Abstract

Our objective in this paper is to answer the following question: what mechanisms are required in a general-purpose multiuser database management system (DBMS) to facilitate the integrity objectives of information systems? In a nutshell our conclusion is that realistic mechanisms do exist. Although existing commercial products fall far short of providing the requisite mechanisms, they can be easily extended to incorporate these mechanisms. Our principal contribution is to identify these mechanisms and point out where gaps still remain. We have also bridged the terminology and concepts of database and security specialists in a coherent manner. In the more detailed considerations the focus of this paper is on relational DBMS's.

Keywords: Integrity, Principles, Mechanisms, Database Management Systems, Security

Integrity Principles and Mechanisms in Database Management Systems *

Ravi Sandhu

Sushil Jajodia

Center for Secure Information Systems
and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030, USA

March 12, 1991

* *Computers & Security*, Volume 10, Number 5, August 1991, pages 413-427.

1 INTRODUCTION

Information integrity means different things to different people, and will probably continue to do so for some time. In spite of considerable effort recent attempts to establish a consensus definition have been unsuccessful [19]. So the first order of business is to define integrity. Our approach to this question is pragmatic and utilitarian. The objective is to settle on a definition within which we can achieve practically useful results, rather than searching for some absolute and philosophically airtight formulation.

1.1 A Definition of Integrity

We define integrity as being concerned with the *improper modification* of information (much as confidentiality is concerned with improper disclosure). We understand modification to include insertion of new information, deletion of existing information as well as changes to existing information. This definition of integrity is considerably broader than the traditional use of this term in the database literature. For instance Date [5] says: "Security refers to the protection of data against unauthorized disclosure, alteration, or destruction; integrity refers to the accuracy or validity of data." The consensus view among security researchers is that integrity is one component of security and accuracy/validity is one component of integrity [8, 19, for instance].

The reader has probably seen similar definitions using "unauthorized" instead of "improper." Our use of the latter term is significant and should not be dismissed lightly. We particularly wish to emphasize two points. Firstly, integrity breaches can and do occur without authorization violations. In other words authorization is only

one piece of the solution and we must also deal with the malicious user who exercises his authorization improperly. Secondly, our definition raises the key question: what do we mean by improper? It is obvious that this question intrinsically cannot have an universal answer, so it is futile to try to answer it outside of a given context. We are specifically interested in information systems used to control and account for an organization's assets and resources. In such systems the primary security goal is prevention of fraud and errors.

1.2 The Insider Threat

It is important to understand that the threat posed by a corrupt authorized user is quite different in the context of integrity as compared to secrecy.

A corrupt user can leak secrets by (i) using the computer to legitimately access confidential information, and then (ii) passing on this information to an improper destination by some non-computer means of communication (e.g., a telephone call). It is simply impossible for the computer to know whether or not step (i) was followed by step (ii). We therefore have no choice but to trust our insiders to be honest and alert. The military and government sectors have established elaborate procedures for this purpose, while the commercial sector is much more informal in this respect. Security research which focuses on secrecy therefore considers the principal threat to be Trojan Horses embedded in programs, i.e., corrupt programs, rather than corrupt users (see [9] for example).

Analogously, a corrupt user can compromise integrity by (i) manipulating stored data or (ii) falsifying source or output documents. A computer system can do little by itself to solve the problem of false source or output documents, for which we

must rely on the traditional techniques of paper-based manual systems. However the manipulation of stored data simply cannot be done without use of the computer. Therefore, in principle, the computer system is in a position to detect or prevent such manipulation. Integrity researchers must therefore focus on the corrupt user as the principal problem. In fact the Trojan Horse problem can itself be viewed as a problem of corrupt system or application programmers who improperly modify the software under their control. Also note that the problem of the corrupt user remains even if we are willing to trust all our software to be free of Trojan Horses.

1.3 Integrity Principles and Mechanisms

Our objective in this paper is to answer the following question: what mechanisms are required in a general-purpose multiuser DBMS to help achieve the integrity objectives of information systems? There are many compelling reasons to focus on DBMSs for this purpose. This is particularly true when we focus on mechanisms. DBMSs provide significant data semantics beyond the usual Operating System (OS) notion of file being an interpreted sequence of bytes. DBMSs also have the wonderful ability to express and manipulate complex relationships. This comes in very handy when dealing with sophisticated integrity policies.

The Operating System must clearly provide some core integrity and security mechanisms. At the very least one needs a mechanism to enforce encapsulation of a database, i.e., to ensure that all manipulation of the database can only be through the DBMS. The question of what minimal features are required in the OS is an important and non-trivial one, but is outside the scope of the present paper. For now let us assume that OS's with the requisite features are available and ask ourselves

what features can the DBMS give us?

The bulk of integrity mechanisms properly belong in the DBMS. Integrity policies are intrinsically application specific and the OS simply cannot provide the means to state application-specific concerns. One might then argue: why not put all the mechanism in the application code? There are several persuasive reasons not to do this. Firstly, any assurance that mechanisms interspersed within application code will be correct or even comprehensible is rather dubious. Secondly, the whole point of a database is to support multiple applications. A particular application may well be in a position to handle all its integrity requirements. Yet it is only the DBMS which can prevent other applications from corrupting the database. Thirdly, putting mechanisms in application code is not conducive to reuse of common mechanism among different applications.

The rest of the paper is organized as follows. In section 2 we discuss principles for achieving integrity in information systems. In section 3 we describe the mechanisms required in a DBMS to support these high level principles. In some of the more detailed consideration we will limit ourselves specifically to relational DBMS's. As we will see traditional DBMS mechanisms provide the foundations for this purpose, but by themselves do not go far enough. Section 4 concludes the paper.

2 INTEGRITY PRINCIPLES

We begin by describing basic principles for achieving information integrity. These principles can be viewed as high level objectives which are made more concrete when specific mechanisms are proposed to support them. In other words these principles

lay down broad goals without specifying how to achieve them. We will subsequently map these principles to DBMS mechanisms. Principles lay out *what* needs to be done while mechanisms establish *how* these principles are to be achieved.

We emphasize that the integrity principles themselves are independent of the DBMS context. They apply equally well to any information system be it a manual paper-based system, a centralized batch system, an interactive and highly distributed system, etc. Our objective in this paper is to interpret these principles in the DBMS context and identify DBMS mechanisms to support them. We also point out that many, if not all, of these principles are equally applicable to secrecy as well as integrity. Our focus in this paper is on integrity. Analysis of the relevance and significance of these principles to secrecy objectives is outside the scope of this paper.

The nine integrity principles enumerated below are abstracted from a variety of sources. The more recent literature includes the Clark and Wilson papers [1, 2, 3] and the NIST workshops [18, 19]. The "older" literature is too numerous to cite individually. For those unfamiliar with this literature some useful starting points are [6, 8, 10, 13, 20]. The reader has probably seen similar lists in the past. We believe the time is right for a revised formulation of major principles, particularly in view of the recent resurgence of interest in integrity. We emphasize that these principles express *what* needs to be done rather than *how* it is going to be accomplished. The latter question is addressed in the next section.

1. *Well-formed Transactions.* Clark and Wilson [1] have defined this principle as follows: "The concept of the well-formed transaction is that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or

ensure the integrity of the data." This principle has also been called *constrained change* [3], i.e., data can only be modified by well-formed transactions rather than by arbitrary procedures. Moreover the well-formed transactions are known ("certified") to be individually correct with some (mostly qualitative) degree of assurance.

2. *Authenticated Users.* This principle stipulates that modifications should only be carried out by users whose identity has been authenticated to be appropriate for the task.

3. *Least Privilege.* The notion of least privilege was one of the earliest principles to emerge in security research. It has classically been stated in terms of processes (executing programs) [20], i.e., a process should have exactly those privileges needed to accomplish its assigned task, and none extra. The principle applies equally well to users, except that it is more difficult to precisely delimit the scope of a user's "task." A process is typically created to accomplish some very specific task and terminates on completion. A user on the other hand is a relatively long-lived entity and will be involved in varied activities during his lifespan. His authorized privileges will therefore exceed those strictly required at any given instant. In the realm of confidentiality least privilege is often called *need-to-know*. In the integrity context it is appropriately called *need-to-do*. Another appropriate term for this principle is *least temptation*, i.e., do not tempt people to commit fraud by giving them greater power than they need.

4. *Separation of Duties.* Separation of duties is a time honored principle for prevention of fraud and errors, going back to the very beginning of commerce.

Simply stated, no single individual should be in a position to misappropriate assets on his own. Operationally this means that a chain of events which affects the balance of assets must require different individuals to be involved at key points, so that without their collusion the overall chain cannot take effect.

5. *Reconstruction of Events.* This principle seeks to deter improper behavior by threatening its discovery. The ability to reconstruct what happened in a system stems from the notion of accountability. Users are accountable for their actions to the extent that it is possible to determine what they did. Reconstruction of events is also a necessary adjunct to least privilege for two reasons. Firstly least privilege, even taken to its theoretical limit, will leave some scope for fraud. Secondly a zealous application of least privilege is not a terribly efficient way to run an organization. It conveys the image of an enterprise enmeshed in red tape.* So practically users must be granted more privileges than are strictly required. We therefore should be able to accurately reconstruct essential elements of a system's history, in order to detect misuse of privileges.

6. *Delegation of Authority.* This principle fills in a piece missing from the Clark and Wilson papers and much of the discussion they have generated.† It concerns the critical question of how privileges are acquired and distributed in an organization? Clearly the procedures to do so must reflect the structure of the

*This comment is made in the context of users rather than processes (transactions). Least privilege with respect to processes is more of an internal issue within the computer system, and its zealous application is most desirable (modulo the performance and cost penalties it imposes).

†The closest concept that Clark and Wilson have to this principle is their Rule E4 which they summarize as follows [1, figure 1]: "Authorization lists changed only by the security officer." This notion of a central security officer as an authorization czar is inappropriate and unworkable. Rational security policies can be put in place only if appropriate authority is vested in end-users.

organization and allow for effective devolution of authority. Individual managers should have maximum flexibility regarding information resources within their domain, tempered by constraints imposed by their superiors. Without this flexibility at the end-user level, the authorization will most likely be inappropriate to the actual needs. This can only result in security being perceived as a drag on productivity and something to be bypassed whenever possible.

7. *Reality Checks.* This principle has been well motivated by Clark and Wilson [3] as follows: "A cross-check with the external reality is a central part of integrity control. ... integrity is meaningful only in terms of the relation of the data to the external world." Or in more concrete terms: "If an internal inventory record does not correctly reflect the number of items in stock, it makes little difference if the value of the recorded inventory has been reflected correctly in the company balance sheet." By definition reality checks entail activity external to the computer system.

8. *Continuity of Operation.* This principle states that system operations should be maintained to some appropriate degree in the face of potentially devastating events which are beyond the organization's control. This catch-all description is intended to include natural disasters, power outages, disk crashes and the like. With this principle we are clearly stepping into the scope of availability. We have mentioned it here for the sake of completeness. One would be hard pressed to claim that a system which does not address this requirement can at the same time have a high measure of integrity.

9. *Ease of Safe Use.*[‡] In a nutshell this principle requires that the "easy" ways to operate a system should also be the safest ones. It is important to acknowledge this principle because of the ample evidence that security measures are all too often incorrectly applied or simply bypassed by system managers. This happens due to a combination of (i) poorly designed defaults (such as indefinite retention of vendor-supplied passwords for privileged accounts), (ii) awkward and cumbersome interfaces (such as requiring many keystrokes to effect simple changes in authorization), (iii) lack of tools for authorization review, or (iv) mismatched policy and mechanism ("...the extent that the user's mental image of his protection goals matches the mechanism he must use, mistakes will be minimized." [20]).

It is inevitable that these principles are fuzzy, abstract and high level. In developing an organization's security policy one would elaborate on each of these principles and make precise the meaning of terms such as "appropriate" and "proper." How to do so systematically is perhaps the most important question in successful application of these principles. In other words how does one articulate a comprehensive policy based on these high level objectives? This question is beyond the scope of this document. Our present focus is on the more technical question: how do these principles translate into concrete mechanisms in a DBMS?

The goals encompassed by these principles may appear overwhelming. After all in the extreme these principles amount to solving the total system correctness problem, which we know is well beyond the state of the art. Fortunately, in our context, the

[‡]Thanks to Stanley Kurzban and William Murray for coining this particular term.

degree to which one would seek to enforce these objectives and the assurance of this enforcement are matters of risk management and cost-benefit analysis. Laying out these principles explicitly does give us the following major benefits.

- The overall problem is partitioned into smaller components for which solutions can be developed independently of each other (i.e., divide and conquer).
- The principles suggest common mechanisms which belong in the DBMS and can be reused across multiple applications.
- The principles provide a set against which the mechanisms of specific DBMS's can be evaluated (in an informal sense).
- The principles similarly provide a set on the basis of which the requirements of specific information systems can be articulated.
- Last, but not the least, the principles invite criticism from the security community particularly regarding what may have been left out.

3 INTEGRITY MECHANISMS

In this section we consider DBMS mechanisms to facilitate application of the principles defined in the previous section. The principles have been applied in practise [16, 26, for instance] but with most of the mechanism built into application code. Providing these mechanisms in the DBMS is an essential prerequisite for their widespread use.

Our mapping of principles to mechanisms is summarized in table 1. Some of these mechanisms are available in commercial products. Others are well established in the

database literature. There are also some newer mechanisms which have been proposed more recently, e.g., transaction controls for separation of duties [22], the temporal model for audit data [12] and propagation constraints for dynamic authorization [21, 23]. Finally there are places where existing mechanisms and proposals need to be extended in novel ways. Overall the required mechanisms are quite practical and well within the reach of today's technology.

3.1 Well-formed Transactions

The concept of a well-formed transaction corresponds very well to the standard DBMS concept of a transaction [10, 11]. A transaction is defined as a sequence of primitive actions which satisfies the following properties.

1. *Failure atomicity*: either all or none of the updates of a transaction take effect. (We understand update to mean modification, i.e., it includes insertion of new data, deletion of existing data and changes to existing data.)

2. *Serializability*: the net effect of executing a set of transactions is equivalent to executing them in some sequential order, even though they may actually be executed concurrently (i.e., their actions are interleaved or simultaneous).

3. *Progress*: every transaction will eventually complete, i.e., there is no indefinite blocking due to deadlock and no indefinite restarts due to livelocks.

4. *Correct state transform*: each transaction if run by itself in isolation and given a consistent state to begin with will leave the database in a consistent state.

We will elaborate on these properties in a moment.

First let us note the basic requirement that the DBMS must ensure that updates are restricted to transactions. Clearly, if users are allowed to bypass transactions and directly manipulate relations in a database, we have no foundation to build upon. In other words updates should be encapsulated within transactions.[§] This restriction may seem too strong because in practice there will always be a need to perform ad hoc updates. However, ad hoc updates can themselves be carried out by means of special transactions! Of course the authorization for these special ad hoc transactions should be carefully controlled and their usage properly audited.

Secondly, it is clear that the set of database transactions is itself going to change during the system life cycle. Now the same nine principles of the previous section apply with respect to maintaining the integrity of the transactions. In particular transactions should be installed, modified and supplanted only by the use of well-formed "transaction-maintenance transactions." One can apply this argument once again to say that the transaction-maintenance transactions themselves need to be maintained by another set of transactions, and so on indefinitely. We believe there is little to be gained by having more than two steps in this potentially unbounded sequence of transaction-maintenance transactions. The rate of change in the transaction set will be significantly slower than the rate of change in the data base proper. Going one step further, the rate of change in the transaction-maintenance transactions will be yet slower to the point where, for all practical purposes, these can be viewed as static over the lifespan of typical systems. With this perspective the data base administrator is responsible for installing and maintaining transaction-maintenance

[§]At this point it is worth recalling that the database itself must be encapsulated within the DBMS by the Operating System.

transactions, which in turn control the maintenance of actual database transactions.

We now return to considering the four properties of DBMS transactions enumerated earlier. The first three properties—failure atomicity, serializability and progress—can be achieved in a purely "syntactic" manner, i.e., completely independent of the application. These three requirements for a transaction are recognized in the database literature as appropriate for the DBMS to implement. Mechanisms to achieve these objectives have been extensively researched in the last fifteen years or so, and our understanding of this area can certainly be described as mature. The basic mechanisms—two-phase locking, timestamps, multi-version databases, two-phase commit, undo-redo logs, shadow pages, deadlock detection and prevention—have been identified and should soon make their way into commercial products. In developing integrity guidelines and/or evaluation criteria one might consider some progressive measure of the extent to which a particular DBMS meets these objectives. For instance, with failure atomicity, is there a guarantee that we will know which of the two possibilities occurred? Similarly, with serializability, does the DBMS enforce the concurrency control protocol or does it rely on transactions to execute explicit commands for this purpose? And, with the issue of progress, do we have a probabilistic or absolute guarantee? Such questions must be systematically addressed.

The fourth property of correct state transforms is the ultimate bottleneck in realizing well-formed transactions. It is also an objective which cannot be achieved without considering the semantics of the application. The correctness issue is of course undecidable in general. In practice we can only assure correctness to some limited degree of confidence by a mix of software engineering techniques such as formal verification,

testing, quality assurance, etc. Responsibility for implementing transactions as correct state transforms has traditionally been assigned to the application programmer. Even in theory DBMS mechanisms can never fully take over this responsibility.

DBMS mechanisms can help in assuring the correctness of a state by enforcing *consistency constraints* on the data. Consistency constraints are also often called integrity constraints or integrity rules in the database literature. Since we are using integrity in a wider sense we prefer the former term.

The relational data model in particular imposes two consistency constraints [4, 5].

- *Entity integrity* stipulates that attributes in the primary key of a base relation cannot have null values. This amounts to requiring that each entity represented in the database must be uniquely identifiable.

- *Referential integrity* is concerned with references from one entity to another. A foreign key is a set of attributes in one relation whose values are required to match those of the primary key of some specific relation. Referential integrity requires that a foreign key either be all null or a matching tuple exist in the latter relation. This amounts to ruling out dangling references to non-existent entities.

Entity integrity is easily enforced. Referential integrity on the other hand requires more effort and has seen limited support in commercial products. The precise manner in which to achieve it is also very dependent on the semantics of the application. This is particularly so when the referenced tuple is deleted. There are several choices as follows: (i) prohibit this delete operation, (ii) delete the referencing tuple (with a

possibility of further cascading deletes), or (iii) set the foreign key attributes in the referencing tuple to null. There are proposals for extending SQL so that these choices can be specified for each foreign key.

The relational model in addition encourages the use of *domain constraints* whereby the values in a particular attribute (column) are constrained to come from some given set. These constraints are particularly easy to state and enforce, at least so long as the domains are defined in terms of primitive types such as integers, decimal numbers and character strings. A variety of *dependency constraints* which constrain the tuples in a given relation have been extensively studied in the database literature.

In the limit a consistency constraint can be viewed as an arbitrary predicate which all correct states of the database must satisfy. The predicate may involve any number of relations. Although this concept is theoretically appealing and flexible in its expressive power, in practice the overhead in checking the predicates for every transaction has been prohibitive. As a result relational DBMS's typically confine their enforcement of consistency constraints to domain constraints and entity integrity.

3.2 Continuity of Operation

The problem of maintaining continuity of operation in the face of natural disasters, hardware failures and other disruptive events has received considerable attention in both theory and practice [10]. The basic technique to deal with such situations is redundancy in various forms. Recovery mechanisms in DBMS's must also ensure that we arrive at a consistent state. In many respects these mechanisms are "syntactic" in the sense of being application independent, much as mechanisms for the first three properties of section 3.1 were.

3.3 Authenticated Users

Authentication is primarily the responsibility of the Operating System. If the Operating System is lacking in its authentication mechanism it would be very difficult to ensure the integrity of the DBMS itself. The integrity of the database would thereby be that much more suspect. It therefore makes sense to not duplicate authentication mechanisms in the DBMS.

Authentication underlies some of the other principles, particularly, least privilege, separation of duties, reconstruction of events and delegation of authority. In all of these the end objective can be achieved to the fullest extent only if authentication is possible at the level of individual users.

3.4 Least Privilege

The principle of least privilege translates into a requirement for fine grained access control. We have earlier noted that least privilege must be tempered with practicality in avoiding excessive red tape. Nevertheless a high-end DBMS should provide for access control at very fine granularity, leaving it to the database designers to apply these controls as they see fit.

It is clear from the Clark-Wilson papers, if not evident from earlier work, that modification of data must be controlled in terms of transactions rather than blanket permission to write. We have already put forth the concept of encapsulated updates for this purpose. In terms of the relational model it is not immediately obvious at what granularity of data this should be enforced.

For purpose of controlling read access DBMSs have employed mechanisms based

on views (as in System R) or query modification (as in INGRES). These mechanisms are extremely flexible and can be as fine grained as desired. However neither one of these mechanisms provides the same flexibility for flexible control of updates. The fundamental reason for this is our theoretical inability to translate updates on views unambiguously into updates of base relations. As a result authorization to control updates is often less sophisticated than authorization for read access.

In relational systems it is natural for obvious reasons to represent the access matrix by one or more relations [25]. At a coarse level we might control access by tuples of the following form

user, transaction, relation

meaning that the specified user can execute the specified transaction on the specified relation. Tuples of the form shown below would give greater selectivity

user, transaction, relation, attribute

This would allow us to control the execution of transactions such as, "give everyone a 5% raise," without giving the same transaction permission to change employee addresses. The following authorization tuple accomplishes this.

Joe, Give-5%-raise, Employees, Salary

A transaction which gives a raise to a specific employee needs a further dimension of authorization to specify which employee it pertains to. Thus, if Joe is authorized to give a 5% raise to John the authorization tuple would look as follows.

Joe, Give-5%-raise, John, Employees, Salary

We are assuming here that John uniquely identifies the employee receiving the raise. The update is restricted to the Salary attribute of a specific tuple with key equal to 'John' in the Employees relation. So it takes a key, relation and attribute to specify the actual parameter of such a transaction.

Now consider a transaction which moves money from account A to account B, i.e., there are two actual parameters of the transaction. In terms of least privilege we need the ability to bind this transaction to updating the two specific accounts A and B. More generally we will have transactions with N parameters identified in a actual parameter list. So we need authorization tuples of the following form,

user, transaction, actual parameter list

where each parameter in the actual parameter list specifies the item authorized for update by specifying one of the following identifiers

- relation,
- relation, attribute,
- key, relation, attribute.

These three cases respectively give us relation level, "column" level and element level granularity of update control.

It is also important to realize that element-level update authorizations should properly be treated as consumable items. For example, once money has been moved from account A to account B the user should not be able to move it again, without fresh authorization to do so.

3.5 Separation of Duties

Separation of duties finds little support in existing products. Although it is possible to use existing mechanisms for this purpose, these mechanisms have not been designed with this end in mind. As a result their use is awkward at best. This fact was noted by the DBMS group at the 1989 NIST data integrity workshop who concluded their report with the following recommendation [19, section 4.3].

> While the group was able to use existing DBMS features to implement separation of roles controls, we were, however, unable to use existing features in a way that would support easy maintenance and certification. We recommend that data definition and/or consistency check features be enhanced to provide operators that lend themselves to the expression of integrity controls and to allow separation of integrity controls and traditional data.

Separation of duties is inherently concerned with sequences of transactions, rather than individual transactions in isolation. For example consider a situation in which payment in the form of a check is prepared and issued by the following sequence of events.

1. A clerk prepares a voucher and assigns an account.
2. The voucher and account are approved by a supervisor.
3. The check is issued by a clerk who must be different from the clerk in step 1. Issuing the check also debits the assigned account. (Strictly speaking we should debit one account and credit another in equal amounts. The important point for our purpose is that issuing a check modifies account balances.)

This sequence embodies separation of duties since the three steps must be executed by different people. The policy moreover has a dynamic flavor in that a particular clerk can prepare vouchers as well as, on different occasions, issue checks. However he cannot issue a check for a voucher prepared by himself.

Implementation of this policy in a paper-based system follows quite directly from its statement.

- The voucher is realized as a form with blank entries for the amount and account, as well as for signatures of the people involved. As the above sequence gets executed these blanks are filled in. On its completion copies of the voucher are filed in various archives for audit purposes.

- The account is represented by say a ledger card, where debit and credit entries are posted along with references to the forms which authorized these entries.

By their very nature paper-based controls rely on employee vigilance and internal/external audits for their effectiveness. Computerization brings with it the scope for enforcing the required controls by means of an infallible, ever vigilant and omniscient automaton, viz., the computer itself.

The crucial question is how do we specify and implement similar controls for separation of duties in a computerized environment? A mechanism for this purpose is described in [22]. This mechanism of *transaction-control expressions* is based on the following difference between vouchers and accounts.

- The voucher is *transient* in that it comes into existence, has a relatively small sequence of steps applied to it and then disappears from the system (possibly

leaving a record in some archive). The history of a voucher can be prescribed as a finite sequence of steps with an a priori maximum length.

- The account on the other hand is *persistent* in the sense it has a long-lived, and essentially unbounded, existence in the system. During its life there may be a very large number of credit and debit entries for it. Of course, at some point the account may be closed and archived. The key point is that we can only prescribe its history as a variable-length sequence of steps with no a priori maximum length.

Both kinds of objects are essential to the logic and correct operation of an information system. Transient objects embody a logically complete history of transactions corresponding to a unit of service provided to the external world by the organization. Persistent objects embody the internal records required to keep the organization functioning with an accurate correspondence to its interactions with the external world.

Separation of duties is achieved by enforcing controls on transient objects, for the most part. The crucial idea, which makes this possible, is that transactions can be executed on persistent objects only as a side effect of executing transactions on transient objects. This thesis is actually simply borrowed from the paper-based world where it has been routinely applied ever since bookkeeping became an integral part of business operations.

With this perspective we arrive at the diagram shown in figure 1. The idea is that a sequence of transactions is viewed as transient data in the database. In this picture there is a double encapsulation of the database, first by transactions on persistent data and then by transactions on transient data. Users can directly only execute the

latter. The former are triggered indirectly as a result when the transient is in the proper state for doing so. In other words transient data is singly encapsulated and has direct application of separation of duties. Persistent data is doubly encapsulated and has indirect application of separation of duties by means of transient data.

3.6 Reconstruction of Events

The ability to reconstruct events in a system serves as a deterrent to improper behavior. In the DBMS context the mechanism to record the history of a system is traditionally called an "audit trail." As with the principle of least privilege, a high-end DBMS should be capable of reconstructing events to the finest detail. In practise this ability must be tempered with the reality that gathering audit data indiscriminately can generate overwhelming volume. Therefore a DBMS must also allow fine-grained selectivity regarding what is audited. It should also structure the audit trail logically so that it is easy to query. For instance, logging every keystroke does give us the ability to reconstruct the system history accurately. However with this primitive logical structure one needs substantial effort to reconstruct a particular transaction. In addition to the actual recording of all events that take place in the database, an audit trail must also provide support for true auditing, i.e., an audit trail must have the capability "for an authorized and competent agent to access and evaluate accountability information by a secure means, within a reasonable amount of time and without undue difficulty" [7]. In this respect DBMSs have a significant advantage, since their powerful querying abilities can be used.

The ability to reconstruct events has different meaning to different people. At one end of the spectrum, we have the requirements of Clark and Wilson [3]. They require

only two things:

1. A complete history of each and every modification made to the value of an item.

2. With each change in value of an item, store the identity of the person making the change.

Of course, the system must be reliable in that it makes exactly those changes that are requested by users and the binding of a value with its author is also exact. Clark and Wilson call this "attribution of change."

This can be easily accomplished if we are willing to extend slightly the standard logging techniques for recovery purposes. For each transaction, a recovery log contains the transaction identifier, some *before-images*, and the corresponding *after-images*. If we augment this by recording in addition the user for each transaction, we have the desired binding of each value to its author. There is one other change that needs to be made. In order to support recovery, there is a need to keep a log only up to a point from which a complete database backup is available. Of course, now there is a need to archive the logs so they remain available.

Others have argued that this simple "attribution of change" is not sufficient. We need an audit trail, a mechanism for a complete reconstruction of every action taken against the database: *who* has been accessing *what* data, *when*, and in what *order*. Thus, it has three basic objects of interest:

1. The user - who initiated a transaction, from what terminal, when, etc.

2. The transaction - what was the exact transaction that was initiated.

3. The data - what was the result of the transaction, what were the database states before and after the transaction initiation.

For this purpose a *database activity model* has been recently proposed [12] that imposes a uniform logical structure upon the past, present, and future data. There is never any loss of historical or current information in this model, thus the model provides a mechanism for complete reconstruction of every action taken on the database. It also logically structures the audit data to facilitate its querying.

3.7 Delegation of Authority

The need to delegate authority and responsibility within an organization is essential to its smooth functioning. It appears in its most developed form with respect to monetary budgets. However the concept applies equally well to the control of other assets and resources of the organization.

In most organizations the ability to grant authorization is never completely unconstrained. For example, a department manger may be able to delegate substantial authority over departmental resources to project managers within his department and yet be prohibited to delegate this authority to project managers outside the department. These situations cloud the classic distinction between discretionary and mandatory policies [17, 24]. The traditional concept of ownership as the basis for delegating authority also becomes less applicable in this context [14]. Finally we need the ability to delegate privileges without having the ability to exercise these privileges. Some mechanisms for this purpose have been recently proposed [14, 23].

The complexity introduced by dynamic authorization has been recognized ever

since researchers considered this problem, e.g., as stated in the following quote [20].

> "... it is relatively easy to envision (and design) systems that statically express a particular protection intent. But the need to change access authorizations dynamically ... introduces much complexity into protection systems."

This fact continues to be true in spite of substantial theoretical advances in the interim [21]. Existing products provide few facilities in this respect and their mechanisms tend to have an ad hoc flavor.

3.8 Reality Checks

This principle inherently requires activity outside of the DBMS. The DBMS does have obligation to provide an internally consistent view of that portion of the database which is being externally verified. This is particularly so if the external inspection is conducted on an ad hoc on-demand basis. The DBMS can also play a significant role in ensuring that information known to be only partially valid and complete is presented as such. That is the DBMS can qualify its answers based on the scope of its knowledge about deviations from the external reality. A mechanism for this purpose has been proposed in [15].

3.9 Ease of Safe Use

Ease of safe use is more an evaluation of the DBMS mechanisms than something to be enforced by the mechanisms themselves. The mechanisms should of course have fail-safe defaults [20], e.g., access is not available unless explicitly granted or this default rule is explicitly changed to grant it automatically. DBMS's do offer a

significant advantage in providing user friendly interfaces intrinsically for their main objective of data manipulation. These interface mechanisms can be leveraged to make the authorization mechanisms easy to use. For instance, having the power of SQL queries to review the current authorizations is a tangible benefit in this regard.

4 CONCLUSION

In a nutshell our conclusion is that realistic DBMS mechanisms do exist to support the integrity objective of information systems. Some are well established in the literature while others have been proposed more recently and are not so well known. Our principal contribution is to identify these mechanisms and to identify the gaps where none existed or had been fully articulated.

In terms of what DBMS mechanisms can do for us, we can group the nine principles enumerated in this paper as shown in table 2. Group I principles are adequately treated by current DBMS mechanisms and have been extensively studied by database researchers. With the single exception of assuring correctness of state transformations these principles can be achieved by DBMS mechanisms. Techniques for implementing well-formed transactions and maintaining continuity of operation across failures have been studied extensively. Their practical feasibility has been amply demonstrated in actual systems. Assuring that well-formed transactions are correct state transformations remains a formidable problem, but there is little that the DBMS can do to alleviate it. As such it is a problem outside the scope of DBMS mechanisms. The DBMS can (i) enforce encapsulation of updates by restricting their occurrence to be within transactions, and (ii) provide controls for installing and maintaining these

transactions.

Group II principles need newer mechanisms and conceptual foundations. Several promising approaches have emerged in the literature. Practical demonstration of their feasibility remains to be done, but in concept they do not present prohibitive implementation problems. They do require that current DBMS's be extended in significant ways. Group II principles are the ones where additional DBMS mechanisms hold the promise of greatest benefit.

Group III principles are important but there is little that DBMS mechanism can do to achieve them. Authentication is principally an operating system problem. Reality checks necessarily involve external procedures. Ease of safe use is more an evaluation of the DBMS mechanisms than something to be enforced by the mechanisms themselves. It is facilitated in the DBMS context due to the intrinsic DBMS requirement of user friendly query languages.

In conclusion for group I principles we need little more than has currently been demonstrated in actual products. For group II principles, current systems do something for each one but do not go far enough. There are several promising proposals but no "worked examples." Group III principles are important but are not fully achievable by DBMS mechanisms alone.

Acknowledgment

We are indebted to John Campbell, Sylvan Pinsky and Howard Stainer for their support and encouragement, making this work possible.

References

[1] Clark, D.D. and Wilson, D.R. "A Comparison of Commercial and Military Computer Security Policies." *Proc. IEEE Symposium on Security and Privacy*, pages 184-194 (1987).

[2] Clark, D.D. and Wilson, D.R. "Comments on the Integrity Model." In [18], section 9, pages 1-6 (1989).

[3] Clark, D.D. and Wilson, D.R. "Evolution of a Model for Computer Integrity." In [19], section A.2, pages 1-13 (1989).

[4] Codd, E.F. "Extending the Relational Database Model to Capture More Meaning." *ACM Transactions on Database Systems* 4(4):397-434 (1979).

[5] Date, C.J. *An Introduction to Database Systems.* Volume I, Addison-Wesley, fourth edition (1986).

[6] Denning, D.E. and Denning, P.J. "Data Security." *ACM Computing Surveys* 11(3):227-249 (1979).

[7] Department of Defense National Computer Security Center. *Department of Defense Trusted Computer Systems Evaluation Criteria.* DoD 5200.28-STD (1985).

[8] Fernandez, E.B., Summers, R.C. and Wood, C. *Database Security and Integrity.* Addison-Wesley (1981).

[9] Gasser, M. *Building a Secure Computer System.* Van Nostrand Reinhold (1988).

[10] Gray, J. "Notes on Data Base Operating Systems." In *Operating Systems—An Advanced Course*, Bayer, R. et al (editors), Springer-Verlag, pages 393-481 (1978).

[11] Gray, J. "Why Do Computers Stop and What Can Be Done About It?" *Proc. IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 3-12 (1986).

[12] Jajodia, S., Gadia, S.K., Bhargava, G. and Sibley, E. "Audit Trail Organization in Relational Databases." In *Database Security III: Status and Prospects*, Spooner, D.L. and Landwehr, C.E. (editors), North-Holland, pages 269-281 (1990).

[13] Linden, T.A. "Operating System Structures to Support Security and Reliable Software." *ACM Computing Surveys* 8(4):409-445 (1976).

[14] Moffett, J.D. and Sloman, M.S. "The Source of Authority for Commercial Access Control." *Computer* 21(2):59-69 (1988).

[15] Motro, A. "Integrity = Validity + Completeness." *ACM Transactions on Database Systems* 14(4):480-502 (1989).

[16] Murray, W.H. "Data Integrity in a Business Data Processing System." In [18].

[17] Murray, W.H. "On the Use of Mandatory." In [18].

[18] *Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS)*, Katzke, S.W. and Ruthberg, Z.G. (editors), NIST, Special Publication 500-160 (January 1989).

[19] *Report of the Invitational Workshop on Data Integrity*, Ruthberg, Z.G. and Polk, W.T. (editors), NIST, Special Publication 500-168 (September 1989).

[20] Saltzer, J.H. and Schroeder, M.D. "The Protection of Information in Computer Systems." *Proceedings of IEEE* 63(9):1278-1308 (1975).

[21] Sandhu, R.S. "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes." *Journal of ACM* 35(2):404-432 (1988).

[22] Sandhu, R.S. "Transaction Control Expressions for Separation of Duties." *Proc. 4th Aerospace Computer Security Applications Conference*, pages 282-286 (1988).

[23] Sandhu, R.S. "Transformation of Access Rights." *Proc. IEEE Symposium on Security and Privacy*, 259-268 (1989).

[24] Sandhu, R.S. "Mandatory Controls for Database Integrity." In *Database Security III: Status and Prospects*, Spooner, D.L. and Landwehr, C.E. (editors), North-Holland, pages 143-150 (1990). Proc. se

[25] Selinger, P.G. "Authorization and Views." In *Distributed Data Bases*, Draffan, I.W and Poole, F. (editors), Cambridge University Press, pages 233-246 (1980).

[26] Wimbrow, J.H. "A Large-Scale Interactive Administrative System." *IBM Systems Journal* 10(4):260-282 (1971).

INTEGRITY PRINCIPLE	DBMS MECHANISMS
Well-formed transactions	Encapsulated updates Atomic transactions Consistency constraints
Authenticated users	Authentication
Least privilege	Fine grain access control
Separation of duties	Transaction controls Layered updates
Reconstruction of events	Audit trail
Delegation of authority	Dynamic authorization Propagation constraints
Reality checks	Consistent snapshots
Continuity of operation	Redundancy Recovery
Ease of safe use	Fail-safe defaults Human factors

Table 1: Integrity Principles and Mechanisms

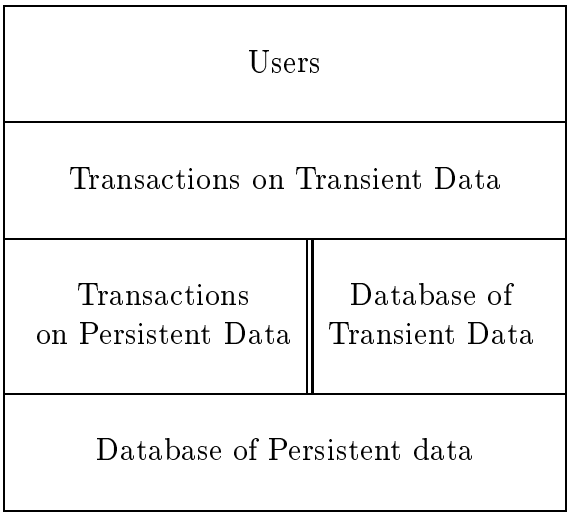

Figure 1: Layered Updates

Group I	Group II	Group III
Well-formed transactions	Least privilege	Authenticated users
Continuity of operation	Separation of duties	Reality checks
	Reconstruction of events	Ease of safe use
	Delegation of authority	

Table 2: Integrity Principles