Essay 22

Integrity in Multilevel Secure Database Management Systems

Catherine Meadows and Sushil Jajodia

Integrity is usually considered to be at odds with security in multilevel databases. Integrity constraints enforce conditions on relations between data, while security constraints enforce separation between data. If an integrity constraint is defined over data at different security levels, a direct conflict results. However, the solution is not to sacrifice the integrity constraint altogether. Compromise solutions can often be found that guarantee some, although not all, of the desired results of the constraint. In this essay we will show that by dividing the desired goals of integrity into three areas — consistency, correctness, and availability — one can often find solutions to integrity problems that achieve some, if not all, of the goals without sacrificing security.

The rest of this essay is structured as follows. After a brief overview of security issues in multilevel databases, we discuss integrity and describe the three integrity properties: consistency, correctness, and availability. Then we discuss integrity in relational databases and integrity of transactions. We also discuss integrity of object-oriented databases.

Security issues in multilevel databases

Most security violations arising from the application of integrity constraints to multilevel secure databases do not result in the direct revelation of data. Instead, security violations arise from the fact that integrity constraints defined over data at more than one security class can provide channels by which information at a high security class can be passed down to users at a lower security class. These channels are of two kinds:

- *Inference channels.* A user at a low security class can use his knowledge of the low security class data and of the constraint (if it

is made available to him) to infer information about high security class data also affected by the constraint.

- *Signaling channels.* Signaling channels are divided into storage channels and timing channels.

  In a storage channel, if satisfaction of an integrity constraint requires that changes to data at a high security class be reflected indirectly in the value of data at a lower security class, a Trojan horse program embedded in a process at a high security class could encode high data as low data by varying the high data involved in the integrity constraint so that it produces detectable changes in the low data. Such a channel could be used to pass on not only information directly affected by the constraint, but any other information the Trojan horse has access to.

  In a timing channel, if an integrity constraint depends on data at both high and low security classes, a Trojan horse program could encode high data by varying the time it takes to make the high data necessary for verification of the integrity constraint available. Again, such a channel could be used to pass on any information to which the Trojan horse has access.

Many of the conflicts between security and integrity in multilevel secure databases arise out of the fundamental incompatibility of what we will call the *basic security principle for databases* with the database integrity properties of consistency, correctness, and availability [MEAD88a]. The basic security principle is as follows: *The security class of a data item should dominate the security classes of all data affecting it.* The reason for the basic security principle is clear: If the value of a datum can be affected by data at levels not dominated by its own, information can flow into the item from those other levels.

The database integrity properties are defined as follows:

- *Consistency.* A database is consistent if, whenever two different methods exist for deriving a piece of information, a request for that information always yields the same response no matter what method is used.
- *Correctness.* A database is correct if all data satisfy all known constraints.
- *Availability.* A database is available if the data in it can be made available to any authorized user, and any user who is authorized to enter or update data may do so.

The three integrity properties are somewhat overlapping. However, since they often can be traded off against each other, we find it useful to distinguish between them.

In the remainder of this essay, we identify the various principles as they come into play. We also identify the trade-offs among them, and discuss techniques for optimizing multilevel secure databases with respect to the various properties.

Basic integrity

In this section, we consider three basic criteria for database correctness given by Date [DATE86], describe the conflicts that arise between these criteria and the basic security principle, and describe the solutions that have been recommended. This section is essentially a summary of the discussions in Hinke and Schaefer [HINK75] and Denning et al. [DENN87], with some comments of our own.

Key integrity. Every tuple in a relation must have a unique key. This constraint does not cause a problem when data is classified at the relational or column level, since in that case all keys in a relation are of the same security class. But consider a low security class user who wants to enter a tuple in a relation in which data are classified at either the tuple or the element level. If a tuple with the same key at a higher security class already exists, then to maintain key integrity, the DBMS must either delete the tuple or inform the user that a tuple with that key already exists. In the first case, data availability is reduced for the high user, possibly to an unacceptable degree. In the second case, the basic principle of database security is violated; the existence of high data is allowed to affect the existence of low data.

Similarly, if data is classified at the element level, the low user cannot be warned that a high-level element already exists when he attempts to insert a low-level element, nor can the high-level element be deleted. Thus two versions of the tuple will exist: one with a high-level value in the element, and one with a low-level value. If data is classified at the element level, the problem is potentially even worse. For then, not only can more than one tuple exist with the same key, but so can more than one element for each attribute of a given tuple.

The strategy of allowing more than one version of a tuple to exist to prevent security violations was first identified by Hinke and Schaefer [HINK75]. The name under which it is known today, *polyinstantiation*, was coined by the SeaView project [DENN87].

Polyinstantiation has been treated in depth in Essay 21, so we will not discuss it in detail here. However, we will briefly review the different approaches that have been taken to polyinstantiation and show how they trade off consistency, correctness, and availability.

It is generally agreed that the most serious threat to integrity comes from polyinstantiated elements. For example, consider the flights relation

in Figure 1, in which the attributes are Flight Number, Destination, and Payload, and each element has its own classification.

Flight Number		Destination		Payload	
701	U	Persian Gulf	S	Nuclear weapons	TS
403	U	Newcastle	U	Coals	C

Figure 1. Flights relation.

Now consider Flight 701 in Figure 1. A low user will see that flight as having null entries in the Destination and Payload attributes. Thus that user, or two such users, could enter their own values for these elements. As Figure 2 shows, this could result in four possible tuples from the high user's point of view. In other words, the number of possible tuples grows exponentially with the number of polyinstantiated elements.

Flight Number		Destination		Payload	
701	U	Persian Gulf	S	Nuclear weapons	TS
701	U	Canada	U	Rugs	U
701	U	Persian Gulf	S	Rugs	U
701	U	Canada	U	Nuclear weapons	TS

Figure 2. Four tuples for Flight 701.

In the original SeaView approach to the problem, all possible tuples were allowed to exist: Thus correctness was sacrificed, since only one of the tuples could be the correct one. In later work, means have been developed for limiting the number of tuples displayed: Lunt and Hsieh [LUNT91] modify the original SeaView algorithm so that polyinstantiation can be limited to a certain extent when two or more fields in a tuple are updated in a single transaction. Jajodia and Sandhu [JAJO91b] give an algorithm that will implement any desired display policy on a per-relation basis. Haigh, O'Brien, and Thomsen [HAIG91a] present an algorithm that allows one to display no more than one tuple per security class.

All of the above techniques limit the display of polyinstantiated tuples. However, correctness is still sacrificed, since even with the most restrictive algorithm more than one tuple with the same key may exist. In some

cases, we can take advantage of this feature by having the low-level tuples be *cover stories* designed to mislead users about the existence of classified data. However, this will not always be the case. As a matter of fact, in some cases the low-level information may be more accurate, if, for example, it is more recent.

These problems are solved by some recent algorithms that eliminate polyinstantiation altogether. In the SWORD database system [WOOD88] a user, before he can enter a high-level element, must log in at the lower level and upgrade that element. Sandhu and Jajodia [SAND91] present an algorithm with a similar effect that does not rely on trusted code, and is less restrictive in its handling of tuples at different security levels.

These algorithms assist one in achieving correctness: No more than one version of an element can exist at any time. However, they do this at the cost of availability. In both cases, restrictions are made on the way in which a user can enter high-level data. The SWORD algorithm is even more restrictive in that, once a high-level tuple has been created by upgrading the key of a low-level tuple, no more low-level tuples can be entered into the relation, since allowing this would either potentially violate key integrity or create a signaling channel. Since high-level tuples are created by upgrading the key of low-level tuples, this means that no new tuples at all can be entered once a high-level tuple has been created.

Entity integrity**.** Every tuple must have a nonnull key. When data is classified at the relational or tuple level, security considerations do not cause a problem with entity integrity. However, if data is classified at the column or element level, problems can arise if the security class of the key is higher than or incomparable with the security classes of other elements in the relation. Thus it is necessary to require that the security class of any field in the key be dominated by the security classes of all data in the relation, as is recommended by Hinke and Schaefer [HINK75]. (Note that this requirement means that all fields in the key must be of the same security class.) Depending on the way relations are defined, this approach may reduce data availability. For instance, consider the following example, similar to one discussed by Hinke and Schaefer [HINK75]: Let *R* be the relation *ABCD* with key *AB*. Suppose that *A* and *D* are highly sensitive, that *B* and *C* are not, and that *C* functionally depends only on *B*. To maintain entity integrity, *C* must be classified at a level at least as high as *A*. This problem can be avoided by breaking *R* into two relations, *ABD* and *BC*.

The solution recommended by Hinke and Schaefer is to store data so that as little redundancy as possible exists in a relation; they suggest using third normal form.

Referential integrity. If an attribute in a relation is designated as a foreign key for another relation, then any tuple appearing in that relation

either has a null value as its entry in that attribute or there is a tuple in that other relation with that entry as the key. In this case, database integrity can be violated when any of the four approaches to data classification is taken. For example, suppose that data are classified at the relational level, and that a foreign key in a relation with a low security class refers to a tuple in a relation with a higher security class. To the user with a lower clearance, the key would appear to be dangling. Similar problems can occur when data are classified at the tuple, column, or element level.

The obvious method for avoiding dangling keys is to require that, if an element A appearing in relation R is designated as a foreign key for relation R', then the security class of A in R must dominate the security class of A in R'. However, as has been pointed out [DENN87], if the security class of A in R strictly dominates the security class of A in R', this has the potential of creating signaling channels. Suppose that an element A appearing in tuple T in relation R is designated as a foreign key for R', and that T' is the tuple containing A in R'. There are two ways of enforcing referential integrity when a user attempts to delete T'. One way is to delete A from T automatically. The other is to prevent the user from deleting T' without first deleting A from T. If T is classified at a higher level than T', the second method opens a signaling channel, since, by repeatedly removing and inserting a tuple T, a Secret process could signal information to an Unclassified process that repeatedly attempts to remove and insert T'. The first method of automatically deleting A from T when T' is deleted does not open any signaling channel, since only data at the higher security level is affected when this approach is taken.

As Gajnak has pointed out [GAJN88], further problems can arise when referential integrity is enforced in the presence of polyinstantiation. When an element refers to a polyinstantiated tuple, which tuple does it refer to? Moreover, when referential integrity is enforced across several links (that is, if A is a foreign key for B, which contains C, which is a foreign key for D), the number of possible references grows exponentially.

The approach taken by SeaView was to enforce referential integrity only over the same security class. Thus correctness is sacrificed to security, but is still enforced in the cases in which security is not violated. Burns [BURN90] has suggested enforcing interclass referential integrity selectively in cases in which the signaling channel can be monitored or for some other reason is deemed not to cause a serious threat.

Referential integrity has not attracted the same degree of attention as key integrity. This is probably because until recently most commercial DBMSs did not enforce referential integrity either. Thus there was no point in providing a feature in multilevel DBMSs that was not even provided in single-level DBMSs. However, as Burns has pointed out [BURN90], this situation is now changing: Commercial DBMSs are now starting to offer referential integrity. Moreover, it is easy to imagine

situations in which interclass referential integrity would be useful: Consider a specification for a classified computer system that will use some unclassified components. It is hoped that in the future researchers will start paying the same kind of attention to referential integrity that they have to polyinstantiation, and introduce new solutions in which correctness, consistency, and availability are traded off against each other to provide the best solution for individual database applications.

Atomicity of transactions

In recent years, more and more attention has been paid to concurrency control in multilevel databases. A number of algorithms have been developed that process multilevel transactions so that security and consistency are both preserved. However, there is one aspect of transaction management that has been pretty much ignored — that is, the preservation of failure atomicity, the property that transactions either succeed completely or fail completely. Consider a multilevel transaction that performs some reads and writes at a low security class and some at a high class. Suppose that the transaction has already performed some or all of the low writes, and one of the high reads or writes fails. If the transaction aborts, then the low writes must be undone to preserve failure atomicity. But this will result in a signaling channel. Nor can we delay the low writes until the high part of the transaction is done, since this will also result in a signaling channel.

There are several ways out of this dilemma. One is to allow a transaction to write only at a single security level. This will have the result of limiting availability; the kinds of transactions available to the user will be reduced. Another is to insist upon failure atomicity only at a single level. Transactions would be required to succeed or fail completely at each level, but could be allowed to succeed partially across levels. Thus correctness would be sacrificed for the sake of security. Another approach can be used when the database architecture is of the kind described by Froscher and Meadows [FROS89], in which the database is divided into single-level databases where each database contains all data classified at its level or below. In this architecture, an update to a data item at one level is propagated to all databases at its level and above. In this case, we can require failure atomicity at each database, and thus each database will be correct. However, since we cannot require failure atomicity among databases, it may be that after a while the databases will no longer be consistent with each other. Moreover, the lower the security class of a database, the more up to date it will be.

The effects of these various approaches to preserving atomicity are unknown and will probably depend on the application; they deserve further study.

Object-oriented databases

Unlike relational databases, object-oriented databases do not have a well-defined set of integrity rules. Each object is responsible for maintaining its own integrity policy. However, there is one general integrity principle that is widely agreed upon — that of the integrity of the object itself. An object should be a self-contained unit. In particular, permission to view or modify an object should imply permission to view or modify the entire object. Moreover, there are other integrity issues that arise in object-oriented DBMSs that must be addressed by the database designer. These include decisions as to whether to allow multiple inheritance and how it should be handled, how naming conflicts are handled in class hierarchies, and what should be inherited in a class hierarchy. Although the answers to these questions are implementation-dependent, we need to answer them in ways that will both satisfy the needs of the application and preserve security.

In this section, we look at both kinds of questions. In the first part, we discuss work that has been done in maintaining object integrity in multilevel object-oriented DBMSs. In the second part, we discuss implementation-dependent integrity issues and their relation to security.

Object integrity. Earlier attempts to develop models for secure object-oriented DBMSs [THUR89a, LUNT90d] modeled objects as multilevel entities, analogous to tuples in a multilevel relational database with element classification. Rules were developed for ensuring integrity in the sense that there should be no part of an object that was inaccessible to everybody. One such rule, for example, was a rule saying that the level of any component of an object should dominate the level of the object itself. However, more recently some researchers have come to believe that, in order to promote object integrity, objects should be single level [JAJO90a, MILL92]. In an object-oriented system, the objects should be self-contained entities. Allowing some users to deal with pieces of an object but not the entire object violates this principle.

Despite the need for integrity of objects, the existence of entities in the paper world that are both multilevel and have the characteristics of objects suggests that some way of representing multilevel objects is needed, although such objects should probably not be the basic components of the system. For example, a report or message will have a single classification, but each paragraph of the report will have its own label, dominated by the label of the report. If we modeled the report in an object-oriented system, then clearly it would be considered as an object. But if we made it single level, that would mean every time someone wanted to make a low-level paragraph of the report available, it would have to be downgraded to the appropriate level, with a resulting risk to security.

Thus, it would seem useful to have a way of modeling multilevel objects, perhaps as composite objects built out of single-level objects.

Probably the earliest attempt to model multilevel objects was that of the NRL Secure Military Message System [LAND84]. The SMMS did not use object-oriented concepts explicitly, but its security policy was formulated in terms of abstract data types, whose use of inheritance was a precursor of the notion of inheritance used in object-oriented systems. The SMMS has been interpreted as an object-oriented system [MEAD92]. In the SMMS, entities are either containers or objects. Each object has a single security label. Containers may contain other entities, while objects may not. Thus objects are the basic building blocks of the SMMS. Each container is given a container label that gives an upper bound on the labels of the entities (either containers or objects) that may be stored in it. If a container is marked Container Clearance Required (CCR), then only individuals cleared to the level of the container may access the entities stored in it via that container; otherwise, one may access any entity inside the container as long as one is cleared to see the entity. Thus, for example, an individual with a Top Secret mail file could still see the Unclassified mail in the file, even if he logged into the system from a nonsecure terminal. However, if a message was marked Container Clearance Required, one would need to be cleared to the level of the message even to read its unclassified paragraphs.

Containers give a natural interpretation of the paper world. However, it is difficult to construct a CCR container securely and conveniently if one uses the Bell-LaPadula paradigm on which most existing secure systems are based. To construct it without illegal write-downs, one would have to assemble each set of components at each security level separately and insert it into the container. This is acceptable if one is constructing and adding to a mail file, but it could be inconvenient if one is working on, say, a message. The other approach would be to construct the entire container at once at the level of the container. In such a case, one would have no assurance of the integrity of the labels of the components.

That these problems arise is not surprising, since the SMMS was designed as an alternative to the Bell-LaPadula model and was motivated largely by the fact that the Bell-LaPadula model failed to provide many of the features taken for granted in the paper world. However, we highlight them because they arise not when dealing with the SMMS; they are problems that one must confront in one form or another whenever one is dealing with multilevel entities in a Bell-LaPadula context.

Jajodia and Kogan [JAJO90a] have described a means of implementing multilevel objects using inheritance. A multilevel object is represented by a number of single-level objects. Each attribute of the object is defined at a single security level. Each single-level object inherits its lower level attributes from a lower level object. Thus, although a user at a given level

will see the single-level object, he will also see the lower level attributes inherited from the lower level objects.

A method for constructing certain kinds of multilevel entities is presented by Lunt and Millen [MILL92]. This technique is used when the relation between two objects may be classified at a level higher than the objects themselves. In such a case, one creates a relationship object classified at the higher level that contains the information about the relation. No integrity constraints are enforced here. As a matter of fact, enforcing integrity constraints could result in inference and signaling channels, since if whenever one deleted an object the object to which it was related was automatically deleted, this might allow someone to deduce the existence of the relation. Thus, in this case correctness is sacrificed to preserve security; a description of the relationship is allowed to remain, but one or more of the related objects no longer exist.

Much remains to be done in the development of means to ensure the integrity of multilevel objects. It is still not clear what kind of integrity these objects should have, or how we should implement it. Moreover, as was shown in the SMMS example, there are certain security and integrity problems that arise when constructing multilevel objects that may prove intractable, unless we look for new points of view from which to approach the security problem.

Implementation-dependent integrity considerations. Although there are integrity principles that apply to object-oriented systems as a class, there are still a number of integrity considerations that must be taken into account when designing an object-oriented database that may vary with the application. These include questions such as the following:

1. Should multiple inheritance be allowed? If so, what restrictions should be put on it?
2. What should be done if there is a naming conflict in a class hierarchy?
3. What do objects inherit in a class hierarchy?
4. How is transaction processing managed?

Although these considerations may not be directly related to security, the choices one makes will perhaps require different notions of integrity that will have different effects on security. For example, consider the question of inheritance of instance variables, and the question of what should be inherited. Should one define a security class for an instance variable, and require that it be inherited as well? Spooner [SPOO89] discusses some of the problems that can arise in this case. If object class *B* inherits variable *V* from object class *A*, then its security class must dominate that of *A*. But what happens if *B*'s security class strictly dominates *A*'s security class, and a method of *B* (classified at *B*'s level) at-

tempts to alter a value of *V*? In such a case, we must either not allow inheritance of security classes of instance variables, or only allow methods inherited from *A* (or other object classes of the same security class as *A*) to alter the value of *V*.

Transaction processing is another thorny issue in multilevel object-oriented databases. Objects perform transactions by sending messages to each other. These messages execute methods. Thus, if object *O* wants to read object *U*, it sends a "read" message to object *U*, which executes a read method. If the security class of object *O* strictly dominates that of object *U*, then this causes a problem. In most multilevel systems, reading down is not considered a problem; indeed, in most cases a multilevel system that did not allow reading down would be pointless. But if the object that reads down needs to send a message to the object being read in order to do it, this opens an apparent signaling channel.

Two approaches have been taken to handling this problem. One, that followed by Jajodia and Sandhu, is to attempt to retain at least the appearance of the object-oriented paradigm, but to implement it in such a way that the signaling channel does not arise [SAND92a]. This has its risks, since it is done at the cost of greater system complexity, which makes errors more likely. The other approach, for example, that taken by Lunt and Millen, is to implement a straightforward Bell-LaPadula style version of read and write [MILL92]. This greatly simplifies the model, but at the risk of losing much of the advantage of the object-oriented paradigm.

As we see, although (or perhaps because) integrity issues for object-oriented systems are not as well-defined as they are for relational databases, the conflicts between security and integrity can have even greater impact and be harder to resolve. Moreover, since multilevel object-oriented databases (and object-oriented databases in general) are less well understood than relational databases, these conflicts have not been as thoroughly explored. It is hoped that as more work is done in this area, more of these conflicts will be brought to light and resolved.

Conclusion

In this essay, we have examined the various kinds of threats to integrity that arise in multilevel secure database systems, and the various ways there are to deal with them. In discussing the trade-offs between database integrity and security, we found it useful to think in terms of three database integrity properties — consistency, correctness, and availability — instead of a general notion of integrity. This was because it often turned out that, whenever a conflict arose between database integrity and security, it could be resolved more or less satisfactorily by introducing a trade-off among the three integrity properties instead of between integrity and security. For example, in the case of the key integrity prop-

erty discussed in the section entitled "Basic integrity," it is possible to maximize availability and correctness at some cost to consistency, without any cost to security. Similarly, in the case of failure atomicity of transactions, it is possible to preserve security, correctness, and consistency by reducing availability, or to preserve security and availability by sacrificing either correctness or consistency, depending on the database architecture used.

In general, of all three database integrity properties, consistency is the easiest to sacrifice, perhaps because consistency is the most easily recoverable. When data availability is sacrificed, data may become irretrievably lost, and when data correctness is sacrificed, data may become irretrievably corrupted. However, it is always possible to recover database consistency, as long as well-defined rules exist for identifying the correct version of the data. That the definition of such rules is not always a trivial problem can be seen from the discussion of polyinstantiation. Thus, a goal of secure database research should be to identify cases in which rules for restoring database integrity can be defined in the context of multilevel database security, and to develop means of defining these rules.