Essay 23

Multilevel Secure Database Management Prototypes

Thomas H. Hinke

The three systems described in this essay each target the most stringent security standards embodied in the A1 requirements as defined by the US Department of Defense "Trusted Computer Systems Evaluation Criteria" [DOD85], which is commonly called the Orange Book. While the initial designs of all these systems — and in some cases the implementations — predate the "Trusted Database Management System Interpretation of the Trusted Computer System Evaluation Criteria" [NCSC91] (commonly called the TDI), they all had the Orange Book as a guide to the fundamental requirements that must be satisfied by a secure system at the A1 level. Also, in some cases, work on these prototypes provided the basis for comments on the evolving TDI.

In addition to their A1 target, these three systems share a similarity in that they were and are intended as research prototypes, not commercial products. This means that they are not held to the requirement of satisfying market conditions, but also that they may not have had the funding to include all of the capabilities that would be required of a commercial offering. Their intent was to push forward the frontier in the area of security, but not necessarily with all the "bells and whistles" of a complete product. This is not to detract from their contributions, which are many, but only to alert the reader to the fact that today's research prototype may lead tomorrow's commercial product by many years.

All of the systems enforce both a mandatory and a discretionary policy. The basis for mandatory enforcement is the *access class*, which includes a hierarchical, linearly ordered component called *levels* (for example, Top Secret > Secret > Confidential > Unclassified), and a nonhierarchical component called *categories*, which is not ordered. The set of access classes is partially ordered and forms a lattice. In this lattice, access class A is said to *dominate* access class B if the hierarchical component of A is greater than or equal to the hierarchical component of B, and the set of categories associated with A is a superset of the set of categories

associated with B. Level A strictly dominates B if the access class of A dominates the access class of B, but is not equal to the access class of B.

While it is recognized that access class includes both a level and a category, this essay will sometimes use the terms access class and security level interchangeably. Also, to illustrate examples, the essay will sometimes describe activity in terms of a high level and a low level, where, depending on context, high will mean an access class that strictly dominates low, or an access class that is not comparable with low.

The access policies for the systems are stated in terms of active *subjects*, which represent users, and *objects* that are to be protected. All of the systems enforce a security policy that states that a subject can read an object only if the access class of the subject dominates the access class of the object. A subject can write into an object only if both have equivalent access classes. A subject is not permitted to write into an object that it strictly dominates, since this would be a write-down in security level and could provide an avenue for Trojan horse code to leak data to a less protected object (that is, one labeled with a lower access class label). This no-write-down policy is called the *-property [BELL75].

In this essay, we will call the access class at which a user is currently interacting with the system the user's current access class or current level. This current level must be dominated by the maximum security clearance held by the user.

The next three sections of this essay will describe each of the three prototype systems, first the SeaView project, then ASD, and finally LDV. Each of these three sections will begin with a discussion of the security model that each system is designed to enforce. The security model includes not only the security policy, but also the security operations that the system is designed to provide. In a sense, the model provides a high-level functional description of the system, with little concern for the details of how this will be accomplished in a secure manner. Following the description of the security model enforced by each system, the architecture of the system will be discussed. This architectural description will show how each of the systems is designed to satisfy the A1 requirements of a security reference monitor, while providing all of the security features called for in the model. In the case of ASD, which was designed for operation as a secure server, this section will show how the ASD architecture fits into the general network environment for which it was designed. The final section of this essay will compare and contrast the various systems, highlighting their fundamental differences.

SeaView

SRI International and Gemini Computers began the SeaView project in 1985 under the sponsorship of the US Air Force Rome Air Development Center. As this is being written, SRI International, Gemini Computers,

and Oracle are currently in the final phase of a contract to implement the SeaView trusted DBMS.

The following SeaView discussion is divided into two major sections. The first section describes the security policy and features provided by the system, and the second describes the SeaView architecture. (Material presented here was taken from the work of T.F. Lunt et al. [LUNT88, LUNT90, LUNT90b].)

SeaView security policy and features. This section will describe the SeaView mandatory and discretionary access policies.

SeaView mandatory access policy. As a system targeting the A1 level, SeaView enforces a mandatory security policy. Its mandatory policy controls access to the granularity of an individual data element in a tuple. This means that within a single tuple, each data element representing a different attribute can have a distinct and possibly different access class. This also means that data elements for the same attributes in different tuples can have distinct and possibly different access classes. The access class of an element is indicated by associating an access-class label with each element of a tuple.

In addition to having an access-class label associated with each element of a tuple, SeaView also associates an access-class label with each individual tuple. This tuple access-class label represents the least upper bound of the access classes of all the information used to derive the data in the tuple. More will be said about this later.

SeaView represents the classification of each tuple and attribute as attributes in their own right. Thus, a relational query language can also make queries based on the classification of attributes.

An element-level granularity of control represents the most specific granularity of control that can be supported in a relational DBMS. It is more specific than attribute-level, tuple-level, or relation-level granularity of control. However, it should be noted that any type of relational granularity of control can be mapped into any other type. If, for example, the application requires element-level granularity of control, but only relation-, tuple-, or attribute-level protection can be provided by the multilevel relational DBMS, then the database can be decomposed by security level into objects with the granularity provided by the DBMS. The desired view of the database can then be reconstituted. As will be described shortly, although SeaView provides the user with an element-level granularity of control, at the storage level it uses a DBMS that provides only a relation-level granularity of control. While relation-level storage was selected for SeaView, a choice of tuple- or attribute-level control could also have been mapped into the element-level granularity that is provided to the user [HINK75].

Mandatory access control within SeaView is governed by the Bell-LaPadula security model [BELL75]. Subjects operating on behalf of a user operate at an access class that reflects the access class at which the user is currently interacting with the system. A subject can read a DBMS object only if the access class of the subject dominates the access class of the object. A subject can write into an object only if the access class of the object is equal to the access class of the subject.

To prevent the output of data that was derived by using data with a more dominant access class, SeaView (and any other system that purports to be secure) requires that the access class of the data output must dominate the access classes of all data used to derive the output data. SeaView handles this by assigning the tuple an access-class label that dominates all of the data contained in the tuple or that was used in deriving the tuple. The individual elements of that tuple will bear the original classifications of the elements stored in the database. In SeaView, element labels are advisory. This means that they are not provided via a path that contains only trusted code, and thus could have been modified by untrusted code in the path.

SeaView discretionary access policy. SeaView enforces a discretionary policy as required for A1 systems. However, under the current design, the discretionary policy-enforcement code would not be A1, in the sense that it is formally specified and verified. It would be considered to be at the C2 evaluation level. This concept of C2 assurance for discretionary and A1 assurance for mandatory has been called balanced assurance [LUNT88]. Although this concept is permitted by the "Trusted Network Interpretation" of the Orange Book for network applications [NCSC87a], it was unsuccessfully proposed by a number of database researchers as an effective method for meeting the A1 requirements for the "Trusted Database Interpretation" of the Orange Book [NCSC91].

The objects to which the SeaView discretionary access control is applied include the entire database or an entire relation. Thus, in contrast to the mandatory access control that is applied to a very fine granularity, the discretionary access control is applied to a much coarser granularity. The relations to which the discretionary controls can be applied include base relations, which are the actual relations in permanent storage; views, which are the data actually seen by a user; and snapshots, which are the relations that result from a relational query. The subjects to which discretionary access controls can be applied include both individual users and groups of users.

The nature of the discretionary access that can be granted is expressed in terms of access modes. A subject can be granted a particular access mode to a discretionary object. The access modes include the following: insert, delete, retrieval, update, reference, null, grant, and give-grant. While insert, delete, retrieval, and update have the usual meanings, the

other access modes have SeaView-specific meanings that will be described in the following paragraphs.

The reference access mode is used to control access to views. A user (for example, user A) can define a view of a relation and provide another user (user B) access to the view, but not to the source relation(s) from which the view is derived. This has the effect of protecting sensitive data in the source relation(s) that user A does not want user B to see. However, as further security protection, SeaView requires that for a user to access a relation through a view, the user must have retrieval access to the view and reference access to the underlying source relations from which the view is derived.

SeaView has the null access mode to provide a means for denying access to a particular object for a user or group. If, for example, user B is to have no access to relation R, then user B is given null access to relation R.

As has been noted, users can be granted or denied access as individual users or as members of defined groups. This raises the possibility that there may be a conflict, in that a user may be granted access under one group and denied access under another group or as an individual. The SeaView discretionary access policy resolves this conflict with the following rules:

- The most specific access takes precedence. Thus, discretionary access permissions or denials specified for a specific user take precedence over any discretionary access permissions or denials specified for any group of which the user is a member.
- In the case of further conflict, denials take precedence over permissions.

To control the propagation of discretionary access modes to other users, SeaView uses the grant and give-grant access modes. The grant access mode allows the recipient to grant any of the accesses, except the grant access mode itself, for the object specified in the grant. The grant access mode itself can be propagated by the recipient only if he has been given the give-grant access mode.

Access modes previously granted can be revoked. The revocation of an access mode applies only to the specific user to which the revocation is applied. In SeaView, the revocation does not affect any subsequent grants that may have been performed by the user to whom access was granted. Thus, if user A gives both retrieval and grant access for relation R to user B, user B can then give retrieval access for relation R to user C. Now, if user A revokes user B's retrieval access to relation R, this affects only user B. User C's retrieval access to relation R is not affected.

SeaView relational integrity constraints. SeaView enforces a number of relational integrity constraints that must hold for data contained in the SeaView database. The first two are variations of the entity integrity and referential integrity constraints that are found in nonmultilevel databases, but that have been adapted to multilevel databases.

The entity integrity constraint requires that no primary key be null [ELMA89]. With SeaView this holds for multilevel relations as well. All of the elements that comprise the key for a relation must be visible at the lowest access class at which any part of the relation can be viewed. This means that the access class of the key elements must be dominated by the access classes of all other elements in the tuple. Thus, each nonkey tuple must possess a key for any access class at which the tuple is visible. This also means that all elements that comprise the key must have the same access class. Otherwise, there will exist some access class at which only a portion of the key will be visible, and this portion will no longer comprise a unique key, since by definition the key is the smallest set of elements that can uniquely identify a tuple.

Referential integrity requires that nonnull, nonprimary key elements of a tuple that references the primary key of another relation (these nonprimary key elements are called foreign keys) must reference a primary key that currently exists as one of the tuples in the referenced relation [ELMA89]. In other words, a nonnull foreign key cannot refer to a currently nonexisting tuple. SeaView requires this to hold for multilevel relations as well. This means that if a foreign key is visible in a tuple at a particular access class, then the key portion of the tuple containing the referenced key must also be visible. The database cannot have foreign key references to primary keys that are not visible at the access class at which the foreign key is visible. Of course, if the foreign key value within the tuple is not visible (that is, null) at a particular level, then there is no requirement that the primary key be visible. SeaView further requires that both the foreign key and the referenced key have the same access class [LUNT90] to eliminate the possibility of referential ambiguity [GAJN88].

To understand the third constraint, the concept of polyinstantiation (the name originally coming out of the SeaView work) must be introduced. While a nonmultilevel relational database does not permit two tuples to have the same primary key, this is not necessarily the case in a multilevel database. This situation arises because the viewability of tuples or elements within tuples is a function of the access class of the user who is viewing the database. Users at lower access classes will not be able to see tuples of elements that strictly dominate the user's access class. From the perspective of this lower access class user, these tuples, or elements, do not exist; they are null. A problem arises if this low access class user attempts to enter a tuple with a key identical to that of a tuple with a strictly dominating or noncomparable access class. A prob-

lem will also arise if the user attempts to enter a value for what appears to be a null data element, when in reality a value already exists, but is invisible since its access class strictly dominates that of the user. A more complete description of polyinstantiation is contained in Essay 21.

There are three solutions to this problem. The first solution is to deny the data entry or update attempt. In effect, this informs the user that data exists whose access class does not permit the user to read it. If the low access class user can detect the existence of this data, this constitutes a covert channel and would be prohibited for systems at the B2 and above evaluation levels.

The second solution is to permit the entry or update. This destroys the dominant access class data, which means that less cleared users can potentially destroy the most classified data, a situation that is generally viewed as unacceptable.

The third solution avoids the covert channel problem by permitting the data entry or update attempt through polyinstantiation. If a user enters a tuple with a key identical to a more dominant tuple, this leads to polyinstantiated tuples. Both tuples will have identical primary keys, but will differ in access class and perhaps the access class of one or more nonkey elements, since the low access class user will have no knowledge of the contents of the dominant access class tuple.

If a user updates what appears to be a null element value with a new one (where an element value already exists at an access class that strictly dominates or is noncomparable with that of the user), this leads to a polyinstantiated element. This means that the particular element has multiple values, each entered at a different access class. In SeaView this is represented as multiple tuples, differing only in the value of the element that is polyinstantiated.

While it is clear that polyinstantiated tuples lead to multiple tuples, in SeaView this is also the result of polyinstantiated elements. SeaView handles polyinstantiated elements by polyinstantiating the tuple, with each of these tuples sharing the same primary key and the same classification of the primary key, but having different values and associated element classifications for the polyinstantiated element. This ensures that each tuple satisfies the requirements of the relational first normal form — that the value of each element is atomic [ELMA89].

SeaView enforces the polyinstantiation integrity rule, which states that tuples with duplicate keys are permitted only in the security-driven cases of tuple- or element-triggered polyinstantiation, as described previously.

The primary problem with polyinstantiation (both tuple and element) is that at higher access classes, users and their application programs will see multiple tuples with the same primary key, when only one is normally expected in a relational database. The user now has the problem of deciding which of the multiple values is to be used. At this stage of

DBMS security with the relational model, the solution of this problem is left to the user or the application writer.

SeaView architecture. The SeaView architecture, shown in Figure 1, takes a strict TCB subset approach, in which layers of TCBs provide increasingly more restrictive controls over the protected data. The lowest TCB layer enforces the mandatory security policy, and the second TCB layer enforces the discretionary security policy. The highest layers are untrusted from both a mandatory and a discretionary perspective, but enforce the SeaView integrity controls and provide the multilevel view software.

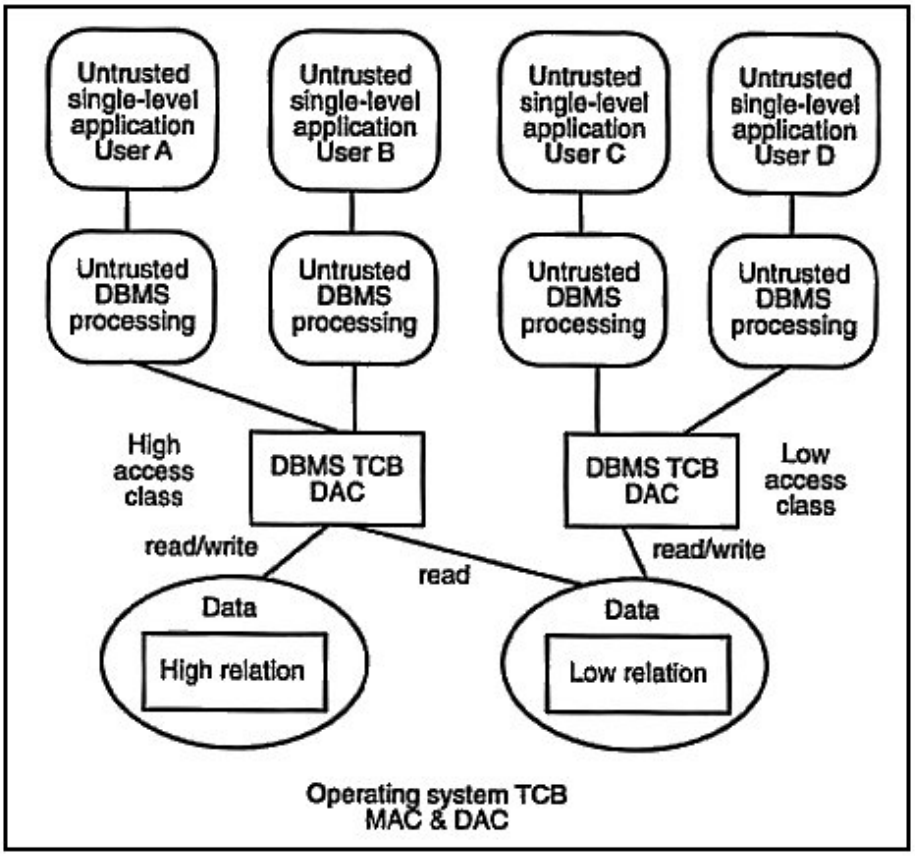

Figure 1. SeaView architecture.

To ensure that the SeaView processing is adequately encapsulated, subjects that support the discretionary controls of layer 2 must be single level, with each supporting only one of the access classes for which there

is current activity. Since the highest layers are untrusted from both a mandatory and a discretionary perspective, there must exist one active subject for each active user.

With this basic outline of the SeaView architecture, the remainder of this section will describe the nature of the objects protected by the TCB layers, the software used by the TCB layers, and the nature of the processing supported by the higher layers.

SeaView object protection. As noted in the previous section, SeaView provides an element-level granularity of control. However, rather than having the lowest, mandatory TCB layer support element-sized objects, SeaView has taken the approach that at the lowest level, the DBMS will support a relation-level granularity of control, with the finer, element-level granularity of control provided by the higher, untrusted view layers of the DBMS.

At the lowest level of abstraction in the DBMS, data is partitioned among base relations by access class. Each base relation is "single level" for all of its nonkey data. The access class of all nonkey data contained within the relation is equal to that of the relation. The base relation also contains primary keys, whose access class may be dominated by the access class of the base relation. Within the base relation, only the primary key data has an explicit access-class label. The access class of all nonkey data is given by the access class associated with the base relation. The access-class labels are required for the primary key to support relational joins. During a join, the key value and the key's access class are used together to reconstitute the multilevel relations from what, except for the keys, are essentially single-level base relations.

The base relations are protected by the lowest, mandatory TCB layer. For this layer, SeaView uses the GEMSOS multilevel secure operating system, which is targeting the A1 evaluation level. Each of SeaView's base relations is stored in a GEMSOS segment of the appropriate access class. Under this approach, GEMSOS — rather than the DBMS code — enforces SeaView's mandatory access policy.

Since all of the higher DBMS layers execute as part of subjects restricted to processing for either a single access class or a single user, GEMSOS will permit these DBMS subjects to read only those operating system objects that they dominate in access class, and write into those operating system objects that are equal to their access class. This means that the DBMS subjects cannot access any base relations that strictly dominate their access class. This concept of using the underlying operating system to provide security for the DBMS was originally suggested in an earlier work [HINK75].

While GEMSOS is used to provide the mandatory controls, it does not provide the discretionary controls. This would require that the DBMS

data be decomposed into objects that are homogeneous with respect to both mandatory and discretionary access permissions.

SeaView software. SeaView uses the Oracle mandatory prototype to provide both the discretionary access controls and the DBMS engine. Multiple instances of Oracle execute, each at a single access class. Thus, Oracle's access to the data is restricted by GEMSOS. Also, since there is an instance of Oracle for each access class, Oracle does not have the capability of compromising mandatory security — each executing instance of Oracle supports only a single access class of processing. Oracle provides its users with a secure DBMS that supports a relation-level granularity of control.

The software that maps the single-level relations provided by Oracle into the multilevel relations with element-level granularity of control that SeaView provides for the user is the multilevel SQL processor (MSQL). This software not only provides the multilevel relational view; it also enforces the various multilevel integrity rules previously described. An instance of MSQL executes for each SeaView subject.

The multilevel relations created by the user must be decomposed into the multiple, single-level base relations that are stored in the GEMSOS segments. Ideally, the decomposition of a user-provided multilevel relation or multilevel tuple into multilevel, single-level relations, and then the subsequent reconstitution of these multiple single-level relations into the multilevel relation originally entered, should be lossless. The data entered should be identical to the data that is viewable again after reconstitution. In addition, it is desirable to be able to enforce the various integrity constraints. As has been pointed out [JAJO90], there is a relationship between the objective of lossless decomposition/reconstitution and the polyinstantiation integrity constraint.

The original SeaView model was designed such that the decomposition and reconstitution of the multilevel relations automatically satisfied the SeaView polyinstantiation integrity constraint. The formalization of the SeaView polyinstantiation integrity rule requires both a functional dependency and a multivalued dependency, which preclude the existence of certain combinations of tuples. The SeaView designers argue that these are not relevant.

When the polyinstantiated tuples that have been created are stored, the polyinstantiated tuples and elements that exist in the relation viewable by the users are removed as the multilevel relation is decomposed into its single-level relation components. These polyinstantiated tuples and elements are then automatically restored when the multilevel relations are reconstituted from their single-level components. In the SeaView design, this is accomplished by using the outer join for performing the reconstitution. The use of the outer join results in the automatic enforcement of the SeaView polyinstantiation rule for the view that is reconstituted.

Had SeaView used another approach for reconstitution, then its polyinstantiation integrity would not necessarily have been automatically enforced. However, a different polyinstantiation integrity constraint would have been enforced. Jajodia and Sandhu [JAJO90] suggest the use of the natural join for reconstitution, with a polyinstantiation rule based only on functional dependency rather than the multivalued dependency of the SeaView polyinstantiation rule. Under their suggestion, any combination of polyinstantiated tuples entered by a user could be decomposed into base relations and reconstituted using the natural join.

An example taken from Jajodia and Sandhu's work [JAJO90] will illustrate this relationship and introduce one of the more challenging open research issues in multilevel DBMSs. The example uses the relation SOD, consisting of the attributes Starship, Objective, and Destination. It will be assumed that Starship is the primary key whose access class must be Unclassified, since it must be dominated by all of the other attributes in the relation. The access classes of Objective and Destination can range from Unclassified to Secret. Assume, after a series of updates, that the following tuples exist, reflecting Objective and Destination:

Starship		Objective		Destination	
Enterprise	U	Exploration	U	Rigel	S
Enterprise	U	Spying	S	Talos	U
Enterprise	U	Exploration	U	Talos	U

As noted by Jajodia and Sandhu [JAJO90], these tuples could represent two secret missions: one to explore Rigel, a secret destination, and the other to spy on Talos, a secret objective. The totally unclassified tuple could represent an unclassified cover story for both the missions.

The original SeaView decomposition/reconstitution approach enforced the SeaView polyinstantiation integrity constraint. This integrity constraint consists of two parts. The first requires that there be a functional dependency from the primary key, the access class of the primary key, and the access class of the *i*th element and the value of the *i*th element. This means that there can be only a single value associated with a particular tuple key and nonkey element class.

The second requires that there be a multivalued dependency from the primary key to the *i*th element access class and value. It is this part of the polyinstantiation integrity constraint that leads to the loss during reconstitution. In the above example, this would require that, when reconstituted, the data included in the three previously presented tuples actually be reconstituted as the four following tuples:

Starship		Objective		Destination	
Enterprise	U	Spying	S	Rigel	S
Enterprise	U	Exploration	U	Rigel	S
Enterprise	U	Spying	S	Talos	U
Enterprise	U	Exploration	U	Talos	U

As can be noted, there exists a tuple instance that was not intended, but the multivalued part of the polyinstantiation integrity constraint is satisfied. What is lost is the fact that the users had entered three tuples, and now there are four tuples, thus losing the intent of the users. To rectify this problem, Jajodia and Sandhu [JAJO90] suggest that the multivalued dependency portion of the polyinstantiation integrity constraint be dropped, and the SeaView use of the outer join to reconstitute the multilevel relations from the single-level base relations be dropped in favor of the natural join.

The SeaView designers argue that the focus should not be on the decomposition/reconstitution approach taken, but rather on the update semantics desired for database operations, including those involving polyinstantiation [LUNT90]. They propose a first cut at a desired set of update semantics and identify the development of a decomposition/reconstitution algorithm as an open research question.

It is beyond the scope of this essay to provide an in-depth discussion of the issues inherent in polyinstantiation integrity constraints, decomposition/reconstitution algorithms, or desired update semantics. However, it is important to alert the reader to the fact that there are deep issues buried in these areas. The interested reader is referred to the cited papers and Essay 21 on polyinstantiation for more information on this area.

Advanced Secure DBMS

The TRW Advanced Secure DBMS (ASD) was developed under an internal TRW research and development project. The goal of ASD was to investigate the technology required to design and implement a trusted DBMS targeting the A1 evaluation class as defined by the Orange Book (DoD 5200.28-STD). The Orange Book, rather than the Trusted Database Interpretation, was the target guidance, since the work on ASD predated the Trusted Database Interpretation. In addition, it should be noted that ASD was simply a research prototype; it was never intended to be a product. ASD was written in Ada to run on the ASOS operating system.

The following ASD discussion is divided into two major sections. The first section describes the security policy and features provided by the system, and the second describes the ASD architecture. (Material presented here was taken from earlier work [HINK88, HINK89].)

ASD security policy and features. This section will describe the ASD mandatory and discretionary access policies.

ASD mandatory access policy. ASD supports a tuple-level granularity of control for its mandatory access. All of the data within a single tuple are considered to have the same access class. Thus a tuple is single level. However, each tuple in a relation can bear a distinct access class; thus the ASD relation is multilevel.

ASD enforces the Bell-LaPadula security policy, which requires that subjects must have a dominant access class to read data. All data that is written will have an access class equal to the access class of the subject that is performing the write [BELL75].

As has been described in detail under the SeaView discussion, tuples within ASD can be polyinstantiated. However, since ASD supports a tuple-level granularity of control, polyinstantiated elements are not possible.

ASD discretionary access policy. While ASD enforces its mandatory access policy for the individual tuples of the relation, it enforces discretionary access controls on the entire relation. All of the tuples within the relation are subjected to the discretionary controls imposed on it.

The discretionary controls are associated with the DBMS relational operations of select, insert, delete, and update. For each relation, discretionary access can be specified as a permission or a prohibition for a subject to perform the specified relational operation on the designated relation. The subjects for which the permissions or prohibitions can be specified include individual users, groups of users, or the public as a whole.

If there is a conflict between discretionary access permissions and prohibitions, the following two rules are used:

- The most specific access constraint takes precedence over the least specific access constraint.
- If the permission and the prohibition have the same specificity, the prohibition takes precedence over the permission.

The primary justification for associating the discretionary access controls with the relation rather than with the tuple is to conserve storage space. The amount of data required to represent the discretionary access permissions or prohibitions is considerably greater than the data re-

quired to represent the mandatory access labels. For example, each of the discretionary access permissions could have a long list of users associated with each type of access. Thus, the decision was made to associate the greater volume of discretionary access control data with the entire relation — where it could apply to all of the tuples within the relation — rather than with each individual tuple.

ASD architecture. This section will first describe the internal ASD architecture, then the network architecture that permits ASD to operate as a trusted server. The ASD architecture is shown in Figure 2.

Internal architecture. The ASD architecture will be described in terms of its process structure, storage structure, and network architecture.

In contrast to SeaView and LDV, ASD takes a trusted process approach to its internal security architecture. What this means is that trusted ASD code, called the ASD TCB, enforces both the discretionary and the mandatory DBMS security policies. This contrasts with both SeaView and LDV, which rely on the operating system TCB to enforce the mandatory policy, with the DBMS TCB enforcing only the discretionary policy.

While the ASD TCB runs as a trusted process under the control of the operating system TCB, this is not to say the ASD TCB is coresident with it. The trusted ASD code does not share the protection domain of the underlying operating system.

Like the other secure DBMSs, ASD consists of both trusted and untrusted code. The untrusted ASD code performs those DBMS operations that are not security relevant. Multiple instantiations of this untrusted code run, one for each user. This code runs at the access class of the user on whose behalf it is processing.

The operating system TCB ensures that the ASD TCB is a security reference monitor. It protects the DBMS TCB from tampering by other processes operating on the system. It also ensures that no process can access the DBMS data except through the ASD TCB. The details of this will be described shortly.

Conceptually, all of the ASD tuples are placed in a single operating system file. The file must be classified at the least upper bound of the tuples it contains. Each of the tuples stored in this file bears a classification assigned by the ASD TCB, based on the security level of the subject that initiated storage of the tuple.

In addition, since the ASD data file contains tuples labeled with different classifications, this file must be protected from modification by untrusted processes that may have a classification equal to the ASD data file. There are a number of ways that this could be accomplished. One of the least desirable would be to use the discretionary access features of the underlying operating system to ensure that only the DBMS TCB could have write access to the ASD data file. This has the unfortunate

consequence of allowing a mandatory access mechanism to rely on a discretionary access mechanism for its tamper-resistant property.

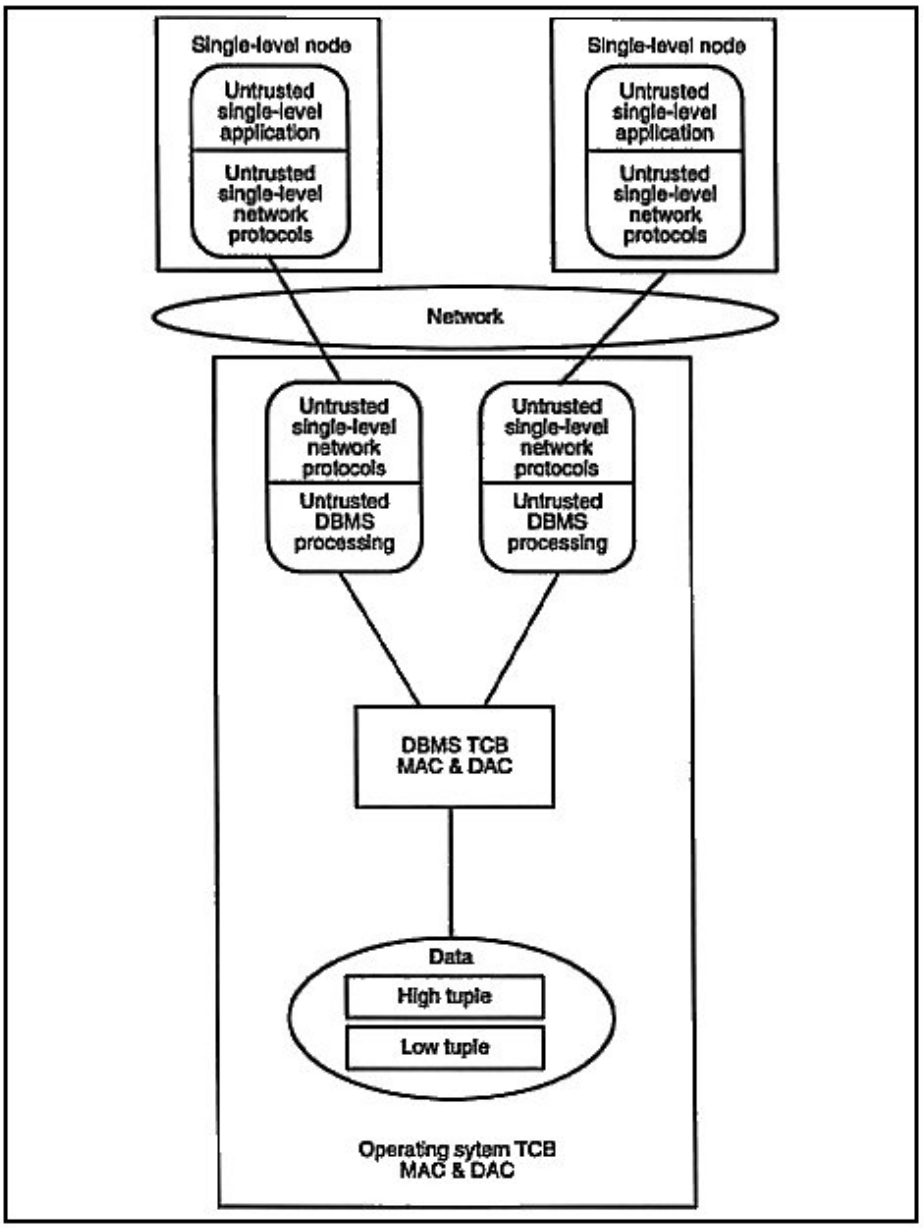

Figure 2. ASD trusted server architecture.

A better approach — and the one used for ASD — is provided by the fact that the underlying ASD operating system supports the Biba integrity policy [BIBA77]. This policy assigns integrity labels to subjects and objects. These labels are analogous to the security labels associated with classification. Under the Biba integrity policy, a subject can write into a protected object only if the integrity level of the subject dominates the integrity level of the object. To protect the ASD data, the operating system object that contains that data is assigned a DBMS integrity compartment label. This label restricts write access to the object to only those subjects that have a dominant DBMS compartment. Since the ASD TCB is the only subject that has this integrity compartment, no other subject can write directly into the operating system object that contains the ASD data. Of course, other subjects can use the facilities of the ASD TCB to store data tuples in the object, but only under the control of the ASD TCB.

Network architecture. ASD is implemented as a trusted server, as shown in Figure 2. Under the trusted server approach, the basic ASD processing engine operates on its own node of the network. It serves other nodes that are likely to be single level, but that operate at different security classifications. ASD provides the multilevel engine that facilitates sharing among levels. A top secret node could retrieve data from the ASD trusted server, where the data was entered from an unclassified node.

To provide network access means that the required network protocols must be supported in a secure manner. The network protocols supported on the ASD server are TCP, IP, and SLIP (Serial Line IP). SLIP permits the TCP/IP protocols to be used over serial lines, such as phone lines. This allows ASD to be used in a tactical environment.

Since the network protocol software actually handles the data, this software will become security relevant if it concurrently or sequentially handles data that is classified at multiple access classes. If, for example, a top secret message is sent to a top secret node, followed by an unclassified message sent to an unclassified node, the potential exists for the software to leak some of the top secret data from the top secret message into the unclassified message. This would lead to a write-down of information, in violation of the mandatory security policy.

One solution to this problem is to make the network protocol software trusted. This solution was rejected, since it would increase the size of the TCB and its complexity, due to the high complexity of network protocol software.

The solution adopted was to use a separate network protocol instance for each security level. Conceptually, a separate stack of network protocol software was associated with each of the two serial ports of the Sun, and each of these ports was connected to a node operating at a different access class.

LDV

Lock Data Views (LDV) is a design for a trusted DBMS that is to run under the control of LOCK (Logical Coprocessor Kernel). LOCK represents a hardware and software approach to providing a secure operating system. LDV is being developed by the Secure Computing Technology Corporation (formerly Honeywell Secure Computing Technology Center), under the sponsorship of the Rome Air Development Center (now Rome Laboratory).

The following LDV discussion is divided into two major sections. The first section describes the security policy and features provided by the system, and the second describes the LDV architecture. (Material presented here was taken from work by J.T. Haigh et al. [HAIG88, HAIG90, HAIG90a, HAIG91].)

LDV security policy and features. This section will describe the LDV mandatory and discretionary access policies.

LDV mandatory access policy. LDV, like SeaView, provides an element-level granularity of control. This means that each element of a tuple can have a distinct and possibly different security classification. The LDV security model includes not only the security policy that is enforced by LDV, but also a description, at an abstract level, of how LDV will enforce the policy.

The first requirement of any secure system is to label the objects. In contrast to the two DBMS systems previously considered in this essay, LDV labels data based on three different criteria: name-based, content-based, and context-based.

Using *name-based object labeling*, the name of the database object determines its security label. For example, an entire relation or all of the data of one attribute of a relation could be uniformly labeled with an access-class label. This is the basic type of labeling provided by the other two systems. For this type of labeling, neither the value of the data nor its context must be considered in determining its access class.

Under *content-based* (or *value-based*) *object labeling*, the value of the contents of a particular data item determines the access-class label of the item. Thus, salaries greater than $50,000 might have an access-class label of Secret, while salaries less than or equal to $50,000 might have an access-class label of Confidential.

With *context-based object labeling*, the label associated with the data is a function of the other data that can be associated with it. Thus, while a salary alone or a name alone might be considered Unclassified, the combination of a salary with a name might be considered Secret.

The object labels are used to label both data stored and data retrieved from the database. In the case of name-based labeling, all data entered

into a named object will inherit the security level of the object. Likewise, all data retrieved from an object will bear its security label. If data is retrieved from multiple objects, then the security label associated with the response will be the least upper bound of the security labels associated with all of the data accessed to make the retrieval.

Data cannot be entered into an object labeled with name-based labels unless the current access class of the user is identical to the access class of the object. Thus, with name-based labeling a user cannot enter data that the system considers more sensitive than the user's current access class. The DBMS will prevent the user from writing data into any DBMS object whose access class is not equal to the user's current access class. However, such is not the case with the other forms of data labeling.

Since LDV supports value- and context-based security labeling, the potential exists for a user to enter data whose access class strictly dominates the access class at which the user is currently interacting with the system. When the data is entered, the system may detect that the value of the data requires an access-class label that strictly dominates the user's current access class. This is not possible under the current designs of the other two systems, since the access class of the data entered is taken to be identical to the user's current access class.

To address this possibility, the LDV design uses a maintenance level and a storage level. The maintenance access level is the level at which the data was originally entered and at which any updates or deletions of the data must occur. The storage access level reflects the actual access class of the data based on the value and context classification rules. The data will actually be stored by the system at the storage level.

It should be recognized that while the maintenance level is the level at which data can be updated or deleted, this does not mean that data so entered can be viewed at this level. Viewing the data, once entered, must be by a subject whose current access class dominates the storage access level of the data. Ultimately, what will be required is a multilevel workstation with which a user can view data through two windows. One window would have an access class that dominates the storage level, providing the ability to view data. The other window would have an access class equal to the maintenance level, providing the ability to update data whose maintenance level is equal to the access class of the window.

To support the element-level granularity of control, an access-class label is associated with each element. A maintenance-level label is associated with each tuple of the relation. Thus, the tuple is the finest granularity at which database maintenance can occur.

The maintenance-level label plus the attributes that comprise the key for the tuple form the enhanced key that uniquely identifies each tuple in the relation. All elements that comprise the enhanced key are stored at the maintenance level of the tuple.

Since all of the data within a tuple is entered from the same access class (the access class of the maintenance level), polyinstantiated elements do not occur. There will not be any null fields that the user will unwittingly attempt to fill, thus leading to a polyinstantiated element.

LDV provides the capability for users to create a new tuple that is substantially the same as an existing one, differing only in one or more element values. To form this new tuple, the user can specify which values of the new tuple are to be derived from the original tuple. For those elements of the new tuple that are to be identical to elements in the previously existing tuple, pointers are used for the element values, rather than the actual element values. This means that any change to these fields in the previously existing tuple will be reflected in the new tuple. For those elements of the new tuple that are to have values distinct from the previously existing tuple, the user can store new values in the appropriate elements of the new tuple.

With this capability, the same effects that are provided by SeaView's polyinstantiated elements can be obtained. While the polyinstantiated elements due to lower access class users updating null values are avoided, as has been noted, the LDV pointers provide the means to create lower access class cover stories, and then create higher access class tuples that reflect some classified operation. However, since the combination of key attributes and storage class must uniquely identify the tuple, any additional tuples that have the same key attribute values must be created with different storage levels.

Access to an LDV object is governed by the Bell-LaPadula security model. Subjects operating on behalf of a user bear an access-class label. A subject can read an LDV object if the access class of the subject dominates the access class of the object. A subject can write only those objects that have an access class equal to the access class of the subject.

The access-class label of the data accessed by the user is based on three security considerations: the access-class label associated with the named DBMS object, the access-class label associated with the value that is being retrieved, and the access-class label associated with the context in which the data is being retrieved. A subject will be able to retrieve the data only if the access class of the subject dominates the most dominant access-class label associated with the data retrieved.

To support the context retrieval constraints, LDV stores a history of the retrievals that have been performed for a particular subject. Each additional retrieval is considered in the context of all previous retrievals maintained by the LDV history mechanism.

As an example, assume that salary data taken alone is Unclassified and name data alone is Unclassified. However, the combination of name and salary data is considered to be Secret. This means that a user could retrieve a list of names, but once these names are retrieved, the user will not be able to retrieve a list of salaries. Or the user could retrieve a list of

salaries, but once these salaries are retrieved, he would not be able to retrieve a list of names. In this case, it is assumed that both the name list and the salary list include a unique identifier that the user could use to associate the names with the salaries.

The problem with a history mechanism is that it can grow without bounds, eventually exceeding the size of the original database. To address this concern, the LDV design allows the history file to be purged when desired by appropriate authorities. Of course, this means that it is possible for a user to record the salary data off line and then wait until the time period maintained by the history mechanism has been exceeded, and then request the associated name data. This is a problem generic to context-based access control, which considers the total temporal context.

LDV relational integrity constraints. LDV enforces entity integrity, which requires the following:

1. The enhanced key, which consists of the key attributes and the maintenance level, uniquely identifies a tuple in the database. This means that no two tuples can have the same enhanced key.
2. No attribute that is part of the enhanced key can be null.
3. The attributes that comprise the enhanced key are uniformly classified at the maintenance level.

The first of these entity integrity rules eliminates the need for multivalued dependencies. which is found in SeaView. The reason is that there is only a single tuple associated with each key and maintenance level combination.

LDV architecture. Internally at the lowest level of abstraction, LDV maintains a relation-level granularity of control, with each relation having a uniform access class. The actual protection of each LDV relation is provided by LOCK, the underlying operating system with its associated hardware support. Each relation is stored in an operating system object with the same classification as the relation.

For data whose access class is derived from name-based labeling or value-based labeling, the data is stored in operating system objects whose access class is that of the data label. However, for data whose context-derived access-class label is higher than the access class of the data components taken separately, the components will be stored at the access class that applies to the component alone, not at the access class of the context-based label. The LDV design ensures that users accessing one of the components will not have any access to the other component, based on the current state of the access history. This is called a store-low-and-upgrade approach.

The alternative is to store such relationships high and then downgrade the individual components. The problem with this approach is that the data to be downgraded could have become contaminated by other data whose access class is high. There is the potential that this other high-level data could then be downgraded when one of the components of the context-relevant object is downgraded. The LDV designers argue that the store-low-and-upgrade approach avoids this vulnerability. Of course, the upgrade software must work correctly; otherwise, sensitive data will be compromised. In reality, if the DBMS TCB is the only software that is allowed access to the data, it probably makes little difference which approach is used.

While all of the DBMS systems presented can be described in terms of their functional decomposition and the partitioning of the various functions into different privilege domains, LDV provides the most extensive privilege partitioning of any of the systems discussed. This partitioning is provided by the type enforcement of the underlying LOCK trusted operating system. Type enforcement is somewhat analogous to strong typing in a programming language such as Pascal. One of the services provided by type enforcement is used to ensure that only LDV subjects can access LDV objects. However, the LDV type enforcement mechanism goes further than this by providing a means to implement a relatively fine granularity least privilege within the LDV software.

Through type enforcement, only subjects of a particular type can perform the operations associated with an object of the specified type. This represents a finer control than is found in ASD, in which all trusted DBMS code is, in effect, part of a single DBMS trusted process. It also represents a finer granularity of control within the DBMS code than is found in SeaView, in which all of the trusted code is part of the operating system trusted computing base.

In LDV, least privilege is carried even further, in that all LDV processes, whether trusted or not, run at a particular security level. In some cases, because of the need for these processes to see data that could have been entered at any security level, they operate at system high. However, those trusted processes that do not require the global viewability provided by system high can operate at a level that is less than system high.

In addition to providing type enforcement, LDV partitions the supported DBMS operations into what are called “pipelines.” Pipelines are sequences of domains that perform operations. Domains provide a set of object types and associated operations that can be used by processes operating within the domain.

LDV supports three pipelines: query/response, data input/update, and database definition/metadata. While ASD and SeaView also support these operations, the code that implements them is not explicitly partitioned into pipelines. In a sense, the pipeline represents a clustering of

subjects and data that can interact with each other to perform particular types of database operations. The type enforcement is the glue that binds these various processes and data together to form a pipeline.

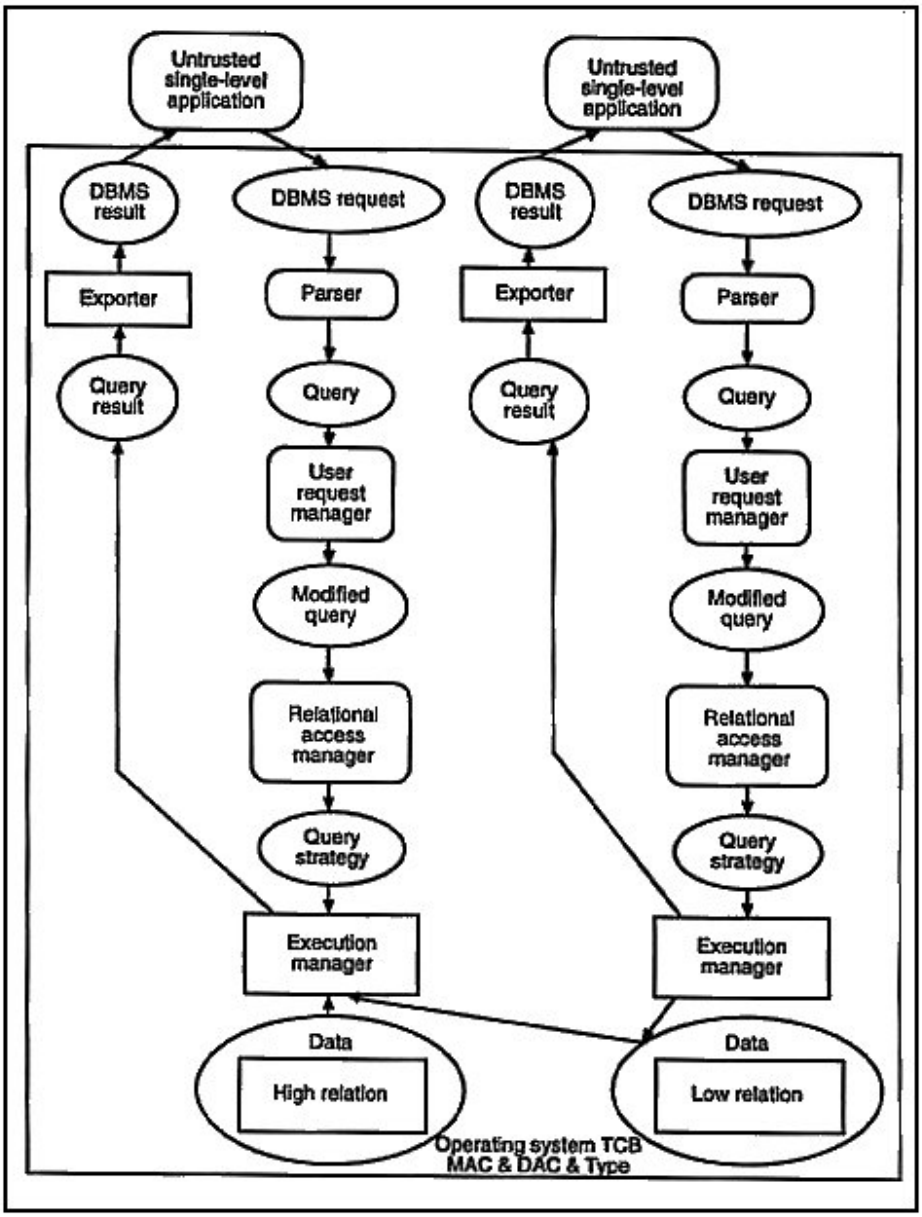

Figure 3. LDV architecture: Query/response pipeline.

The LDV architecture for the query/response pipeline is shown in Figure 3. Within LDV, trust is defined as the ability to override the *-property. Those domains that require trust are indicated by rectangular boxes with square corners. The rectangular boxes with rounded corners are untrusted domains. It can be observed that most of the domains in each of these pipelines consist of untrusted processes.

As shown in the figure, the query/response pipeline consists of five domains: parser, user request manager, relational access manager, execution manager, and exporter. The data in the query/response pipeline is partitioned into the following six types: DBMS request, query, modified query, query strategy, query result, and DBMS result. Table 1 is an example of a domain definition table for the query/response pipeline. In this example, the write-type of access is not partitioned into its various types.

The table shows that the LDV pipeline is simply a joining together of various domains by the access privileges that each domain has to its input and export data. This provides a finer level of granularity in the implementation of the DBMS code than is possible with either ASD or SeaView. Within ASD, the DBMS software is either trusted to enforce mandatory and discretionary policy or untrusted. Within SeaView, the various GEMSOS rings could be used to provide some additional partitioning, similar to Multics, but not at as fine granularity as in LDV.

Table 1. LDV query/response pipeline domain definition table.

Types	**Subjects**					
	Client	Parser	User request manager	Relational access manager	Execution manager	Exporter
	Untrusted	Untrusted	Untrusted	Untrusted	Some trusted code	Some trusted code
DBMS request	Write	Read				
Query		Write	Read			
Modified query			Write	Read		
Query strategy				Write	Read	
Query result					Write	Read
DBMS result	Read					Write

As will be noted, while most of the LDV pipeline consists of untrusted domains and associated untrusted code, even this untrusted code is partitioned through the LOCK type mechanism. To allow this code to be untrusted, each of these domains operates at the level of the user that submitted the query and will receive the response. Thus, there will be multiple instances of this code, one for each level at which users are interacting with the system.

In the query/response pipeline, the only trusted processes exist in the execution manager and the exporter. The execution manager addresses context-dependent access policy and inference. The acceptability of data to be released to a user is determined by the access class of the user, the access class at which the data is stored, and the context in which the data is released. This context is not limited to just the data that is requested as part of the same query, but also includes data that has been released in the past. Information about what has been previously released is contained in a history database, which keeps track of accesses by users or groups. In addition, the execution manager is intended to prevent the release of data into a group where the new data, combined with that accessed in the past, would permit inference of unauthorized data. The history manager runs in the execution manager domain at database high. Database-high operation is required so that the history manager will be able to see all of the history. It must be trusted to properly interpret the history and not to downgrade classified information when it reports the result of its history consultation to the level at which the query response is to be provided.

Trust in the exporter is required if the response needs to be reclassified to a higher level, due to aggregation constraints based on the cardinality of the response. Cardinality limits might, for example, permit the release of up to 10 names and phone numbers from a classified agency, but not the release of greater than 10 names and numbers. The release of too many names and phone numbers could reveal sensitive information about the capabilities of the agency. If the release of data must be prevented due to such cardinality concerns, LDV proposes to release a message to the user indicating that such information cannot be provided, rather than releasing the results of the query.

The input/output pipeline consists of the following four domains that are a subset of the domains in the query/response pipeline: parser, user request manager, relational access manager, and execution manager. The data in the input/output pipeline is partitioned into the following five types: DBMS request, which feeds into the parser; update, which joins the parser to the user request manager; updates and levels, which joins the user request manager and the relational access manager; update strategy, which joins the relational access manager to the execution manager; and the DBMS results, which provides the output to the user from the execution manager.

The only trusted code in the input/update pipeline is contained in the user request manager and the execution manager. The user request manager contains an upgrader that raises the level of data input, based on content or context constraints. The upgrader determines the level at which data entered into the database should be stored. The upgrader is a database-high process, since it must be able to see all appropriate constraints. These constraints may themselves be classified at a level that dominates the level at which the data is entered, or even the level at which the data should be stored after applying the appropriate classification constraints. The execution manager is trusted to insert the data into the proper files.

LDV decomposes each relation into multiple relations, each representing a single attribute. These decomposed relations are then stored at the level of the attribute. However, LDV stores data that may be subject to cardinality-based aggregation constraints or context constraints at the lowest level at which this data could be accessed (ignoring these types of constraints), rather than at the highest level that could apply, based on the constraints. While the data is stored at a low level of classification, it is accessible only by appropriate LDV subjects. As further protection, all data stored on disk is encrypted.

The final pipeline is the database definition/metadata pipeline. This pipeline is used to define new relations and the associated security data required to enforce access to these relations.

SeaView, ASD, and LDV comparison

The three systems described in this essay can be compared in terms of their granularity of control, the nature of the security policy that they enforce, and the nature of their internal architecture.

Granularity of control. This section will compare two aspects of granularity of control. First, the stored and exported granularity of control of the DBMSs will be compared. Then, the match between operating system object and DBMS object will be compared.

Stored and exported granularity of control. Table 2 shows the relationship between the security granularity of the DBMS storage object and the security granularity of the view that is provided to the user. For SeaView and LDV, there is a difference in granularity of view and storage. This means that software — in this case, untrusted software — must be provided to translate the storage granularity into the view granularity. To support this translation, there must be decomposition and reconstitution algorithms and software with the goal that this process be lossless. As noted in the polyinstantiation discussion for SeaView, there are open research questions in this area.

ASD, on the other hand, does not have this problem since no translation is required between what the user views and what is stored. Also, since the granularity of control is the tuple, there will not be any polyinstantiated elements. While ASD could impose polyinstantiation integrity rules, those rules would not be connected with the decomposition/reconstitution algorithms as they are in SeaView, since decomposition and reconstitution are not required.

DBMS/operating system granularity match. Both SeaView and LDV store their objects in operating system objects that have the same classification as the DBMS objects stored. This results in an operating system object that contains DBMS objects uniformly classified with respect to access class. As will be noted in the next section, this is why neither SeaView nor LDV enforces mandatory security. However, the operating system objects do not contain DBMS objects that are uniform in their discretionary access. Thus, the DBMSs must provide the discretionary enforcement.

Table 2. Relationship between the security granularity of the DBMS storage object and the security granularity of the view provided to the user.

View Object	**Storage Object**			
	Element	Tuple or row	Attribute or column	Relation or table
Element				SeaView LDV
Tuple or row		ASD		
Attribute or column				
Relation or table				

In contrast, the underlying operating system object which supports ASD does not contain DBMS objects that are uniform in access class. The operating system object contains DBMS objects that are classified at multiple access classes. Thus, the operating system cannot provide either mandatory or discretionary controls to these DBMS objects. All the operating system can do is ensure that only the DBMS TCB has access to those operating system objects that contain DBMS objects.

Security policy enforced. Both SeaView and ASD enforce a name-based access policy. This means that the name of the object is adequate to determine its security. For ASD, the access-class label is associated with each tuple, so in a sense it is associated with the tuple identifier or name.

While LDV enforces a name-based access policy, its design also calls for the enforcement of both a context-based and value-based policy. In addition, the initial LDV design calls for the system to address the inference problem, including the problem of restricting the release of data based on its cardinality.

Internal architecture. Relying on an extension of the taxonomy presented elsewhere [HINK90], this section will classify the various systems in terms of the nature of the security policy enforced by each of the major components of the DBMS, and in terms of the way the internal architecture is constrained by least privilege.

In Table 3, the horizontal axis indicates the nature of the DBMS policy enforcement provided by the DBMS code. The vertical axis indicates the nature of the partitioning inherent in the trusted code (both operating system and DBMS).

Table 3. Policy enforcement provided by the DBMS code and the partitioning inherent in the trusted code (OS: operating system).

	Null DBMS TCB: No trusted code	DBMS TCB enforces discretionary policy	DBMS TCB enforces mandatory and discretionary policies
DBMS TCB and untrusted code partitioned into multiple domains		LDV	
DBMS TCB partitioned into multiple domains			
DBMS and OS TCB in different domains	Hinke-Schaefer	SeaView	ASD
DBMS and OS TCB share same domain			Coresident DBMS

As noted, both the SeaView and LDV systems enforce only discretionary access policy. They are thus members of the class of trusted DBMSs in which the DBMS TCB can only subset the privileges provided by the underlying operating system. Since in both cases the DBMS contains trusted code that provides additional discretionary controls, these are classified as subset privilege systems in which the DBMS trusted component is nonnull.

LDV provides a greater degree of partitioning of the privileges of its DBMS software, both trusted and untrusted, than does either ASD or SeaView. ASD, on the other hand, represents a trusted subject approach, in that both the DBMS TCB and the underlying operating system TCB have responsibility for enforcing mandatory and discretionary security. The DBMS TCB performs this enforcement on the finer granularity DBMS objects (for example, tuples) that are contained within the operating system object, which is classified at the least upper bound of the DBMS data stored in the object.

Table 3 also contains two other generic DBMS architectures that provide a point of contrast to the research prototypes discussed in this essay. The Hinke-Schaefer approach relied totally on the underlying operating system for its security [HINK75]. Thus it has a null DBMS TCB. Nevertheless, it is shown in the row "DBMS and OS TCB in different domains": Even though the DBMS TCB is null, the untrusted DBMS code executes in a domain distinct from the operating system.

The second generic system shown is the coresident DBMS. Under this architecture, both the DBMS TCB and the operating system TCB share the same protection domain. In a sense, the DBMS TCB is just an extension of the operating system TCB.