# INTEGRITY IN TRUSTED DATABASE SYSTEMS

Roger R. Schell
Gemini Computers, Inc.
P.O. Box 222417
Carmel, CA 93922

Dorothy E. Denning
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

## INTRODUCTION

A trusted computer system is designed to be 'secure' with respect to some well-defined security policy. There are two major classes of information security policy: (1) secrecy policies, which govern the disclosure of information and (2) integrity policies, which govern its modification. Although much of the literature on computer security emphasizes secrecy, for many systems integrity is of equal or greater importance. The DoD Trusted Computer System Evaluation Criteria[1] is careful to encompass (although not require) security policies that include integrity. A trusted computer system is designed to protect 'sensitive information,' which is defined in the Criteria as information that must be protected from "unauthorized disclosure, alteration, loss or destruction."

In databases, the term 'integrity' is interpreted broadly, as illustrated by the following definition taken from Date[2]:

> "The term *integrity* is used in database contexts with the meaning of *accuracy, correctness,* or *validity*. The problem of integrity is the problem of ensuring that the data in the database is accurate -- that is, the problem of guarding the database against invalid updates. Invalid updates may be caused by errors in data entry, by mistakes on the part of the operator or the application programmer, by system failures, even by deliberate falsification. The last of these, however is not so much a matter of integrity as it is of *security* ... The term 'integrity' is also very commonly used to refer just to the special situation ... in which it is possible that two concurrently executing transactions, each correct in itself, may interfere with each other in such a manner as to produce incorrect results."

In this paper, we address all aspects of integrity in that all are essential to the operation of secure database systems.

### Classes of Integrity Policies

There are two distinguishable aspects of integrity policies: whether a given modification of information is *authorized*, and whether the modification results in information that is in some sense *consistent* or *correct*. Authorization is subdivided into two categories: (1) *mandatory integrity authorization*, which is based on integrity classifications, reflecting importance of data, and clearances, reflecting user trustworthiness, and (2) *discretionary integrity authorization*, which is based on users' needs to modify information. Both mandatory and discretionary integrity controls can protect data from malicious tampering and destruction as well as from accidental modification and destruction through operator errors (e.g., an operator may inadvertently attempt to delete the wrong relation) or faulty software.

Consistency is subdivided into three categories: (1) *database integrity rules*, which define correct states of a database in terms of relationships among the data, (2) *recovery management*, which returns the database to a consistent state after failure, and (3) *concurrency controls*, which ensure that concurrent transactions do not interfere, thereby creating inconsistent states of the database.

We shall discuss each aspect of integrity in more depth after first discussing assurance for these different aspects.

### Assurance

The notion of a security perimeter is essential to obtaining assurance that a security policy is actually enforced by the Trusted Computing Base (TCB) of a system. As stated in the Criteria "the bounds of the TCB equate to the 'security perimeter' " and "includes all those portions ... essential to the support of the policy." That is, the security perimeter is with respect to the security policy being enforced. Thus, the two categories of policy, viz., mandatory and discretionary, may well have two distinct security perimeters. This, of course, only applies to systems of Class B1 or above, because Class C systems do not support a mandatory policy.

The mandatory policy, for both secrecy and integrity, can be enforced with a very high degree of assurance against concerted attacks, including Trojan horses. As the evaluation classes move from B1 to B2, B3, and finally A1, the primary distinctions relate to the use of improved architecture, specification, verification, and testing to increase the assurance in the mandatory access controls enforced by the TCB. It is expected that the higher evaluation classes will be used to protect against users with a wider range of authorizations.

In contrast, because of their richer policies, discretionary access controls have inherent limitations (known as the 'safety problem'[3]) and more complex mechanisms than mandatory controls. This is especially true for database systems that protect data at the granularity of individual elements and have powerful access mechanisms, such as views, which rely on much of the database system for their support. Because of the inherent as well as technological limitations, little meaningful assurance of discretionary controls can be obtained beyond that of Class C2; in particular, one cannot obtain high assurance against Trojan horses. Fortunately, this matches well the real-world need for discretionary controls for need-to-know and corresponding integrity enforcement. Moreover, because discretionary controls operate within the confines of mandatory controls, the damage that can result from their failure is limited.

Because of the sharp distinction in the possible assurance for mandatory versus discretionary controls in a database system, the following discussion presumes that there may be two distinct security perimeters for systems at Class B2 and above: an inner perimeter (the 'reference monitor') for mandatory controls, and an outer perimeter (or perimeters) for discretionary and consistency controls. The maximum assurance that seems required, and the maximum practical, for the portion of the TCB outside the mandatory perimeter appears to be that prescribed for Class C2.

As discussed later, the assurance requirements for Class B2 and above, in particular the need to control covert channels, affects the meaning of consistency and the functionality of other aspects of a database system. However, having separate security perimeters makes it possible to more meaningfully address these problems.

## AUTHORIZATION INTEGRITY

### Mandatory Integrity Authorization

Mandatory security policies are particularly important because they describe global and persistent properties that are required for authorizations in a secure system. As defined in the Criteria[1], mandatory policies employ a reliable label to reflect the degree of protection required for information and to reflect the authorization of a subject to access information. When considering integrity, these labels reflect what the Criteria refers to as the 'sensitivity designation of the information,' or what is commonly termed the *integrity access class*, or simply *integrity class*, of the information objects. There is a comparable label that reflects an individual's 'authorization for the information;' this label is assigned to corresponding subjects. The primary systems of interest are those that can be represented by a Formal Security Policy Model, as defined in the Criteria. For such a system it is shown that if the initial state of the system is secure with respect to the policy, then all future states of the system will be secure.

For mandatory secrecy policies, the secrecy access classes must form a lattice. This requirement may be appropriate for mandatory integrity policies as well, although nonlattice mandatory integrity policies have been proposed[4].

For lattice-based policies, the integrity classes could correspond to integrity levels (analogous to secrecy levels such as SECRET), category sets of disjoint integrity compartments (analogous to secrecy compartments such as CRYPTO), or both.

Six mandatory security policies have been variously proposed to deal with integrity. In the context of the above concept of mandatory policy, each of these is examined as a possible integrity policy for databases:

1. Strict integrity
2. Low-water mark
3. Ring policy
4. Multilevel security with no write-up
5. Program integrity
6. Domains and types

The first three policies were introduced by Biba[5] as possible policies for multilevel-secure systems.

**Strict Integrity Policy**. This policy is an exact dual of multilevel secrecy as defined in the Bell and LaPadula model[6]. Each subject and object is assigned a fixed integrity class taken from the lattice of integrity classes, and strict integrity is preserved by prohibiting a subject from reading down or writing up in integrity.

There are two distinct considerations in assigning integrity classes to objects and subjects. First, the integrity class of the object to be protected from unauthorized modification must reflect the sensitivity of the information, viz., the potential damage that could result. Second, the integrity class of the subject must reflect its trustworthiness for making modifications. However, it is essential to note that the modifications by a subject are effected by the programs it executes and the data that control the execution of these programs. Thus, if a high integrity class is assigned to objects (files or segments) containing programs and program data, this assignment must reflect a determination that the resulting execution will produce only acceptable modifications.

The strict integrity model was initially introduced to deal with the threat of deliberate falsification or contamination of very sensitive information. One such application in which high integrity is of great importance is the preparation of targeting data that are used to control ballistic missiles. The practical threat is not so much that an unauthorized individual will be allowed to use such a system, but rather that a program and/or data maliciously prepared will be incorporated into a Trojan horse to retarget the weapons towards inconsequential or even friendly targets. This kind of Trojan horse could be implanted in what has become popularly known as a 'virus,' and strict integrity has been recognized as one of the few effective defenses.

There is a growing body of experience with the implementation and use of strict integrity in highly trusted operating systems. For example, in the Honeywell SCOMP, the first Class A1 system on the Evaluated Products List, strict integrity is included as part of the protection for segments. This mechanism is used for the protection for

security related information such as audit data. In addition, the Gemini GEMSOS[7] has incorporated strict integrity as part of the sensitivity label for all subjects, objects, and devices; this approach has been found useful when designing the integrity protection both of sensitive application information and of system information used to support the security controls themselves. Although there has been little comparable experience in database systems, the I.P. Sharp multilevel database model[8] incorporates strict integrity along with multilevel secrecy.

**Low-Water Mark Policy**. This policy is analogous to the high-water mark security policy of the ADEPT-50 system[9]. A subject's integrity class is dynamic and decreases as the subject reads data of lower integrity. If the integrity classes of objects are static (as in the strict integrity policy), a subject will be unable to write into an object with a higher integrity class than it has read; if the object classes are dynamic, then their integrity classes are possibly lowered if the subject writes into the object. As summarized by Biba[5], "This policy, in practice, has rather disagreeable behavior. . . . In a sense, a subject can sabotage (inadvertently) its own processing by making objects necessary for its function inaccessible (for modification). The problem is serious since there is no recovery short of reinitializing the subject." To the best of our knowledge, this policy has not been included in any system design.

**Ring Policy**. By prohibiting read-downs in integrity class, it seems the strict integrity policy and the low-water mark policy could prove to be quite restrictive for most systems, especially database systems. Because database processes must have both read and write access to user data, system tables, index files, logs, and other structures to answer queries and update the database, it would appear that the only workable assignment of integrity classes is system low. Because of the restrictiveness of the two preceding policies, Biba also introduced a more flexible policy called the ring policy. Each subject and object has a fixed integrity class, and a subject is only allowed to write into objects whose integrity classes are dominated by the subject's class. No restrictions are placed on reading, so a subject can write high integrity data even if it has read data of a lower integrity. Unfortunately, the relaxation of this policy makes the integrity class of the subject essentially meaningless, because there are no restrictions on even what programs the subject can execute. Thus, what would appear to be a high integrity subject can, without restriction, be executing erroneous or malicious programs that destroy the high integrity information to which the subject has access. In reality, this policy fails to meet the requirements for a mandatory policy. Moreover, there is no real experience using this policy as a basis for mandatory integrity.

**Multilevel Security with No Write-Up**. Extending the Bell and LaPadula model to prohibit 'writing-up' in secrecy class provides a limited form of mandatory integrity. In particular, this extended policy model addresses the 'write-up' problem of the mandatory secrecy policy, which allows a subject to write up in secrecy class. The extended model would prevent a SECRET subject, for example, from inserting data labeled as TOP-SECRET into a multilevel relation or from overwriting a TOP-SECRET element (which it cannot observe). This approach appears to protect subjects from lower-level subjects. Closer examination makes it clear that this approach is a case of the ring policy just addressed in which the secrecy labels, such as SECRET, are also used as the integrity labels; the difference is thus only syntactic with no difference in the results of the policy. Of course, this policy also has the same weaknesses as the ring policy.

**Program Integrity Policy**. The restrictions of the strict integrity policy remain a concern, so it seems important to try to identify a more flexible but useful policy. The real world supports some notion of integrity class through job levels and chain of command. However, the flows between different levels (usually adjacent) are bidirectional, so information flows both up and down in integrity class. Moreover, the trust placed on the information provided by any individual is often more a function of the individual than position. The key to the effective protection in this context is that the individuals are trusted to make only the desired modifications of high integrity information, even though they have been exposed to information of lower integrity classes.

This same concept can be applied to software by imposing more stringent requirements on assigning an object containing executable code a high integrity class. It seems unreasonable to assume that once a program has observed data of low integrity that it is incapable of writing data of higher integrity, or because data are entered by a user of low integrity into a database, that indexes and other structures on the database must be treated of low integrity also -- there is little relationship between the quality of the data that go into a database and the quality of the system structures that represent it.

This problem has been approached by distinguishing read access from execute access (which are treated identically in the preceding policies). Based on this distinction, Shirley and Schell[10] have defined a program integrity policy in which a subject is only allowed to write into objects of less than or equal integrity class and only allowed to execute objects of greater than or equal integrity. As with the ring policy, there are no restrictions on reading. This policy appears to be better suited for databases because the database processes could operate with a high integrity class, where they would be able to read and update the entire database. Users and application processes would be assigned integrity classes reflecting their 'trustworthiness'. Furthermore, Shirley has shown not only that this is a mandatory policy but also that it is the identical policy implemented by the hardware protection ring mechanism of Multics and several other systems (no connection with Biba's use of the term 'ring'). Thus there is a substantial body of experience with this policy, and it has indeed been shown to be quite useful in operating systems. There is no comparable body of direct experience with database systems.

An even closer look at the program integrity policy reveals the somewhat unexpected result that it is just a special case of the strict integrity policy. To understand this, it should be recalled that in the Bell and LaPadula model there is the notion of a 'trusted subject.' When interpreted for integrity, as in the case of the strict integrity policy, a trusted subject is trusted exactly to be able to read low

integrity information without damaging the integrity of high integrity data. This notion of trusted subject is too coarse for the problem at hand because a trusted subject can read any integrity class. However, the notion has been refined in the Gemini GEMSOS[7] to identify a 'multilevel subject' that has both a minimum and maximum class. Now, if the subject in each protection ring is regarded as multilevel (with respect to integrity classes) with a maximum integrity equal to the ring of execution and a minimum integrity equal to the least trusted ring, the strict integrity policy in this case becomes the program integrity policy if the multilevel subject is trusted not to execute any program with a lower integrity class than its maximum.

**Domains and Types.** Domains and types have been proposed as a means to specify a mandatory integrity policy, as illustrated by the Honeywell SAT system[4]. Here, each object is typed, and each domain has a list of types that it can observe and modify plus a list of domains that it can call. Although this policy model is similar to discretionary policies based on the access matrix model, the set of types, domains, and rights cannot be altered. Because it is a relatively new approach, its properties are not yet completely clear. So far, there is no experience applying this type of policy to a database system, although Honeywell is working on it.

### Discretionary Integrity Authorization

Discretionary integrity authorization policies control access to data at the user or user group level. The usual approach to controlling access in database systems includes *authorization lists*, which specify what operations a user (or group) is authorized to perform on some set of data. For integrity, the operations of interest include update, insert, and delete.

The authorization lists of database systems are included in the data model at different layers of abstraction. At the lowest layer, they are associated with files, records, or elements. At the highest layer, they are associated with *views* or *subschema* on the data. The high-level approach has the advantage of specifying a context for access. The context -- i.e., exact set of elements that fall within the target of a view -- is dynamic, changing as the underlying database is updated. Because it is easier and more natural for users, the high-level approach has proven to be far more useful than the low-level approach, and is embodied in many systems including SQL/DS, DB2, ORACLE, and INGRES (though in a somewhat different form).

The discretionary security policy contained in the Trusted Computer System Evaluation Criteria[1] is appropriate for database systems as long as the concept of object is interpreted to mean views (actually view specifications or subschema) rather than just physical elements, records, or files. Note that this does not mean that discretionary controls cannot be associated with individual records and elements; such controls are easily defined as views on the database.

The Criteria specify that discretionary controls are to be applied to 'each named object.' There is no requirement that the named objects be disjoint in memory, and in some operating systems a file may be accessed via different path names through different directories with different discretionary authorizations placed on the different names. Similarly, applying discretionary controls to views is consistent with the Criteria because views are just a way of naming objects. Also, there is no requirement that the 'named objects' of the discretionary policy be the same objects or even at the same layer of abstraction as the 'storage objects' of the mandatory policy.

## CONSISTENCY INTEGRITY

### Database Integrity Rules

Database integrity rules protect a database from data entry errors as well as from other errors made by the operator or by software. They define the correct states of the database and may specify actions to take if an update would cause the database to enter an incorrect state. They are similar to exception conditions built into programs, except that the conditions are represented in the database (as metadata) rather than in the application programs so that they can be automatically applied to all transactions updating the database.

In a relational system, there are two common types of database integrity rules: domain integrity rules and relational integrity rules. *Domain integrity rules* are context-free rules specifying the allowable set of values (i.e., domain) for an attribute, e.g., DRIVER.AGE is greater than 16 but less than 100. *Relational integrity rules* are context-sensitive rules specifying more global constraints on individual tuples or sets of related tuples, e.g., that every tuple in a PROGRAMMER relation has a corresponding tuple in an EMPLOYEE relation (this is a form of 'referential integrity'). Many relational systems, e.g., INGRES, provide mechanisms whereby users can define rather complex integrity rules.

Integrity rules play a vital part in ensuring the integrity of a database. Indeed, they are a very important part of access controls because most systems are vulnerable to errors as well as to sabotage. It is probably fair to say that a database system would not be regarded as a useful trusted system if it does not support integrity rules.

There are, however, intrinsic problems associated with integrity rules in a multilevel system that is rated at the evaluation level of B2 or higher, arising from the requirement to protect against covert channels. Because the implementation of integrity rules is outside the mandatory security perimeter, the database subjects that enforce the integrity rules must be denied access to data that is classified higher than the subject level. Thus, if the subjects are processing a transaction on behalf of a user, the only data visible to those subjects will be data that is classified at a level dominated by the user's level. If the database system were given access to data not dominated by the user's level, then a Trojan Horse in the database system could leak the unauthorized data -- that is, unless the database system (or a large portion thereof) were part of the mandatory security perimeter. Because the latter is neither feasible nor desirable, in multilevel systems rated at the level of B2 or higher, we are forced to consider integrity constraints as constraints on the subset of the database dominated by the user's clearance.

To see how this revised interpretation of integrity constraints affects the enforcement of integrity rules, consider the relational model, which requires each tuple in a relation to have a unique primary key. Suppose the tuples in a multilevel relation are classified SECRET or TOP-SECRET, and suppose the relation contains a TOP-SECRET tuple with primary key FOO. This tuple will be invisible to subjects operating on behalf of SECRET users. Thus, if a SECRET user attempts to insert a new tuple, also with key FOO, the system will accept the tuple. Because the access class becomes the only means of distinguishing the tuples, the class must then be considered to be part of the primary key. We refer to the coexistence of multiple tuples with the same primary key except for access class as *polyinstantiated* tuples[11].

Problems also arise with respect to referential integrity. For example, suppose a TOP-SECRET user creates a TOP-SECRET tuple in a relation T(ID, A), which is associated with a SECRET tuple in a relation S(ID, B) through the join attribute ID. The relation S represents the entities named by the primary key ID. If a SECRET user deletes the referenced tuple in S, referential integrity will be violated. But because the SECRET user, as well as all subjects that run on that user's behalf, cannot know of the existence of the TOP-SECRET tuple, this cannot be avoided.

As a third example of the problems that arise from invisible data, consider a relation that contains the weights of items on board various flights. Suppose there is maximum weight restriction of 5000 for any given flight and that some of the items on board a flight are classified SECRET while others are TOP-SECRET. If the integrity constraint is specified simply as an upper bound of 5000 for the total of all weights for a flight, a flight could be overloaded because the TOP-SECRET weights would be invisible when the constraint is applied at the SECRET level to determine whether an additional SECRET item can be placed on board. A possible solution is to have separate constraints for SECRET and TOP-SECRET weights.

Thus, in B2 or higher systems, the consistency defined by integrity constraints must be interpreted with respect to the secrecy class of the subject applying the constraint. However, whether there should be some notion of inter-level consistency, or how this might be specified, is unclear. It is also unclear how triggers fit into this notion since a trigger activated by an operation on behalf of a user having one secrecy class cannot read up or write down in secrecy class. Finally, we note that if the database is polyinstantiated at the tuple or element level, problems arise in applying the integrity constraints because more than one tuple or element with different values may be selected by the constraint, each with different outcomes. Thus, the integrity rules must specify which values to select among polyinstantiated values.

In a multilevel system, the concept of integrity constraints should also be extended to include constraints on the classifications assigned to data. For relational systems, we have found that several properties should hold:

- The complete definition (schema) for a relation, including the names of all attributes, should have a single access class that is dominated by the access classes of all data that is to go into the relation. Integrity rules that constrain the data going into the relation should also be assigned this access class.

- The attributes representing the primary key in a relation should be uniformly classified -- that is, within any given tuple, the elements forming the primary key should have the same access class.

- The classification of the primary key should be dominated by the classifications of all other elements within a tuple.

In that integrity rules enforce constraints on the relationships among data in the database, they can be associated with inference problems. For example, if an integrity constraint states that C = A + B for attributes A, B, and C, where A and B are SECRET but C is TOP-SECRET, then a SECRET user with access to A, B, and the integrity constraint can infer C. In this particular case, the best strategy for dealing with the problem may be to use the integrity constraint to force classifications on the data to prevent the inference -- e.g., classify A or B, or both, as TOP-SECRET. In cases where the rule of inference is complex and unknown, it may be more appropriate to classify the integrity constraint (which can be viewed as an inference rule).

In summary, although a multilevel secure database system should provide database integrity rules, the mandatory secrecy policy affects the interpretation and application of integrity constraints.

**Recovery Management**

Another vital aspect of database integrity is protecting the database from operator or software errors, including system crashes. The accepted method of dealing with such errors and faults is based on the concept of a *transaction*, which is a sequence of operations that behaves atomically -- that is, it either successfully completes (*commits*) all updates or else it has no effect on the state of the database (*rolls back*). The overall integrity policy for trusted systems should include the concept of transactions with commit and roll-back.

Multilevel updates raise some difficult issues regarding transaction management. For example, if a trusted user can simultaneously insert or update multilevel data (within the user's range of trust), it may be desirable to decompose these updates into single-level updates represented as single-level transactions and performed by single-level database subjects. However, the unit itself must also be treated as a transaction, so the concept of a multilevel transaction with single-level nested transactions appears to be very useful. The problem is rolling back the low portions of the transaction if the high portions fail.

Assuming recovery management is outside of the mandatory security perimeter, it is not clear how the database recovery log should be managed and processed in systems that are rated at the level of B2 or higher. However, some of the techniques used for general-purpose operating systems to ensure the consistency of file systems during

backup and recovery may be useful.

### Concurrency Controls

An important aspect of database integrity is ensuring that concurrent transactions do not interfere with each other, giving rise to inconsistent states of the database. *Serializability*, which states that any transaction schedule must be equivalent to one in which the transactions execute serially, has been shown to be a necessary and sufficient condition for global consistency[12], although there are systems that enforce somewhat weaker policies. Some notion of global consistency, however, is an essential aspect of the overall integrity policy for trusted database management systems. The concurrency policy should also address the problems of *deadlock*, where multiple transactions cannot proceed because they are waiting on each other, and *livelock*, where a transaction never exits from a wait state, both of which create denial-of-service problems.

In B2 or higher systems, the concurrency mechanisms must use techniques other than simple locks because read-write locks on multilevel data provide a signalling channel. Event counters[13] are not vulnerable to covert channels, but require that higher-level transactions roll back when a lower-level one causes an update that could interfere with its behavior.

## CONCLUSIONS

We do not know enough about the application of mandatory integrity policies to databases to recommend any one in particular or even state that one be mandated at all. While the strict integrity policy without trusted subjects may be appropriate for some threat environments, the more flexible program integrity policy, which uses restricted trusted subjects to manage a database, may be appropriate for most environments. It would be premature to adapt a particular mandatory policy in criteria for trusted database systems until such a policy has been experimentally tried in at least one operational environment and has been demonstrably successful. On the other hand, a discretionary policy along the lines of that given in the criteria is extremely useful provided it is interpreted to apply to views rather than just elements, records, or files.

Database integrity rules should be included in an overall integrity policy because they provide users with considerable assurance that the data is protected against many errors. This is one of the best ways in which the users themselves can greatly enhance the integrity of their data. However, the interpretation and application of integrity rules is constrained by the requirements for mandatory security. Similarly, any trusted system should support the concepts of atomic transactions, recovery, and noninterference, though again the features are constrained by the mandatory security requirements.

Although we believe it is vital for trusted systems to support these different integrity policies, it is neither necessary nor possible to have the same degree of assurance in the enforcement of them all. Whereas Classes A and B are appropriate for mandatory access controls, Class C2 is appropriate for discretionary controls and consistency controls, which are considerably more complex than mandatory controls and require much of the database system for their support.

To provide a high degree of assurance, the mandatory integrity policy must be enforced by the reference monitor. In addition to enforcing the mandatory secrecy policy, the reference monitor ensures the integrity of all data in the system, including the labels that represent the secrecy and integrity access classes. If the data are vulnerable to tampering during storage or transmission to and from the reference monitor, cryptographic checksums may be used to ensure the integrity of the data and its labels. For cryptographic checksums to be meaningful, it is essential that the processes that compute and validate the checksums and manage the key be under the strict control of a reference monitor.

## ACKNOWLEDGMENTS

An earlier version of this paper was prepared for the National Computer Security Center's Invitational Workshop on Database Security, where both authors participated in a working group on integrity and inference. The current version has benefited greatly from the group discussions, and we would like to thank the other group members, namely A. Arsenault, W. E. Boebert, D. Bonyun, D. Downs, K. Jacobs, R. Miller, G. Raudnbaugh, J. Spain, and S. Walker. We also thank T. Lunt, M. Heckman, and P. Neumann for their comments on this paper. This research was supported by the U.S. Air Force, RADC under contract F30602-85-C-0243.

## REFERENCES

1. Dept. of Defense, Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, 1983, CSC-STD-001-83

2. Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, Vol. II, 1983.

3. Harrison, M. A., Ruzzo, W. L. and Ullman, J. D., "Protection in Operating Systems", *Comm. ACM*, Vol. 19, No. 8, Aug. 1976, pp. 461-471.

4. Boebert, W. E. and Kain, R. Y., "A Practical Alternative to Hierarchical Integrity Policies", *Proc. of the 8th DOD/NBS Computer Security Conf.*, 1985, pp. 18-27.

5. Biba, K. J., "Integrity Considerations for Secure Computer Systems", Tech. report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Mass., April 1977.

6. Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations and Model", Tech. report M74-244, The MITRE Corp., Bedford, Mass., May 1973.

7. Schell, R. R., Tao, T. F., and Heckman, M., "Designing the GEMSOS Security Kernel for Security and Performance", *Proc. 8th Dod/NBS Computer Security Conf.*, 1985, pp. 108-119.

8. Grohn, M. J., "A Model of a Protected Data Management System", Tech. report ESD-TR-76-289, I. P. Sharp Assoc. Ltd., June 1976.

9. Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System", *Proc. Fall Jt. Computer Conf.*, Vol. 351969, pp. 119-133.

10. Shirley, L. J. and Schell, R. R., "Mechanism Sufficiency Validation by Assignment", *Proc. of the 1981 Symp. on Security and Privacy*, Apr. 1981, pp. 26-32.

11. Lunt, T. F., Denning, D. E., Schell, R. R., Heckman, M., "Polyinstantiation in a Secure Relational Database System", Tech. report, SRI International, May 1986.

12. Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M., "Consistency and Serializability in Concurrent Database Systems", *SIAM J. Comp.*, Vol. 13, No. 3, Aug. 1984, pp. 508-530.

13. Reed, D. P. and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers", *Comm. ACM*, Vol. 22, No. 2, Feb. 1979, pp. 115-123.

Ninth National Computer Security Conference, Gaithersburg, MD, Sept. 15- 18, 1986.