped or static, is always constructed from a complete and correct aggregate object.
- Simplicity and efficiency: the implementation of this technique is fairly simple; the view construction is simple and efficient, since there are no transformations from an instantiation representation to an aggregate object.

On the negative side, this implementation

- Inhibits upward flow of membership information after cutting interlevel scooping relations.
- Affects large amounts of information due to small changes: that is, a scooping link may be severed because one object was inserted or removed from a very large aggregate. This has the effect of isolating the upper level from the membership changes occurring at a lower level for all the aggregate's members (at the moment of separation). This symptom can be alleviated by allowing the (simple) inspection at level *l* of all the lower level views via TCB calls that allow any dominated level to be specified, thus permitting continuous (programmer assisted) upward data flow, while still presenting a coherent image of the aggregate.

4.8 A Refined Model: Membership as an Attribute

A solution has been found for unordered aggregates and is described in this section.

We represent the membership of an object as a datum (object) visible at the lowest level where both object and aggregate are visible. That is, given: *a*: Aggregate; visible at L_1 *x*: Object; visible at L_2 we assume the existence of ***in***(*x, a*): Boolean; visible at **lub**{L_1, L_2}

For simplicity, we associate these virtual attributes with the aggregate; also, for any pair (*x, a*) for which we have no explicit answer, we assume that ***in***(*x, a*) is false. Explicit annotations appear when membership data is modified, and the annotation data is made visible at the level of the update. The lowest instantiation of the aggregate can be assumed to be a full-bodied aggregate object, or an empty one with a number of explicit annotations. Thus, an *l*-instantiation can be modeled as follows <*initial state, modifier*$_1$, ...>; where

- *Initial state* is either scooped (*i. e.*, a copy of the lower level view), or static (a reference to a full-bodied aggregate).
- *Modifier*$_1$, ... represent the additional membership information at that level; the modifiers are always static.

Using this model, *l*-view construction uses the following algorithm:

- Scoop all the lower level values; *i. e.*, copy the lower level *l*-view into a_l.
  For each t = ***in***(x, a) at level l; if t is false and x is in a_l, remove it; otherwise, if t is true and x is not in a_l, insert it.

On the positive side, this model provides

- Upward data flow of membership information: membership data at lower levels is seen at higher levels.

  Exclusions and inclusions are representable.
- Multiple options for access control: it is possible to decouple *l*-views by severing the scooping link on *initial state*; also, it is possible to maintain sensitive membership information without losing access to the latest lower level information (by scooping *initial state*).

On the negative side, view construction suffers additional computation overhead.

4.8.1 Implementation model for Sets

The *l*-instantiation for a set is represented as a 3-tuple <*initial, insert, remove*> where

initial	is the initial state of the set; can be scooped. The scooped value is computed by constructing the lower level view, not by simply giving the *initial* value of the lower level.
insert	a set of the objects that *must be in* the set at that level; always static.
remove	a set of the objects that *must be excluded from* the set at that level; always static.

The sets *insert* and *remove* are disjoint (otherwise, insertion/removal sequence becomes relevant). Any or all of the sets can be empty. None of the three sets contain members that cannot be visible at level *l*.

Note that access at level *l* requires no information at higher levels and affects no information at lower levels. Thus, these operations can be performed on the untrusted client.

4.8.2 Implementation models for Dictionaries

We model a dictionary as a set of pairs <*key, target*> where

key	is a reference to the key object.
target	is a reference to the object associated with key.

As with sets, a dictionary viewed at level *l* cannot have *key* references to objects which cannot be visible at *l*, since this would make it impossible to use the key to place or locate the *target* object. As with sets, we maintain 3-tuples at each level after polyinstantiation. In this case, the sets *insert* and *remove* do not share any pairs with identical *keys*.

Alternatively, we can polyinstantiate the parts of <*key, target*> pairs, but that is problematic, because this may introduce *key* or *target* references to objects which are invisible at some levels.

Further, we can model *key* and *target* to provide scooping defaults. This can be used to provide default non-null targets at particular keys prior to provision of such information at a lower level.

4.8.3 Implementation model for Arrays and Lists

Unlike dictionaries and sets, arrays and lists (ONTOS `OC_Array`, `OC_List`) impose positional constraints on their members, and make the representation of generalized modifiers difficult. Lists and Arrays can be modified at randomly accessed locations, and have their boundaries shifted. It is impossible to determine the intended meaning of an operation by studying only the modifications.

If we keep track of these updates as absolute array indices and then update the lower level view of the array to a different size, the changes would most probably be applied with the wrong meaning. If we keep track of these updates as updates relative to the head and tail of the array, we cannot represent changes at absolute indices. Note that it is not possible to determine analysis the programmer's intended meaning by static or dynamic, since a programmer may optimize operations thus obfuscating the intended meaning.

In addition to supporting many array operations (random addressing), lists also support expansion and shrinkage at the head, tail, and middle thus changing the indices of (potentially all) list members. This seems to indicate that the representation of modifications in a generally usable fashion is virtually impossible for these two classes. However, if we consider the following variants on `OC_List` and `OC_Array`, we can derive a practical representation of change:

1. The list class allows expansion only at the head or the tail, and does not support random addressing (a double ended queue); *i. e.*, the allowable operations are
   - insert at head
   - insert at tail.

- remove first member.
- remove last member.

2. The array class has fixed boundaries, and cannot expand or shrink; in this case, the data model becomes that of a simple object with a finite number of properties. We may allow different boundaries at different levels of visibility, as long as the boundaries never change at all levels.

Note: our model subsumes lower level C++ vectors, where the lower bound is always fixed at 0.

4.9 Aggregate Summary

In representing multilevel aggregates, TOP will provide the following options:

1. An atomic representation at every (instantiated) level: the construction of an *l*-view requires no information from a lower level view, and is not sensitive to changes in the lower level state after polyinstantiation. The technique is very general, simple, and applicable without need for special case analysis. However, this technique is far too coarse if emphasis is placed on up-to-date access to all visible membership information.
2. In addition to total separation between levels, TOP also presents techniques for presenting multilevel aggregates that retain upward data flow of membership information (inclusions and exclusions). The techniques are applicable to sets and dictionaries (`OC_Set`, `OC_Dictionary`), but cannot be applied to list and arrays (`OC_List` and `OC_Array`) because we cannot derive a robust representation of modification for these classes.
3. For generalized lists and arrays, only the first option is applicable.

We will support restricted array and list forms that can be managed incrementally.

5. REFERENCES

[1] Bell, D. E., and LaPadula, L. J., *Secure Computer Systems: Unified Exposition and MULTICS Interpretation*, MTR-2997 Rev. 1, MITRE Corp., Bedford, Mass., March 1976.

[2] Burns, R. K., "Integrity and Secrecy: Fundamental Conflicts in the Database Environment", *Written under U.S. Air Force, RADC contract number F19628-89-C-0001.*

[3] Department of Defense, *Trusted Computer System Evaluation Criteria*, DoD 5200.29-STD, December 1985.

[4] Jajodia, S., and B. Kogan, "Integrating an Object-Oriented Data Model With Multilevel Security", *Proc 1990 IEEE Symposium on Security and Privacy,* Oakland, CA, October 1990.

[5] Keefe, T., W. T. Tsai, and B. Thuraisingham, "SODA - A Secure Object-Oriented Database System", *Computers and Security,* Vol. 8, No. 6, October 1989.

[6] Lunt, T., "Multilevel Security for Object-Oriented Database Systems", *Proceedings of the 3rd IFIP WG 11.3 Workshop on Database Security,* Monterey, CA, September 1989.

[7] Millen, J., and T. Lunt, "Security for Object-Oriented Database Systems", *Proc 1992 IEEE Comp Society Sympos Security and Privacy,* Oakland, CA, May 1992.

[8] Morgenstern M., "A Security Model for Multilevel Objects with Bi-directional Relationships", *Proc. 4th IFIP 11.3 Working Conf. Database Security,* Halifax, England, 1990.

[9] National Computer Security Center, *Trusted Database Interpretation*, Fort George G. Meade, MD, NCSC-TG-021, Version 1, April 1991.

[10] Rosenthal, A., W. Herndon, B. Thuraisingham, and R. Graubart, "Multilevel Security for Object-Oriented Database Management Systems", Working Paper No. WP-92B0000375, The MITRE Corporation, Bedford, MA, 1993.

[11] Schaefer, M., Hubbard, B., *et al., Secure DBMS Auditor*, Final Technical Report, RADC-TR-90-130, Rome Laboratory, Griffiss AFB, NY, July, 1990.

[12] Thuraisingham, B., October 1989, "A Multilevel Secure Object-Oriented Data Model", *Proceedings of the 12th National Computer Security Conference,* Baltimore, MD.

[13] Thuraisingham, B., March/April 1990, "Security in Object-Oriented Database Systems", *Journal of Object-oriented Programming,* Vol. 2, No. 6.

[14] ONTOS DB 3.1, *Tools and Utilities Guide, Developer's Guide, Reference Manual* Volume 1: *Class Library*, Volume 2: *Exceptions, Volume 3: Function Library, ONTOS, Burlington, MA. 1994.*