6.2. Security Requirements

The basic security requirements of database systems are not unlike those of other computing systems we have studied. The basic problemsaccess control, exclusion of spurious data, authentication of users, and reliabilityhave appeared in many contexts so far in this book. Following is a list of requirements for database security.

- *Physical* database integrity. The data of a database are immune to physical problems, such as power failures, and someone can reconstruct the database if it is destroyed through a catastrophe.
- *Logical database integrity.* The structure of the database is preserved. With logical integrity of a database, a modification to the value of one field does not affect other fields, for example.
- *Element integrity.* The data contained in each element are accurate.
- **Auditability.** It is possible to track who or what has accessed (or modified) the elements in the database.
- *Access control.* A user is allowed to access only authorized data, and different users can be restricted to different modes of access (such as read or write).
- *User authentication.* Every user is positively identified, both for the audit trail and for permission to access certain data.
- *Availability.* Users can access the database in general and all the data for which they are authorized.

We briefly examine each of these requirements.

Integrity of the Database

If a database is to serve as a central repository of data, users must be able to trust the accuracy of the data values. This condition implies that the database administrator must be assured that updates are performed only by authorized individuals. It also implies that the data must be protected from corruption, either by an outside illegal program action or by an outside force such as fire or a power failure. Two situations can affect the integrity of a database: when the whole database is damaged (as happens, for example, if its storage medium is damaged) or when individual data items are unreadable.

Integrity of the database as a whole is the responsibility of the DBMS, the operating system, and the (human) computing system manager. From the perspective of the operating system and the computing system manager, databases and DBMSs are files and programs, respectively. Therefore, one way of protecting the database as a whole is to regularly back up all files on the system. These periodic backups can be adequate controls against catastrophic failure.

Sometimes it is important to be able to reconstruct the database at the point of a failure. For instance, when the power fails suddenly, a bank's clients may be in the middle of making transactions or students may be in the midst of registering online for their classes. In these cases, we want to be able to restore the systems to a stable point without forcing users to redo their recently completed transactions. To handle these situations, the DBMS must maintain a log of transactions. For example, suppose the banking system is designed so that a message is generated in a log (electronic or paper or both) each time a transaction is processed. In the event of a system failure, the system can obtain accurate account balances by reverting to a backup copy of the database and reprocessing all later

transactions from the log.

Element Integrity

The **integrity** of database elements is their correctness or accuracy. Ultimately, authorized users are responsible for entering correct data into databases. However, users and programs make mistakes collecting data, computing results, and entering values. Therefore, DBMSs sometimes take special action to help catch errors as they are made and to correct errors after they are inserted.

This corrective action can be taken in three ways. First, the DBMS can apply **field checks,** activities that test for appropriate values in a position. A field might be required to be numeric, an uppercase letter, or one of a set of acceptable characters. The check ensures that a value falls within specified bounds or is not greater than the sum of the values in two other fields. These checks prevent simple errors as the data are entered. (Sidebar 6-1 demonstrates the importance of element integrity.)

A second integrity action is provided by **access control.** To see why, consider life without databases. Data files may contain data from several sources, and redundant data may be stored in several different places. For example, a student's home address may be stored in many different campus files: at class registration, for dining hall privileges, at the bookstore, and in the financial aid office. Indeed, the student may not even be aware that each separate office has the address on file. If the student moves from one residence to another, each of the separate files requires correction. Without a database, there are several risks to the data's integrity. First, at a given time, there could be some data files with the old address (they have not yet been updated) and some simultaneously with the new address (they have already been updated). Second, there is always the possibility that the data fields were changed incorrectly, again leading to files with incorrect information. Third, there may be files of which the student is unaware, so he or she does not know to notify the file owner about updating the address information. These problems are solved by databases. They enable collection and control of this data at one central source, ensuring the student and users of having the correct address.

Sidebar 6-1: Element Integrity Failure Crashes Network

Crocker and Bernstein [CRO89] studied catastrophic failures of what was then known as the ARPANET, the predecessor of today's Internet. Several failures came from problems with the routing tables used to direct traffic through the network.

A 1971 error was called the "black hole." A hardware failure caused one node to declare that it was the best path to every other node in the network. This node sent this declaration to other nodes, which soon propagated the erroneous posting throughout the network. This node immediately became the black hole of the network because all traffic was routed to it but never made it to the real destination.

The ARPANET used simple tables, not a full-featured database management system, so there was no checking of new values prior to their being installed in the distributed routing tables. Had there been a database, integrity checking software could have performed error checking on the newly distributed values and raised a flag for human review.

However, the centralization is easier said than done. Who owns this shared central file? Who has authorization to update which elements? What if two people apply conflicting modifications? What if modifications are applied out of sequence? How are duplicate records detected? What action is taken when duplicates are found? These are policy questions that must be resolved by the database administrator. Sidebar 6-2 describes how these issues are addressed for managing the configuration of programs; similar formal processes are needed for

managing changes in databases.

The third means of providing database integrity is maintaining a **change log** for the database. A change log lists every change made to the database; it contains both original and modified values. Using this log, a database administrator can undo any changes that were made in error. For example, a library fine might erroneously be posted against Charles W. Robertson, instead of Charles M. Robertson, flagging Charles W. Robertson as ineligible to participate in varsity athletics. Upon discovering this error, the database administrator obtains Charles W.'s original eligibility value from the log and corrects the database.

Auditability

For some applications it may be desirable to generate an audit record of all access (read or write) to a database. Such a record can help to maintain the database's integrity, or at least to discover after the fact who had affected which values and when. A second advantage, as we see later, is that users can access protected data incrementally; that is, no single access reveals protected data, but a set of sequential accesses viewed together reveals the data, much like discovering the clues in a detective novel. In this case, an audit trail can identify which clues a user has already been given, as a guide to whether to tell the user more.

As we noted in Chapters 4 and 5, granularity becomes an impediment in auditing. Audited events in operating systems are actions like *open file* or *call procedure*; they are seldom as specific as *write record* 3 or *execute instruction* I. To be useful for maintaining integrity, database audit trails should include accesses at the record, field, and even element levels. This detail is prohibitive for most database applications.

Furthermore, it is possible for a record to be accessed but not reported to a user, as when the user performs a select operation. (Accessing a record or an element without transferring to the user the data received is called the **pass-through problem**.) Also, you can determine the values of some elements without accessing them directly. (For example, you can ask for the average salary in a group of employees when you know the number of employees in the group is only one.) Thus, a log of all records accessed directly may both overstate and understate what a user actually knows.

Sidebar 6-2: Configuration Management and Access Control

Software engineers must address access control when they manage the configurations of large computer systems. The code of a major system and changes to it over time are actually a database. In many instances multiple programmers make changes to a system at the same time; the configuration management database must help ensure that the correct and most recent changes are stored.

There are three primary ways to control the proliferation of versions and releases [PFL06a].

- *Separate files:* A separate file can be kept for each different version or release. For instance, version 1 may exist for machines that store all data in main memory, and version 2 is for machines that must put some data out to a disk. Suppose the common functions are the same in both versions, residing in components C1 through C_k, but memory management is done by component M_1 for version 1 and M_2 for version 2. If new functionality is to be added to the memory management routines, keeping both versions current and correct may be difficult; the results must be the same from the user's point of view.
- *Deltas:* One version of the system is deemed the main version, and all other versions are considered to be variations from the main version. The

database keeps track only of the differences, in a file called a *delta* file. The delta contains commands that are "applied" to the main version to transform it into the alternative version. This approach saves storage space but can become unwieldy.

- *Conditional compilation:* All versions are handled by a single file, and conditional statements are used to determine which statements apply under which conditions. In this case, shared code appears only once, so only one correction is needed if a problem is found. But the code in this single file can be very complex and difficult to maintain.

In any of these three cases, it is essential to control access to the configuration files. It is common practice for two different programmers fixing different problems to need to make changes to the same component. If care is not taken in controlling access, then the second programmer can inadvertently "undo" the changes made by the first programmer, resulting in not only recurrence of the initial problems but also introduction of additional problems. For this reason, files are controlled in several ways, including being locked while changes are made by one programmer, and being subject to a group of people called a configuration control board who ensure that no changed file is put back into production without the proper checking and testing. More information about these techniques can be found in [PFL06a].

Access Control

Databases are often separated logically by user access privileges. For example, all users can be granted access to general data, but only the personnel department can obtain salary data and only the marketing department can obtain sales data. Databases are very useful because they centralize the storage and maintenance of data. Limited access is both a responsibility and a benefit of this centralization.

The database administrator specifies who should be allowed access to which data, at the view, relation, field, record, or even element level. The DBMS must enforce this policy, granting access to all specified data or no access where prohibited. Furthermore, the number of modes of access can be many. A user or program may have the right to read, change, delete, or append to a value, add or delete entire fields or records, or reorganize the entire database.

Superficially, access control for a database seems like access control for operating systems or any other component of a computing system. However, the database problem is more complicated, as we see throughout this chapter. Operating system objects, such as files, are unrelated items, whereas records, fields, and elements are related. Although a user cannot determine the contents of one file by reading others, a user might be able to determine one data element just by reading others. The problem of obtaining data values from others is called **inference**, and we consider it in depth later in this chapter.

It is important to notice that you can access data by inference without needing direct access to the secure object itself. Restricting inference may mean prohibiting certain paths to prevent possible inferences. However, restricting access to control inference also limits queries from users who do not intend unauthorized access to values. Moreover, attempts to check requested accesses for possible unacceptable inferences may actually degrade the DBMS's performance.

Finally, size or granularity is different between operating system objects and database objects. An access control list of several hundred files is much easier to implement than an access control list for a database with several hundred files of perhaps a hundred fields each. Size affects the efficiency of processing.

User Authentication

The DBMS can require rigorous user authentication. For example, a DBMS might insist that a user pass both specific password and time-of-day checks. This authentication supplements the authentication performed by the operating system. Typically, the DBMS runs as an application program on top of the operating system. This system design means that there is no trusted path from the DBMS to the operating system, so the DBMS must be suspicious of any data it receives, including user authentication. Thus, the DBMS is forced to do its own authentication.

Availability

A DBMS has aspects of both a program and a system. It is a program that uses other hardware and software resources, yet to many users it is the only application run. Users often take the DBMS for granted, employing it as an essential tool with which to perform particular tasks. But when the system is not availablebusy serving other users or down to be repaired or upgradedthe users are very aware of a DBMS's unavailability. For example, two users may request the same record, and the DBMS must arbitrate; one user is bound to be denied access for a while. Or the DBMS may withhold unprotected data to avoid revealing protected data, leaving the requesting user unhappy. We examine these problems in more detail later in this chapter. Problems like these result in high availability requirements for a DBMS.

Integrity/Confidentiality/Availability

The three aspects of computer securityintegrity, confidentiality, and availabilityclearly relate to database management systems. As we have described, integrity applies to the individual elements of a database as well as to the database as a whole. Thus, integrity is a major concern in the design of database management systems. We look more closely at integrity issues in the next section.

Confidentiality is a key issue with databases because of the inference problem, whereby a user can access sensitive data indirectly. Inference and access control are covered later in this chapter.

Finally, availability is important because of the shared access motivation underlying database development. However, availability conflicts with confidentiality. The last sections of the chapter address availability in an environment in which confidentiality is also important.

PREV NEXT

6.3. Reliability and Integrity

Databases amalgamate data from many sources, and users expect a DBMS to provide access to the data in a reliable way. When software engineers say that software has **reliability**, they mean that the software runs for very long periods of time without failing. Users certainly expect a DBMS to be reliable, since the data usually are key to business or organizational needs. Moreover, users entrust their data to a DBMS and rightly expect it to protect the data from loss or damage. Concerns for reliability and integrity are general security issues, but they are more apparent with databases.

A DBMS guards against loss or damage in several ways that we study them in this section. However, the controls we consider are not absolute: No control can prevent an authorized user from inadvertently entering an acceptable but incorrect value.

Database concerns about reliability and integrity can be viewed from three dimensions:

- *Database integrity:* concern that the database as a whole is protected against damage, as from the failure of a disk drive or the corruption of the master database index. These concerns are addressed by operating system integrity controls and recovery procedures.
- *Element integrity:* concern that the value of a specific data element is written or changed only by authorized users. Proper access controls protect a database from corruption by unauthorized users.
- *Element accuracy:* concern that only correct values are written into the elements of a database. Checks on the values of elements can help prevent insertion of improper values. Also, constraint conditions can detect incorrect values.

Protection Features from the Operating System

In Chapter 4 we discussed the protection an operating system provides for its users. A responsible system administrator backs up the files of a database periodically along with other user files. The files are protected during normal execution against outside access by the operating system's standard access control facilities. Finally, the operating system performs certain integrity checks for all data as a part of normal read and write operations for I/O devices. These controls provide basic security for databases, but the database manager must enhance them.

Two-Phase Update

A serious problem for a database manager is the failure of the computing system in the middle of modifying data. If the data item to be modified was a long field, half of the field might show the new value, while the other half would contain the old. Even if errors of this type were spotted easily (which they are not), a more subtle problem occurs when several fields are updated and no single field appears to be in obvious error. The solution to this problem, proposed first by Lampson and Sturgis [LAM76] and adopted by most DBMSs, uses a two-phase update.

Update Technique

During the first phase, called the **intent** phase, the DBMS gathers the resources it needs to perform the update. It may gather data, create dummy records, open files, lock out other users, and calculate final answers; in short, it does everything to prepare for the update, but it makes no changes to the database. The first phase is repeatable an unlimited number of times because it takes no permanent action. If the system fails during execution of the first phase, no harm is done because all these steps can be restarted and repeated after the

system resumes processing.

The last event of the first phase, called **committing,** involves the writing of a **commit flag** to the database. The commit flag means that the DBMS has passed the point of no return: After committing, the DBMS begins making permanent changes.

The second phase makes the permanent changes. During the second phase, no actions from before the commit can be repeated, but the update activities of phase two can also be repeated as often as needed. If the system fails during the second phase, the database may contain incomplete data, but the system can repair these data by performing all activities of the second phase. After the second phase has been completed, the database is again complete.

Two-Phase Update Example

Suppose a database contains an inventory of a company's office supplies. The company's central stockroom stores paper, pens, paper clips, and the like, and the different departments requisition items as they need them. The company buys in bulk to obtain the best prices. Each department has a budget for office supplies, so there is a charging mechanism by which the cost of supplies is recovered from the department. Also, the central stockroom monitors quantities of supplies on hand so as to order new supplies when the stock becomes low.

Suppose the process begins with a requisition from the accounting department for 50 boxes of paper clips. Assume that there are 107 boxes in stock and a new order is placed if the quantity in stock ever falls below 100. Here are the steps followed after the stockroom receives the requisition.

1. The stockroom checks the database to determine that 50 boxes of paper clips are on hand. If not, the requisition is rejected and the transaction is finished.

2. If enough paper clips are in stock, the stockroom deducts 50 from the inventory figure in the database (107 - 50 = 57).

3. The stockroom charges accounting's supplies budget (also in the database) for 50 boxes of paper clips.

4. The stockroom checks its remaining quantity on hand (57) to determine whether the remaining quantity is below the reorder point. Because it is, a notice to order more paper clips is generated, and the item is flagged as "on order" in the database.

5. A delivery order is prepared, enabling 50 boxes of paper clips to be sent to accounting.

All five of these steps must be completed in the order listed for the database to be accurate and for the transaction to be processed correctly.

Suppose a failure occurs while these steps are being processed. If the failure occurs before step 1 is complete, there is no harm because the entire transaction can be restarted. However, during steps 2, 3, and 4, changes are made to elements in the database. If a failure occurs then, the values in the database are inconsistent. Worse, the transaction cannot be reprocessed because a requisition would be deducted twice, or a department would be charged twice, or two delivery orders would be prepared.

When a two-phase commit is used, **shadow values** are maintained for key data points. A shadow data value is computed and stored locally during the intent phase, and it is copied to the actual database during the commit phase. The operations on the database would be performed as follows for a two-phase commit.

Intent:

1. Check the value of COMMIT-FLAG in the database. If it is set, this phase cannot be performed. Halt or loop, checking COMMIT-FLAG until it is not set.

2. Compare number of boxes of paper clips on hand to number requisitioned; if more are requisitioned than are on hand, halt.

3. Compute TCLIPS = ONHAND - REQUISITION.

4. Obtain BUDGET, the current supplies budget remaining for accounting department. Compute TBUDGET = BUDGET - COST, where COST is the cost of 50 boxes of clips.

5. Check whether TCLIPS is below reorder point; if so, set TREORDER = TRUE; else set TREORDER = FALSE.

Commit:

1. Set COMMIT-FLAG in database.

2. Copy TCLIPS to CLIPS in database.

3. Copy TBUDGET to BUDGET in database.

4. Copy TREORDER to REORDER in database.

5. Prepare notice to deliver paper clips to accounting department. Indicate transaction completed in log.

6. Unset COMMIT-FLAG.

With this example, each step of the intent phase depends only on unmodified values from the database and the previous results of the intent phase. Each variable beginning with T is a shadow variable used only in this transaction. The steps of the intent phase can be repeated an unlimited number of times without affecting the integrity of the database.

Once the DBMS begins the commit phase, it writes a commit flag. When this flag is set, the DBMS will not perform any steps of the intent phase. Intent steps cannot be performed after committing because database values are modified in the commit phase. Notice, however, that the steps of the commit phase can be repeated an unlimited number of times, again with no negative effect on the correctness of the values in the database.

The one remaining flaw in this logic occurs if the system fails after writing the "transaction complete" message in the log but before clearing the commit flag in the database. It is a simple matter to work backward through the transaction log to find completed transactions for which the commit flag is still set and to clear those flags.

Redundancy/Internal Consistency

Many DBMSs maintain additional information to detect internal inconsistencies in data. The additional information ranges from a few check bits to duplicate or shadow fields, depending on the importance of the data.

Error Detection and Correction Codes

One form of redundancy is error detection and correction codes, such as parity bits, Hamming codes, and cyclic redundancy checks. These codes can be applied to single fields, records, or

the entire database. Each time a data item is placed in the database, the appropriate check codes are computed and stored; each time a data item is retrieved, a similar check code is computed and compared to the stored value. If the values are unequal, they signify to the DBMS that an error has occurred in the database. Some of these codes point out the place of the error; others show precisely what the correct value should be. The more information provided, the more space required to store the codes.

Shadow Fields

Entire attributes or entire records can be duplicated in a database. If the data are irreproducible, this second copy can provide an immediate replacement if an error is detected. Obviously, redundant fields require substantial storage space.

Recovery

In addition to these error correction processes, a DBMS can maintain a log of user accesses, particularly changes. In the event of a failure, the database is reloaded from a backup copy and all later changes are then applied from the audit log.

Concurrency/Consistency

Database systems are often multiuser systems. Accesses by two users sharing the same database must be constrained so that neither interferes with the other. Simple locking is done by the DBMS. If two users attempt to read the same data item, there is no conflict because both obtain the same value.

If both users try to modify the same data items, we often assume that there is no conflict because each knows what to write; the value to be written does not depend on the previous value of the data item. However, this supposition is not quite accurate.

To see how concurrent modification can get us into trouble, suppose that the database consists of seat reservations for a particular airline flight. Agent A, booking a seat for passenger Mock, submits a query to find which seats are still available. The agent knows that Mock prefers a right aisle seat, and the agent finds that seats 5D, 11D, and 14D are open. At the same time, Agent B is trying to book seats for a family of three traveling together. In response to a query, the database indicates that 8ABC and 11DEF are the two remaining groups of three adjacent unassigned seats. Agent A submits the update command

```
SELECT (SEAT-NO = '11D')
ASSIGN 'MOCK,E' TO PASSENGER-NAME
```

while Agent B submits the update sequence

```
SELECT (SEAT-NO = '11D')
ASSIGN 'EHLERS,P' TO PASSENGER-NAME
```

as well as commands for seats 11E and 11F. Then two passengers have been booked into the same seat (which would be uncomfortable, to say the least).

Both agents have acted properly: Each sought a list of empty seats, chose one seat from the list, and updated the database to show to whom the seat was assigned. The difficulty in this situation is the time delay between reading a value from the database and writing a modification of that value. During the delay time, another user has accessed the same data.

To resolve this problem, a DBMS treats the entire queryupdate cycle as a single atomic operation. The command from the agent must now resemble "read the current value of seat PASSENGER-NAME for seat 11D; if it is 'UNASSIGNED', modify it to 'MOCK,E' (or 'EHLERS,P')." The readmodify cycle must be completed as an uninterrupted item without allowing any other users access to the PASSENGER-NAME field for seat 11D. The second agent's request to book would not be considered until after the first agent's had been completed; at that time, the

value of PASSENGERNAME would no longer be 'UNASSIGNED'.

A final problem in concurrent access is readwrite. Suppose one user is updating a value when a second user wishes to read it. If the read is done while the write is in progress, the reader may receive data that are only partially updated. Consequently, the DBMS locks any read requests until a write has been completed.

Monitors

The **monitor** is the unit of a DBMS responsible for the structural integrity of the database. A monitor can check values being entered to ensure their consistency with the rest of the database or with characteristics of the particular field. For example, a monitor might reject alphabetic characters for a numeric field. We discuss several forms of monitors.

Range Comparisons

A range comparison monitor tests each new value to ensure that the value is within an acceptable range. If the data value is outside the range, it is rejected and not entered into the database. For example, the range of dates might be 131, "/," 112, "/," 19002099. An even more sophisticated range check might limit the day portion to 130 for months with 30 days, or it might take into account leap year for February.

Range comparisons are also convenient for numeric quantities. For example, a salary field might be limited to $200,000, or the size of a house might be constrained to be between 500 and 5,000 square feet. Range constraints can also apply to other data having a predictable form.

Range comparisons can be used to ensure the internal consistency of a database. When used in this manner, comparisons are made between two database elements. For example, a grade level from K8 would be acceptable if the record described a student at an elementary school, whereas only 912 would be acceptable for a record of a student in high school. Similarly, a person could be assigned a job qualification score of 75100 only if the person had completed college or had had at least ten years of work experience. **Filters** or **patterns** are more general types of data form checks. These can be used to verify that an automobile plate is two letters followed by four digits, or the sum of all digits of a credit card number is a multiple of 9.

Checks of these types can control the data allowed in the database. They can also be used to test existing values for reasonableness. If you suspect that the data in a database have been corrupted, a range check of all records could identify those having suspicious values.

State Constraints

State constraints describe the condition of the entire database. At no time should the database values violate these constraints. Phrased differently, if these constraints are not met, some value of the database is in error.

In the section on two-phase updates, we saw how to use a commit flag, which is set at the start of the commit phase and cleared at the completion of the commit phase. The commit flag can be considered a state constraint because it is used at the end of every transaction for which the commit flag is not set. Earlier in this chapter, we described a process to reset the commit flags in the event of a failure after a commit phase. In this way, the status of the commit flag is an integrity constraint on the database.

For another example of a state constraint, consider a database of employees' classifications. At any time, at most one employee is classified as "president." Furthermore, each employee has an employee number different from that of every other employee. If a mechanical or software failure causes portions of the database file to be duplicated, one of these uniqueness constraints might be violated. By testing the state of the database, the DBMS could identify records with duplicate employee numbers or two records classified as

"president."

Transition Constraints

State constraints describe the state of a correct database. **Transition constraints** describe conditions necessary before changes can be applied to a database. For example, before a new employee can be added to the database, there must be a position number in the database with status "vacant." (That is, an empty slot must exist.) Furthermore, after the employee is added, exactly one slot must be changed from "vacant" to the number of the new employee.

Simple range checks and filters can be implemented within most database management systems. However, the more sophisticated state and transition constraints can require special procedures for testing. Such user-written procedures are invoked by the DBMS each time an action must be checked.

Summary of Data Reliability

Reliability, correctness, and integrity are three closely related concepts in databases. Users trust the DBMS to maintain their data correctly, so integrity issues are very important to database security.

PREV NEXT

6.4. Sensitive Data

Some databases contain what is called sensitive data. As a working definition, let us say that **sensitive data** are data that should not be made public. Determining which data items and fields are sensitive depends both on the individual database and the underlying meaning of the data. Obviously, some databases, such as a public library catalog, contain no sensitive data; other databases, such as defense-related ones, are totally sensitive. These two casesnothing sensitive and everything sensitiveare the easiest to handle because they can be covered by access controls to the database as a whole. Someone either is or is not an authorized user. These controls are provided by the operating system.

The more difficult problem, which is also the more interesting one, is the case in which *some but not all* of the elements in the database are sensitive. There may be varying degrees of sensitivity. For example, a university database might contain student data consisting of name, financial aid, dorm, drug use, sex, parking fines, and race. An example of this database is shown in Table 6-6. Name and dorm are probably the least sensitive; financial aid, parking fines, and drug use the most; sex and race somewhere in between. That is, many people may have legitimate access to name, some to sex and race, and relatively few to financial aid, parking fines, or drug use. Indeed, knowledge of the existence of some fields, such as drug use, may itself be sensitive. Thus, security concerns not only the data elements but also their context and meaning.

Table 6-6. Sample Database.

Name	Sex	Race	Aid	Fines	Drugs	Dorm
Adams	M	C	5000	45.	1	Holmes
Bailey	M	B	0	0.	0	Grey
Chin	F	A	3000	20.	0	West
Dewitt	M	B	1000	35.	3	Grey
Earhart	F	C	2000	95.	1	Holmes
Fein	F	C	1000	15.	0	West
Groff	M	C	4000	0.	3	West
Hill	F	B	5000	10.	2	Holmes
Koch	F	C	0	0.	1	West
Liu	F	A	0	10.	2	Grey
Majors	M	C	2000	0.	2	Grey

Furthermore, we must take into account different degrees of sensitivity. For instance, although they are all highly sensitive, the financial aid, parking fines, and drug-use fields may not have the same kinds of access restrictions. Our security requirements may demand that a

few people be authorized to see each field, but no one be authorized to see all three. The challenge of the access control problem is to limit users' access so that they can obtain only the data to which they have legitimate access. Alternatively, the access control problem forces us to ensure that sensitive data are not to be released to unauthorized people.

Several factors can make data sensitive.

- *Inherently sensitive.* The value itself may be so revealing that it is sensitive. Examples are the locations of defensive missiles or the median income of barbers in a town with only one barber.
- *From a sensitive source.* The source of the data may indicate a need for confidentiality. An example is information from an informer whose identity would be compromised if the information were disclosed.
- *Declared sensitive.* The database administrator or the owner of the data may have declared the data to be sensitive. Examples are classified military data or the name of the anonymous donor of a piece of art.
- Part of a sensitive *attribute* or a sensitive *record.* In a database, an entire attribute or record may be classified as sensitive. Examples are the salary attribute of a personnel database or a record describing a secret space mission.
- Sensitive *in relation to previously disclosed information.* Some data become sensitive in the presence of other data. For example, the longitude coordinate of a secret gold mine reveals little, but the longitude coordinate in conjunction with the latitude coordinate pinpoints the mine.

All of these factors must be considered to determine the sensitivity of the data.

Access Decisions

Remember that a database administrator is a *person* who decides *what* data should be in the database and *who* should have access to it. The database administrator considers the need for different users to know certain information and decides who should have what access. Decisions of the database administrator are based on an access *policy*.

The database manager or DBMS is a *program* that operates on the database and auxiliary control information to implement the decisions of the access policy. We say that the database manager decides to permit user *x* to access data *y*. Clearly, a program or machine cannot decide anything; it is more precise to say that the program performs the instructions by which *x* accesses *y* as a way of implementing the policy established by the database administrator. (Now you see why we use the simpler wording.) To keep explanations concise, we occasionally describe programs as if they can carry out human thought processes.

The DBMS may consider several factors when deciding whether to permit an access. These factors include availability of the data, acceptability of the access, and authenticity of the user. We expand on these three factors below.

Availability of Data

One or more required elements may be inaccessible. For example, if a user is updating several fields, other users' accesses to those fields must be blocked temporarily. This blocking ensures that users do not receive inaccurate information, such as a new street address with an old city and state, or a new code component with old documentation. Blocking is usually temporary. When performing an update, a user may have to block access to several fields or several records to ensure the consistency of data for others.

Notice, however, that if the updating user aborts the transaction while the update is in progress, the other users may be permanently blocked from accessing the record. This indefinite postponement is also a security problem, resulting in denial of service.

Acceptability of Access

One or more values of the record may be sensitive and not accessible by the general user. A DBMS should not release sensitive data to unauthorized individuals.

Deciding what is sensitive, however, is not as simple as it sounds, because the fields may not be directly requested. A user may have asked for certain records that contain sensitive data, but the user's purpose may have been only to project the values from particular fields that are not sensitive. For example, a user of the database shown in Table 6-6 may request the NAME and DORM of any student for whom FINES is not 0. The exact value of the sensitive field FINES is not disclosed, although "not 0" is a partial disclosure. Even when a sensitive value is not explicitly given, the database manager may deny access on the grounds that it reveals information the user is not authorized to have.

Alternatively, the user may want to derive a nonsensitive statistic from the sensitive data; for example, if the average financial aid value does not reveal any individual's financial aid value, the database management system can safely return the average. However, the average of one data value discloses that value.

Assurance of Authenticity

Certain characteristics of the user external to the database may also be considered when permitting access. For example, to enhance security, the database administrator may permit someone to access the database only at certain times, such as during working hours. Previous user requests may also be taken into account; repeated requests for the same data or requests that exhaust a certain category of information may be used to find out all elements in a set when a direct query is not allowed. As we shall see, sensitive data can sometimes be revealed by combined results from several less sensitive queries.

Types of Disclosures

Data can be sensitive, but so can their characteristics. In this section, we see that even descriptive information about data (such as their existence or whether they have an element that is zero) is a form of disclosure.

Exact Data

The most serious disclosure is the *exact value of a sensitive data item* itself. The user may know that sensitive data are being requested, or the user may request general data without knowing that some of it is sensitive. A faulty database manager may even deliver sensitive data by accident, without the user's having requested it. In all of these cases the result is the same: The security of the sensitive data has been breached.

Bounds

Another exposure is disclosing bounds on a sensitive value; that is, indicating that a sensitive value, y, is between two values, L and H. Sometimes, by using a narrowing technique not unlike the binary search, the user may first determine that $L \leq y \leq H$ and then see whether $L \leq y \leq H/2$, and so forth, thereby permitting the user to determine y to any desired precision. In another case, merely revealing that a value such as the athletic scholarship budget or the number of CIA agents exceeds a certain amount may be a serious breach of security.

Sometimes, however, bounds are a useful way to present sensitive data. It is common to release upper and lower bounds for data without identifying the specific records. For example, a company may announce that its salaries for programmers range from $50,000 to $82,000. If you are a programmer earning $79,700, you can presume that you are fairly well off, so you have the information you want; however, the announcement does not disclose who are the highest- and lowest-paid programmers.

Negative Result

Sometimes we can word a query to determine a negative result. That is, we can learn that *z* is *not* the value of *y*. For example, knowing that 0 is not the total number of felony convictions for a person reveals that the person was convicted of a felony. The distinction between 1 and 2 or 46 and 47 felonies is not as sensitive as the distinction between 0 and 1. Therefore, disclosing that a value is not 0 can be a significant disclosure. Similarly, if a student does not appear on the honors list, you can infer that the person's grade point average is below 3.50. This information is not too revealing, however, because the range of grade point averages from 0.0 to 3.49 is rather wide.

Existence

In some cases, the existence of data is itself a sensitive piece of data, regardless of the actual value. For example, an employer may not want employees to know that their use of long distance telephone lines is being monitored. In this case, discovering a LONG DISTANCE field in a personnel file would reveal sensitive data.

Probable Value

Finally, it may be possible to determine the probability that a certain element has a certain value. To see how, suppose you want to find out whether the president of the United States is registered in the Tory party. Knowing that the president is in the database, you submit two queries to the database:

How many people have 1600 Pennsylvania Avenue as their official residence? (Response: 4)

How many people have 1600 Pennsylvania Avenue as their official residence and have YES as the value of TORY? (Response: 1)

From these queries you conclude there is a 25 percent likelihood that the president is a registered Tory.

Summary of Partial Disclosure

We have seen several examples of how a security problem can result if characteristics of sensitive data are revealed. Notice that some of the techniques we presented used information *about* the data, rather than direct access to the data, to infer sensitive results. A successful security strategy must protect from both direct and indirect disclosure.

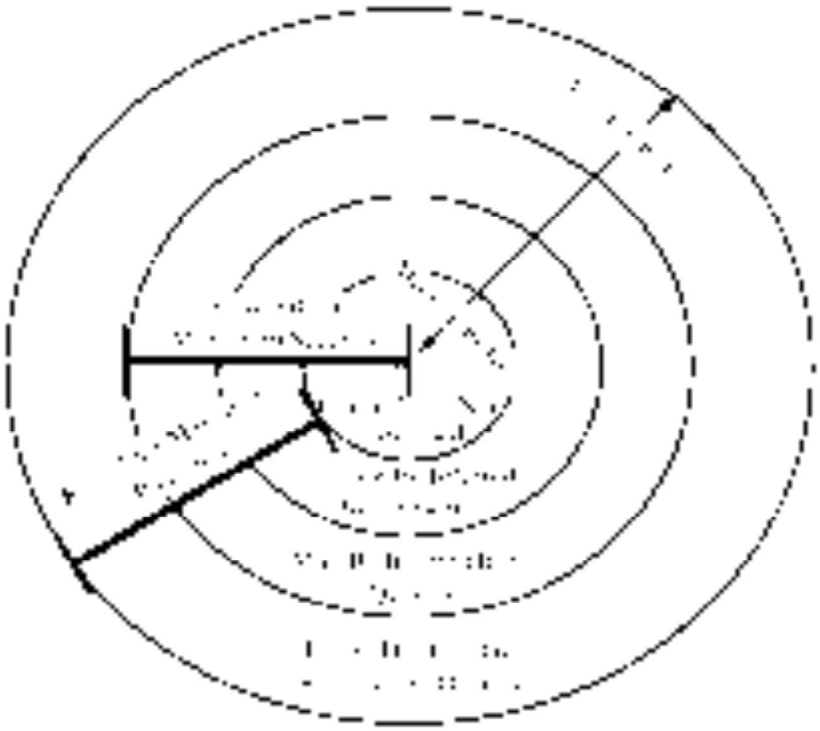
Security versus Precision

Our examples have illustrated how difficult it is to determine which data are sensitive and how to protect them. The situation is complicated by a desire to share nonsensitive data. For reasons of confidentiality we want to disclose only those data that are not sensitive. Such an outlook encourages a conservative philosophy in determining what data to disclose: less is better than more.

On the other hand, consider the users of the data. The conservative philosophy suggests rejecting any query that mentions a sensitive field. We may thereby reject many reasonable and nondisclosing queries. For example, a researcher may want a list of grades for all students using drugs, or a statistician may request lists of salaries for all men and for all women. These queries probably do not compromise the identity of any individual. We want to disclose as much data as possible so that users of the database have access to the data they need. This goal, called **precision,** aims to protect all sensitive data while revealing as much nonsensitive data as possible.

We can depict the relationship between security and precision with concentric circles. As Figure 6-3 shows, the sensitive data in the central circle should be carefully concealed. The outside band represents data we willingly disclose in response to queries. But we know that the user may put together pieces of disclosed data and infer other, more deeply hidden, data. The figure shows us that beneath the outer layer may be yet more nonsensitive data that the user cannot infer.

Figure 6-3. Security versus Precision.



The ideal combination of security and precision allows us to maintain perfect confidentiality with maximum precision; in other words, we disclose all and only the nonsensitive data. But achieving this goal is not as easy as it might seem, as we show in the next section. Sidebar 6-3 gives an example of using imprecise techniques to improve accuracy. In the next section, we consider ways in which sensitive data can be obtained from queries that appear harmless.

Sidebar 6-3: Accuracy and Imprecision

Article I of the U.S. Constitution charges Congress with determining the "respective numbers… of free…and all other persons…within every…term of ten years." This count is used for many things, including apportioning the number of representatives to Congress and distributing funds fairly to the states. Although difficult in 1787, this task has become increasingly challenging. The count cannot simply be based on residences, because some homeless people would be missed. A fair count cannot be obtained solely by sending a questionnaire for each person to complete and return, because some people cannot read and, more significantly, many people do not return such forms. And there is always the possibility that a form would be lost in the mail.

For the 2000 census the U.S. Census Bureau proposed using statistical sampling and estimating techniques to approximate the population. With these techniques they would select certain areas in which to take two counts: a regular count and a second, especially diligent search for every person residing in the area. In this way the bureau could determine the "undercount," the number of people missed in the regular count. They could then use this undercount factor to adjust the regular count in other similar areas and thus obtain a more accurate, although less precise, count.

The Supreme Court ruled that statistical sampling techniques were acceptable for determining revenue distribution to the states but not for allocating representatives in Congress. As a result, the census can never get an exact, accurate count of the number of people in the United States or even in a major

U.S. city. At the same time, concerns about precision and privacy prevent the Census Bureau from releasing information about any particular individual living in the United States.

Does this lack of accuracy and exactness mean that the census is not useful? No. We may not know exactly how many people live in Washington, D.C., or the exact information about a particular resident of Washington, D.C., but we can use the census information to characterize the residents of Washington, D.C. For example, we can determine the maximum, minimum, mean, and median ages or incomes, and we can investigate the relationships among characteristics, such as between education level and income. So accuracy and precision help to reflect the balance between protection and need to know.

PREV NEXT

6.5. Inference

Inference is a way to infer or derive sensitive data from nonsensitive data. The inference problem is a subtle vulnerability in database security.

The database in Table 6-7 can help illustrate the inference problem. Recall that AID is the amount of financial aid a student is receiving. FINES is the amount of parking fines still owed. DRUGS is the result of a drug-use survey: 0 means never used and 3 means frequent user. Obviously this information should be kept confidential. We assume that AID, FINES, and DRUGS are sensitive fields, although only when the values are related to a specific individual. In this section, we look at ways to determine sensitive data values from the database.

Table 6-7. Sample Database (repeated).

Name	Sex	Race	Aid	Fines	Drugs	Dorm
Adams	M	C	5000	45.	1	Holmes
Bailey	M	B	0	0.	0	Grey
Chin	F	A	3000	20.	0	West
Dewitt	M	B	1000	35.	3	Grey
Earhart	F	C	2000	95.	1	Holmes
Fein	F	C	1000	15.	0	West
Groff	M	C	4000	0.	3	West
Hill	F	B	5000	10.	2	Holmes
Koch	F	C	0	0.	1	West
Liu	F	A	0	10.	2	Grey
Majors	M	C	2000	0.	2	Grey

Direct Attack

In a direct attack, a user tries to determine values of sensitive fields by seeking them directly with queries that yield few records. The most successful technique is to form a query so specific that it matches exactly one data item.

In Table 6-7, a sensitive query might be

```
List NAME where
      SEX=M ∧ DRUGS=1
```

This query discloses that for record ADAMS, DRUGS=1. However, it is an obvious attack

because it selects people for whom DRUGS=1, and the DBMS might reject the query because it selects records for a specific value of the sensitive attribute DRUGS.

A less obvious query is

```
List NAME where
      (SEX=M ∧ DRUGS=1) ∨
      (SEX≠M ∧ SEX≠F) ∨
      (DORM=AYRES)
```

On the surface, this query looks as if it should conceal drug usage by selecting other non-drug-related records as well. However, this query still retrieves only one record, revealing a name that corresponds to the sensitive DRUG value. The DBMS needs to know that SEX has only two possible values so that the second clause will select no records. Even if that were possible, the DBMS would also need to know that no records exist with DORM=AYRES, even though AYRES might in fact be an acceptable value for DORM.

Organizations that publish personal statistical data, such as the U.S. Census Bureau, do not reveal results when a small number of people make up a large proportion of a category. The rule of "*n* items over *k* percent" means that data should be withheld if *n* items represent over *k* percent of the result reported. In the previous case, the one person selected represents 100 percent of the data reported, so there would be no ambiguity about which person matches the query.

Indirect Attack

Another procedure, used by the U.S. Census Bureau and other organizations that gather sensitive data, is to release only statistics. The organizations suppress individual names, addresses, or other characteristics by which a single individual can be recognized. Only neutral statistics, such as sum, count, and mean, are released.

The indirect attack seeks to infer a final result based on one or more intermediate statistical results. But this approach requires work outside the database itself. In particular, a statistical attack seeks to use some apparently anonymous statistical measure to infer individual data. In the following sections, we present several examples of indirect attacks on databases that report statistics.

Sum

An attack by sum tries to infer a value from a reported sum. For example, with the sample database in Table 6-7, it might seem safe to report student aid total by sex and dorm. Such a report is shown in Table 6-8. This seemingly innocent report reveals that no female living in Grey is receiving financial aid. Thus, we can infer that any female living in Grey (such as Liu) is certainly not receiving financial aid. This approach often allows us to determine a negative result.

Table 6-8. Sums of Financial Aid by Dorm and Sex.

	Holmes	Grey	West	Total
M	5000	3000	4000	12000
F	7000	0	4000	11000
Total	12000	3000	8000	23000

Count

The count can be combined with the sum to produce some even more revealing results. Often these two statistics are released for a database to allow users to determine average values. (Conversely, if count and mean are released, sum can be deduced.)

Table 6-9 shows the count of records for students by dorm and sex. This table is innocuous by itself. Combined with the sum table, however, this table demonstrates that the two males in Holmes and West are receiving financial aid in the amount of $5000 and $4000, respectively. We can obtain the names by selecting the subschema of NAME, DORM, which is not sensitive because it delivers only low-security data on the entire database.

Table 6-9. Count of Students by Dorm and Sex.

	Holmes	Grey	West	Total
M	1	3	1	5
F	2	1	3	6
Total	3	4	4	11

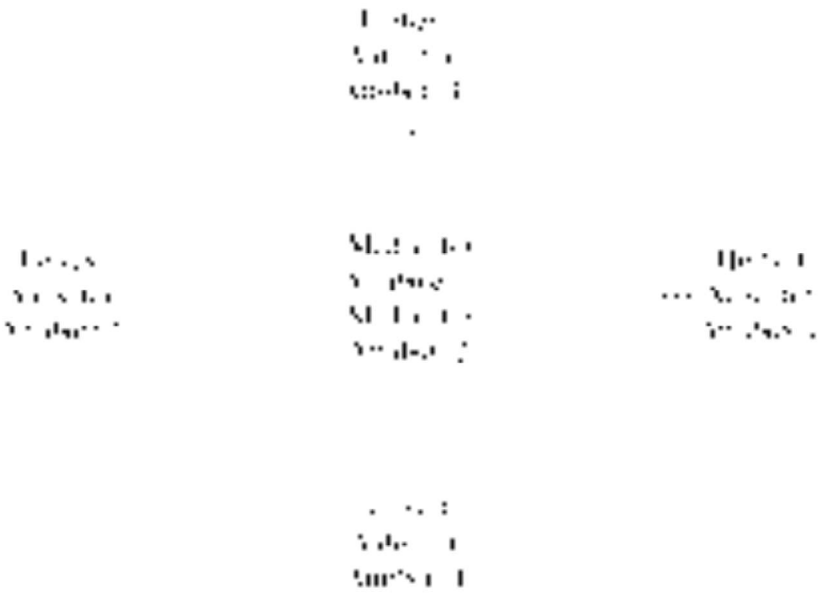
Mean

The arithmetic mean (average) allows exact disclosure if the attacker can manipulate the subject population. As a trivial example, consider salary. Given the number of employees, the mean salary for a company and the mean salary of all employees except the president, it is easy to compute the president's salary.

Median

By a slightly more complicated process, we can determine an individual value from medians. The attack requires finding selections having one point of intersection that happens to be exactly in the middle, as shown in Figure 6-4.

Figure 6-4. Intersecting Medians.



For example, in our sample database, there are five males and three persons whose drug use value is 2. Arranged in order of aid, these lists are shown in Table 6-10. Notice that Majors is the only name common to both lists, and conveniently that name is in the middle of each list. Someone working at the Health Clinic might be able to find out that Majors is a white male whose drug-use score is 2. That information identifies Majors as the intersection of these two lists and pinpoints Majors' financial aid as $2000. In this example, the queries

Table 6-10. Inference from Median of Two Lists.

Name	Sex	Drugs	Aid
Bailey	M	0	0
Dewitt	M	3	1000
Majors	M	2	2000
Groff	M	3	4000
Adams	M	1	5000
Liu	F	2	0
Majors	M	2	2000
Hill	F	2	5000

```
q = median(AID where SEX = M)
p = median(AID where DRUGS = 2)
```

reveal the exact financial aid amount for Majors.

Tracker Attacks

As already explained, database management systems may conceal data when a small number of entries make up a large proportion of the data revealed. A **tracker attack** can fool the database manager into locating the desired data by using additional queries that produce small results. The tracker adds additional records to be retrieved for two different queries; the two sets of records cancel each other out, leaving only the statistic or data desired. The approach is to use intelligent padding of two queries. In other words, instead of trying to identify a unique value, we request *n* - 1 other values (where there are *n* values in the database). Given *n* and *n* - 1, we can easily compute the desired single element.

For instance, suppose we wish to know how many female Caucasians live in Holmes Hall. A query posed might be

count ((SEX=F) ∧ (RACE=C) ∧ (DORM=Holmes))

The database management system might consult the database, find that the answer is 1, and refuse to answer that query because one record dominates the result of the query.

However, further analysis of the query allows us to track sensitive data through nonsensitive queries.

The query

q=count((SEX=F) ∧ (RACE=C) ∧ (DORM=Holmes))

is of the form

q = count(a ∧ b ∧ c)

By using the rules of logic and algebra, we can transform this query to

q = count(a ∧ b ∧ c) = count(a) count(a ∧ ¬ (b ∧ c))

Thus, the original query is equivalent to

count (SEX=F)

minus

count ((SEX=F) ∧ ((RACE≠C) ∨ (DORM≠Holmes)))

Because count(*a*) = 6 and count(*a* ∧¬ (*b* ∧*c*)) = 5, we can determine the suppressed value easily: 6 - 5 = 1. Furthermore, neither 6 nor 5 is a sensitive count.

Linear System Vulnerability

A tracker is a specific case of a more general vulnerability. With a little logic, algebra, and luck in the distribution of the database contents, it may be possible to construct a series of queries that returns results relating to several different sets. For example, the following system of five queries does not overtly reveal any single *c* value from the database. However, the queries' equations can be solved for each of the unknown *c* values, revealing them all.

$$
\begin{aligned}
q_1 &= c_1 + c_2 + c_3 + c_4 + c_5 \\
q_2 &= c_1 + c_2 + c_4 \\
q_3 &= c_3 + c_4 \\
q_4 &= c_4 + c_5 \\
q_5 &= c_2 + c_5
\end{aligned}
$$

To see how, use basic algebra to note that $q_1 - q_2 = c_3 + c_5$, and $q_3 - q_4 = c_3 - c_5$. Then, subtracting these two equations, we obtain $c_5 = ((q_1 - q_2) - (q_3 - q_4))/2$. Once we know c_5, we can derive the others.

In fact, this attack can also be used to obtain results *other than* numerical ones. Recall that we can apply logical rules to *and* ($\wedge$) and *or* ($\vee$), typical operators for database queries, to derive values from a series of logical expressions. For example, each expression might represent a query asking for precise data instead of counts, such as the equation

$$q = s_1 \vee s_2 \vee s_3 \vee s_4 \vee s_5$$

The result of the query is a set of records. Using logic and set algebra in a manner similar to our numerical example, we can carefully determine the actual values for each of the s_i.

Controls for Statistical Inference Attacks

Denning and Schlörer [DEN83a] present a very good survey of techniques for maintaining security in databases. The controls for all statistical attacks are similar. Essentially, there are two ways to protect against inference attacks: Either controls are applied to the queries or controls are applied to individual items within the database. As we have seen, it is difficult to determine whether a given query discloses sensitive data. Thus, query controls are effective primarily against direct attacks.

Suppression and concealing are two controls applied to data items. With **suppression,** sensitive data values are not provided; the query is rejected without response. With **concealing,** the answer provided is *close to* but not exactly the actual value.

These two controls reflect the contrast between security and precision. With suppression, any results provided are correct, yet many responses must be withheld to maintain security. With concealing, more results can be provided, but the precision of the results is lower. The choice between suppression and concealing depends on the context of the database. Examples of suppression and concealing follow.

Limited Response Suppression

The *n*-item *k*-percent rule eliminates certain low-frequency elements from being displayed. It is not sufficient to delete them, however, if their values can also be inferred. To see why, consider Table 6-11, which shows counts of students by dorm and sex.

Table 6-11. Students by Dorm and Sex.

	Holmes	Grey	West	Total
M	1	3	1	5
F	2	1	3	6
Total	3	4	4	11

The data in this table suggest that the cells with counts of 1 should be suppressed; their counts are too revealing. But it does no good to suppress the MaleHolmes cell when the value 1 can be determined by subtracting FemaleHolmes (2) from the total (3) to determine 1, as shown in Table 6-12.

Table 6-12. Students by Dorm and Sex, with Low Count Suppression.

	Holmes	Grey	West	Total
M	-	3	-	5
F	2	-	3	6
Total	3	4	4	11

When one cell is suppressed in a table with totals for rows and columns, it is necessary to suppress at least one additional cell on the row and one on the column to provide some confusion. Using this logic, all cells (except totals) would have to be suppressed in this small sample table. When totals are not provided, single cells in a row or column can be suppressed.

Combined Results

Another control combines rows or columns to protect sensitive values. For example, Table 6-13 shows several sensitive results that identify single individuals. (Even though these counts may not seem sensitive, they can be used to infer sensitive data such as NAME; therefore, we consider them to be sensitive.)

Table 6-13. Students by Sex and Drug Use.

	Drug Use			
Sex	**0**	**1**	**2**	**3**
M	1	1	1	2
F	2	2	2	0

These counts, combined with other results such as sum, permit us to infer individual drug-use values for the three males, as well as to infer that no female was rated 3 for drug use. To suppress such sensitive information, it is possible to combine the attribute values for 0 and 1, and also for 2 and 3, producing the less sensitive results shown in Table 6-14. In this instance, it is impossible to identify any single value.

Table 6-14. Suppression by Combining Revealing Values.

	Drug Use	
Sex	**0 or 1**	**2 or 3**
M	2	3
F	4	2

Another way of combining results is to present values in ranges. For example, instead of releasing exact financial aid figures, results can be released for the ranges $01999,

$20003999, and $4000 and above. Even if only one record is represented by a single result, the exact value of that record is not known. Similarly, the highest and lowest financial aid values are concealed.

Yet another method of combining is by rounding. This technique is actually a fairly well-known example of combining by range. If numbers are rounded to the nearest multiple of 10, the effective ranges are 05, 615, 1625, and so on. Actual values are rounded up or down to the nearest multiple of some base.

Random Sample

With random sample control, a result is not derived from the whole database; instead the result is computed on a random sample of the database. The sample chosen is large enough to be valid. Because the sample is not the whole database, a query against this sample will not necessarily match the result for the whole database. Thus, a result of 5 percent for a particular query means that 5 percent of the records chosen for the sample for this query had the desired property. You would expect that approximately 5 percent of the entire database will have the property in question, but the actual percentage may be quite different.

So that averaging attacks from repeated, equivalent queries are prevented, the same sample set should be chosen for equivalent queries. In this way, all equivalent queries will produce the same result, although that result will be only an approximation for the entire database.

Random Data Perturbation

It is sometimes useful to perturb the values of the database by a small error. For each x_i that is the true value of data item i in the database, we can generate a small random error term ε_i and add it to x_i for statistical results. The ε values are both positive and negative, so that some reported values will be slightly higher than their true values and other reported values will be lower. Statistical measures such as sum and mean will be close but not necessarily exact. Data perturbation is easier to use than random sample selection because it is easier to store all the ε values in order to produce the same result for equivalent queries.

Query Analysis

A more complex form of security uses query analysis. Here, a query and its implications are analyzed to determine whether a result should be provided. As noted earlier, query analysis can be quite difficult. One approach involves maintaining a query history for each user and judging a query in the context of what inferences are possible given previous results.

Conclusion on the Inference Problem

There are no perfect solutions to the inference problem. The approaches to controlling it follow the three paths listed below. The first two methods can be used either to limit queries accepted or to limit data provided in response to a query. The last method applies only to data released.

- *Suppress obviously sensitive information.* This action can be taken fairly easily. The tendency is to err on the side of suppression, thereby restricting the usefulness of the database.

- *Track what the user knows.* Although possibly leading to the greatest safe disclosure, this approach is extremely costly. Information must be maintained on all users, even though most are not trying to obtain sensitive data. Moreover, this approach seldom takes into account what any two people may know together and cannot address what a single user can accomplish by using multiple IDs.

- *Disguise the data.* Random perturbation and rounding can inhibit statistical attacks that depend on exact values for logical and algebraic manipulation. The users of the database receive slightly incorrect or possibly inconsistent results.

It is unlikely that research will reveal a simple, easy-to-apply measure that determines exactly which data can be revealed without compromising sensitive data.

Nevertheless, an effective control for the inference problem is just knowing that it exists. As with other problems in security, recognition of the problem leads to understanding of the purposes of controlling the problem and to sensitivity to the potential difficulties caused by the problem. However, just knowing of possible database attacks does not necessarily mean people will protect against those attacks, as explained in Sidebar 6-4. It is also noteworthy that much of the research on database inference was done in the early 1980s, but this proposal appeared almost two decades later.

Aggregation

Related to the inference problem is **aggregation**, which means building sensitive results from less sensitive inputs. We saw earlier that knowing either the latitude or longitude of a gold mine does you no good. But if you know both latitude and longitude, you can pinpoint the mine. For a more realistic example, consider how police use aggregation frequently in solving crimes: They determine who had a motive for committing the crime, when the crime was committed, who had alibis covering that time, who had the skills, and so forth. Typically, you think of police investigation as starting with the entire population and narrowing the analysis to a single person. But if the police officers work in parallel, one may have a list of possible suspects, another may have a list with possible motive, and another may have a list of capable persons. When the intersection of these lists is a single person, the police have their prime suspect.

Addressing the aggregation problem is difficult because it requires the database management system to track which results each user had already received and conceal any result that would let the user derive a more sensitive result. Aggregation is especially difficult to counter because it can take place outside the system. For example, suppose the security policy is that anyone can have *either* the latitude or longitude of the mine, but not both. Nothing prevents you from getting one, your friend from getting the other, and the two of you talking to each other.

Recent interest in data mining has raised concern again about aggregation. **Data mining** is the process of sifting through multiple databases and correlating multiple data elements to find useful information. Marketing companies use data mining extensively to find consumers likely to buy a product. As Sidebar 6-5 points out, it is not only marketers who are interested in aggregation through data mining.

Sidebar 6-4: Iceland Protects Privacy Against Inference

In 1998, Iceland authorized the building of a database of citizens' medical records, genealogy, and genetic information. Ostensibly, this database would provide data on genetic diseases to researchersmedical professionals and drug companies. Iceland is especially interesting for genetic disease research because the gene pool has remained stable for a long time; few outsiders have moved to Iceland, and few Icelanders have emigrated. For privacy, all identifying names or numbers would be replaced by a unique pseudonym. The Iceland health department asked computer security expert Ross Anderson to analyze the security aspects of this approach.

Anderson found several flaws with the proposed approach [AND98a]:

- Inclusion in the genealogical database complicates the task of maintaining individuals' anonymity because of distinctive family features. Moreover, parts of the genealogical database are already public because information about individuals is published in their birth and death records. It would be rather easy to identify someone in a family of three children born, respectively, in 1910, 1911, and 1929.

- Even a life's history of medical events may identify an individual. Many people would know that a person broke her leg skiing one winter and contracted a skin disease the following summer.
- Even small sample set restrictions on queries would fail to protect against algebraic attacks.
- To analyze the genetic data, which by its nature is necessarily of very fine detail, researchers would require the ability to make complex and specific queries. This same powerful query capability could lead to arbitrary selection of combinations of results.

For these reasons (and others), Anderson recommended against continuing to develop the public database. In spite of these problems, the Iceland Parliament voted to proceed with its construction and public release [JON00].

Aggregation was of interest to database security researchers at the same time as was inference. As we have seen, some approaches to inference have proven useful and are currently being used. But there have been few proposals for countering aggregation.

PREV NEXT

6.6. Multilevel Databases

So far, we have considered data in only two categories: either sensitive or nonsensitive. We have alluded to some data items being more sensitive than others, but we have allowed only yes-or-no access. Our presentation may have implied that sensitivity was a function of the *attribute,* the column in which the data appeared, although nothing we have done depended on this interpretation of sensitivity. Such a model appears in Table 6-15, where two columns are identified (by shading) as sensitive. In fact, though, sensitivity is determined not just by attribute but also in ways that we investigate in the next section.

Table 6-15. Attribute-Level Sensitivity. (Sensitive attributes are shaded.)

Name	Department	Salary	Phone	Performance
Rogers	training	43,800	4-5067	A2
Jenkins	research	62,900	6-4281	D4
Poling	training	38,200	4-4501	B1
Garland	user services	54,600	6-6600	A4
Hilten	user services	44,500	4-5351	B1
Davis	administration	51,400	4-9505	A3

Sidebar 6-5: Who Wrote Shakespeare's Plays?

Most people would answer "Shakespeare" when asked who wrote any of the plays attributed to the bard. But for 150 years literary scholars have had their doubts. In 1852, it was suggested that Edward de Vere, Earl of Oxford, wrote at least some of the works. For decades scholarly debate raged, citing what was known of Shakespeare's education, travels, work schedule, and the few other facts known about him.

In the 1980s a new analytic technique was developed: computerized analysis of text. Different researchers studied qualities such as word choice, images used in different plays, word pairs, sentence structure, and the likeany structural element that could show similarity or dissimilarity. (See, for example, [FAR96a] and [KAR01] , as well as www.shakespearefellowship.org.) The debate continues as researchers develop more and more qualities to correlate among databases (the language of the plays and other works attributed to Shakespeare). The debate will probably never be settled.

But the technique has proven useful. In 1996, an author called Anonymous published the novel *Primary Colors.* Many people tried to determine who the author was. But Donald Foster, a professor at Vassar College, aided by some simple computer tools, attributed the novel to Joe Klein, who later admitted being the author. Neumann [NEU96] in the Risks forum, notes how hard it is to lie convincingly, even having tried to alter your writing style, given "telephone

records, credit-card records, airplane reservation databases, library records, snoopy neighbors, coincidental encounters, etc."in short, given aggregation.

The approach has uses outside the literary field. In 2002 the SAS Institute, vendors of statistical analysis software, introduced data mining software intended to find patterns in old e-mail messages and other masses of text. The company suggests the tool might be useful in identifying and blocking spam. Another possible use is detecting lies, or perhaps just flagging potential inconsistencies. It could also help locate the author of malicious code.

The Case for Differentiated Security

Consider a database containing data on U.S. government expenditures. Some of the expenditures are for paper clips, which is not sensitive information. Some salary expenditures are subject to privacy requirements. Individual salaries are sensitive, but the aggregate (for example, the total Agriculture Department payroll, which is a matter of public record) is not sensitive. Expenses of certain military operations are more sensitive; for example, the total amount the United States spends for ballistic missiles, which is not public. There are even operations known only to a few people, and so the amount spent on these operations, or even the fact that anything was spent on such an operation, is highly sensitive.

Table 6-15 lists employee information. It may in fact be the case that Davis is a temporary employee hired for a special project, and her whole record has a different sensitivity from the others. Perhaps the phone shown for Garland is her private line, not available to the public. We can refine the sensitivity of the data by depicting it as shown in Table 6-16.

Table 6-16. Data and Attribute Sensitivity.

Name	Department	Salary	Phone	Performance
Rogers	training	43,800	4-5067	A2
Jenkins	research	62,900	6-4281	D4
Poling	training	38,200	4-4501	B1
Garland	user services	54,600	6-6600	A4
Hilten	user services	44,500	4-5351	B1
Davis	administration	51,400	4-9505	A3

From this description, three characteristics of database security emerge.

- The security of a single element may be different from the security of other elements of the same record or from other values of the same attribute. That is, the security of one element may differ from that of other elements of the same row or column. This situation implies that security should be implemented for each individual element.
- Two levelssensitive and nonsensitiveare inadequate to represent some security situations. Several grades of security may be needed. These grades may represent ranges of allowable knowledge, which may overlap. Typically, the security grades form

a lattice.

- The security of an aggregatea sum, a count, or a group of values in a databasemay differ from the security of the individual elements. The security of the aggregate may be higher or lower than that of the individual elements.

These three principles lead to a model of security not unlike the military model of security encountered in Chapter 5, in which the sensitivity of an object is defined as one of *n* levels and is further separated into compartments by category.

Granularity

Recall that the military classification model applied originally to paper documents and was adapted to computers. It is fairly easy to classify and track a single sheet of paper or, for that matter, a paper file, a computer file, or a single program or process. It is entirely different to classify individual data items.

For obvious reasons, an entire sheet of paper is classified at one level, even though certain words, such as *and*, *the*, or *of*, would be innocuous in any context, and other words, such as codewords like *Manhattan project*, might be sensitive in any context. But defining the sensitivity of each value in a database is similar to applying a sensitivity level to each individual word of a document.

And the problem is still more complicated. The word *Manhattan* by itself is not sensitive, nor is *project*. However, the combination of these words produces the sensitive codeword *Manhattan project*. A similar situation occurs in databases. Therefore, not only can every *element* of a database have a distinct sensitivity, every *combination of elements* can also have a distinct sensitivity. Furthermore, the combination can be more or less sensitive than any of its elements.

So what would we need in order to associate a sensitivity level with each value of a database? First, we need an access control policy to dictate which users may have access to what data. Typically, to implement this policy each data item is marked to show its access limitations. Second, we need a means to guarantee that the value has not been changed by an unauthorized person. These two requirements address both confidentiality and integrity.

Security Issues

In Chapter 1, we introduced three general security concerns: integrity, confidentiality, and availability. In this section, we extend the first two of these concepts to include their special roles for multilevel databases.

Integrity

Even in a single-level database in which all elements have the same degree of sensitivity, integrity is a tricky problem. In the case of multilevel databases, integrity becomes both more important and more difficult to achieve. Because of the *-property for access control, a process that reads high-level data is not allowed to write a file at a lower level. Applied to databases, however, this principle says that a high-level user should not be able to write a lower-level data element.

The problem with this interpretation arises when the DBMS must be able to read all records in the database and write new records for any of the following purposes: to do backups, to scan the database to answer queries, to reorganize the database according to a user's processing needs, or to update all records of the database.

When people encounter this problem, they handle it by using trust and common sense. People who have access to sensitive information are careful not to convey it to uncleared individuals. In a computing system, there are two choices: Either the process cleared at a high level cannot write to a lower level or the process must be a "trusted process," the computer

equivalent of a person with a security clearance.

Confidentiality

Users trust that a database will provide correct information, meaning that the data are consistent and accurate. As indicated earlier, some means of protecting confidentiality may result in small changes to the data. Although these perturbations should not affect statistical analyses, they may produce two different answers representing the same underlying data value in response to two differently formed queries. In the multilevel case, two different users operating at two different levels of security might get two different answers to the same query. To preserve confidentiality, precision is sacrificed.

Enforcing confidentiality also leads to unknowing redundancy. Suppose a personnel specialist works at one level of access permission. The specialist knows that Bob Hill works for the company. However, Bob's record does not appear on the retirement payment roster. The specialist assumes this omission is an error and creates a record for Bob.

The reason that no record for Bob appears is that Bob is a secret agent, and his employment with the company is not supposed to be public knowledge. A record on Bob actually is in the file but, because of his special position, his record is not accessible to the personnel specialist. The DBMS cannot reject the record from the personnel specialist because doing so would reveal that there already is such a record at a sensitivity too high for the specialist to see. The creation of the new record means that there are now two records for Bob Hill: one sensitive and one not, as shown in Table 6-17. This situation is called **polyinstantiation,** meaning that one record can appear (be instantiated) many times, with a different level of confidentiality each time.

Table 6-17. Polyinstantiated Records.

Name	Sensitivity	Assignment	Location
Hill, Bob	C	Program Mgr	London
Hill, Bob	TS	Secret Agent	South Bend

This problem is exacerbated because Bob Hill is a common enough name that there might be two different people in the database with that name. Thus, merely scanning the database (from a high-sensitivity level) for duplicate names is not a satisfactory way to find records entered unknowingly by people with only low clearances.

We might also find other reasons, unrelated to sensitivity level, that result in polyinstantiation. For example, Mark Thyme worked for Acme Corporation for 30 years and retired. He is now drawing a pension from Acme, so he appears as a retiree in one personnel record. But Mark tires of being home and is rehired as a part-time contractor; this new work generates a second personnel record for Mark. Each is a legitimate employment record. In our zeal to reduce polyinstantiation, we must be careful not to eliminate legitimate records such as these.

PREV NEXT

6.7. Proposals for Multilevel Security

As you can already tell, implementing multilevel security for databases is difficult, probably more so than in operating systems, because of the small granularity of the items being controlled. In the remainder of this section, we study approaches to multilevel security for databases.

Separation

As we have already seen, separation is necessary to limit access. In this section, we study mechanisms to implement separation in databases. Then, we see how these mechanisms can help to implement multilevel security for databases.

Partitioning

The obvious control for multilevel databases is partitioning. The database is divided into separate databases, each at its own level of sensitivity. This approach is similar to maintaining separate files in separate file cabinets.

This control destroys a basic advantage of databases: elimination of redundancy and improved accuracy through having only one field to update. Furthermore, it does not address the problem of a high-level user who needs access to some low-level data combined with high-level data.

Nevertheless, because of the difficulty of establishing, maintaining, and using multilevel databases, many users with data of mixed sensitivities handle their data by using separate, isolated databases.

Encryption

If sensitive data are encrypted, a user who accidentally receives them cannot interpret the data. Thus, each level of sensitive data can be stored in a table encrypted under a key unique to the level of sensitivity. But encryption has certain disadvantages.

First, a user can mount a chosen plaintext attack. Suppose party affiliation of REP or DEM is stored in encrypted form in each record. A user who achieves access to these encrypted fields can easily decrypt them by creating a new record with party=DEM and comparing the resulting encrypted version to that element in all other records. Worse, if authentication data are encrypted, the malicious user can substitute the encrypted form of his or her own data for that of any other user. Not only does this provide access for the malicious user, but it also excludes the legitimate user whose authentication data have been changed to that of the malicious user. These possibilities are shown in Figures 6-5 and 6-6.

Figure 6-5. Cryptographic Separation: Different Encryption Keys.

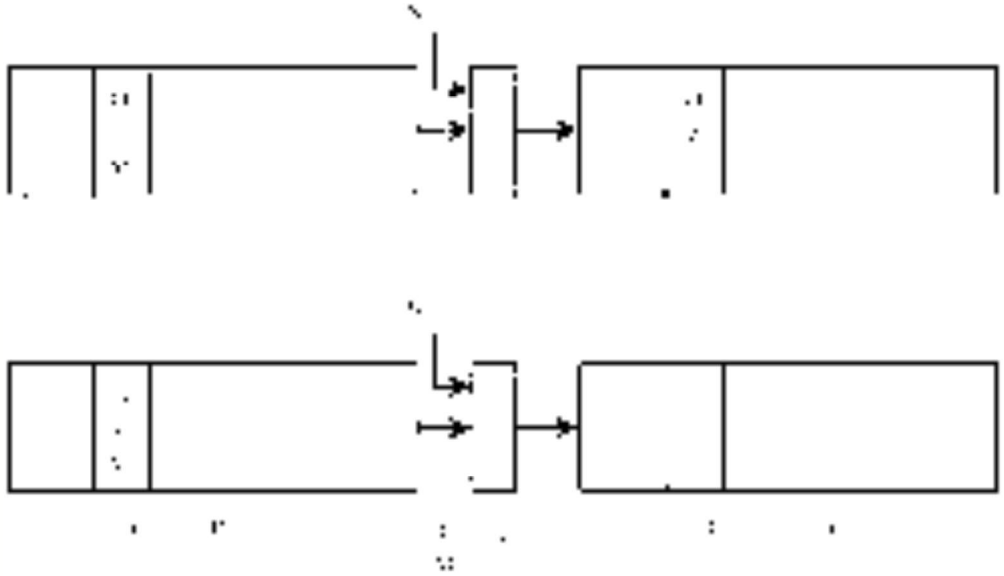
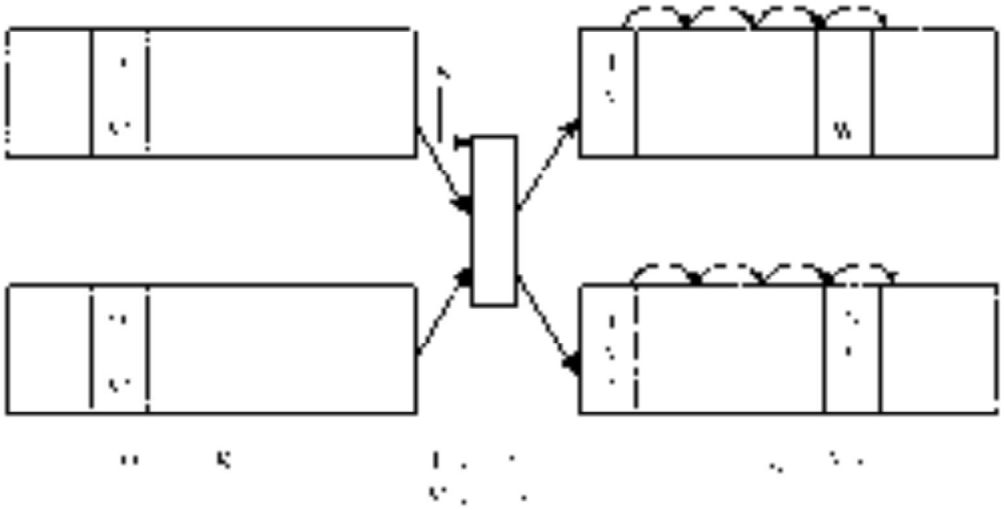


Figure 6-6. Cryptographic Separation: Block Chaining.



Using a different encryption key for each record overcomes these defects. Each record's fields can be encrypted with a different key, or all fields of a record can be cryptographically linked, as with cipher block chaining.

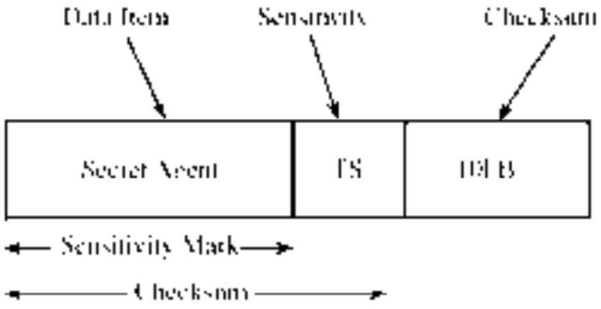
The disadvantage, then, is that each field must be decrypted when users perform standard database operations such as "select all records with SALARY > 10,000." Decrypting the SALARY field, even on rejected records, increases the time to process a query. (Consider the query that selects just one record but that must decrypt and compare one field of each record to find the one that satisfies the query.) Thus, encryption is not often used to implement separation in databases.

Integrity Lock

The **integrity lock** was first proposed at the U.S. Air Force Summer Study on Data Base Security [AFS83]. The lock is a way to provide both integrity and limited access for a database. The operation was nicknamed "spray paint" because each element is figuratively painted with a color that denotes its sensitivity. The coloring is maintained with the element, not in a master database table.

A model of the basic integrity lock is shown in Figure 6-7. As illustrated, each apparent data item consists of three pieces: the actual data item itself, a sensitivity label, and a checksum. The sensitivity label defines the sensitivity of the data, and the checksum is computed across both data and sensitivity label to prevent unauthorized modification of the data item or its label. The actual data item is stored in plaintext, for efficiency because the DBMS may need to examine many fields when selecting records to match a query.

Figure 6-7. Integrity Lock.

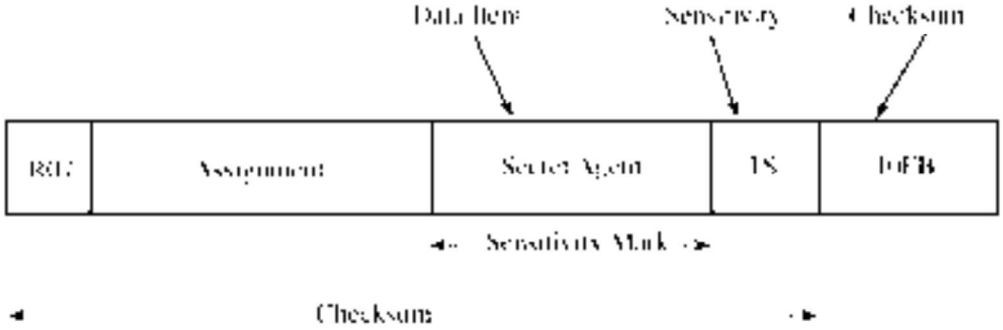


The sensitivity label should be

- *unforgeable,* so that a malicious subject cannot create a new sensitivity level for an element
- *unique,* so that a malicious subject cannot copy a sensitivity level from another element
- *concealed,* so that a malicious subject cannot even determine the sensitivity level of an arbitrary element

The third piece of the integrity lock for a field is an error-detecting code, called a **cryptographic checksum**. To guarantee that a data value or its sensitivity classification has not been changed, this checksum must be unique for a given element, and must contain both the element's data value and something to tie that value to a particular position in the database. As shown in Figure 6-8, an appropriate cryptographic checksum includes something unique to the record (the record number), something unique to this data field within the record (the field attribute name), the value of this element, and the sensitivity classification of the element. These four components guard against anyone's changing, copying, or moving the data. The checksum can be computed with a strong encryption algorithm or hash function.

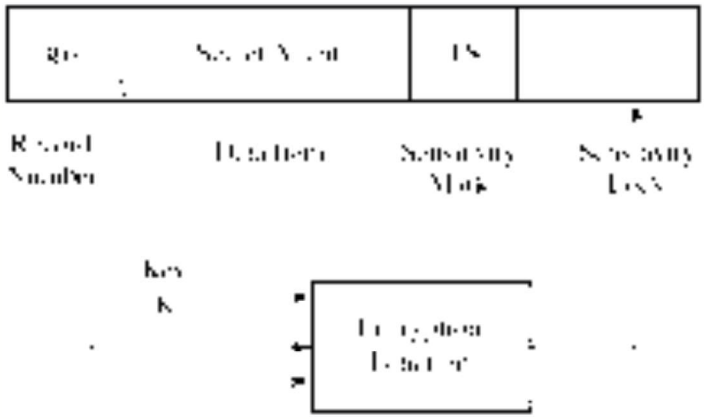
Figure 6-8. Cryptographic Checksum.



Sensitivity Lock

The sensitivity lock shown in Figure 6-9 was designed by Graubert and Kramer [GRA84b] to meet these principles. A **sensitivity lock** is a combination of a unique identifier (such as the record number) and the sensitivity level. Because the identifier is unique, each lock relates to one particular record. Many different elements will have the same sensitivity level. A malicious subject should not be able to identify two elements having identical sensitivity levels or identical data values just by looking at the sensitivity level portion of the lock. Because of the encryption, the lock's contents, especially the sensitivity level, are concealed from plain view. Thus, the lock is associated with one specific record, and it protects the secrecy of the sensitivity level of that record.

Figure 6-9. Sensitivity Lock.



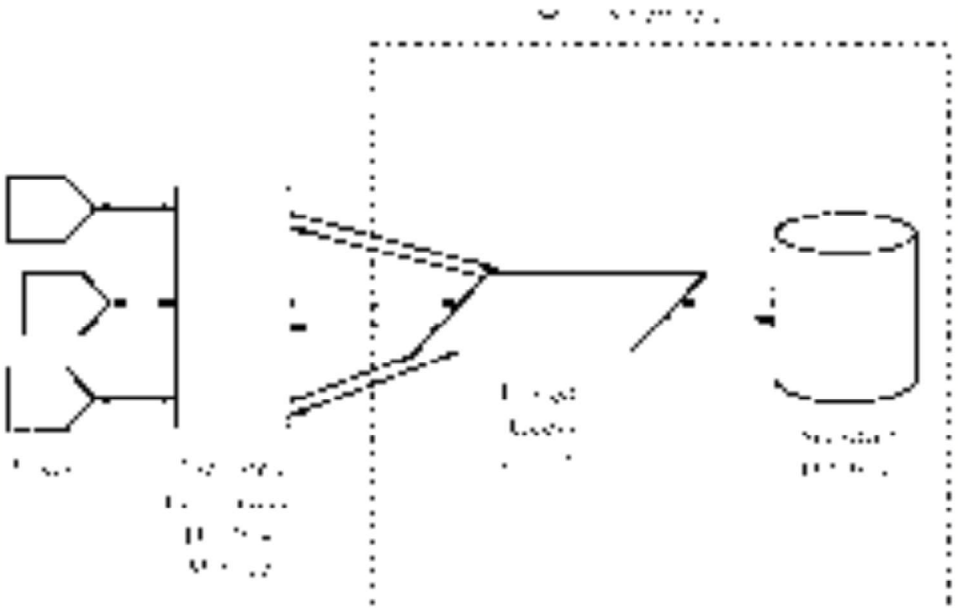
Designs of Multilevel Secure Databases

This section covers different designs for multilevel secure databases. These designs show the tradeoffs among efficiency, flexibility, simplicity, and trustworthiness.

Integrity Lock

The integrity lock DBMS was invented as a short-term solution to the security problem for multilevel databases. The intention was to be able to use any (untrusted) database manager with a trusted procedure that handles access control. The sensitive data were obliterated or concealed with encryption that protected both a data item and its sensitivity. In this way, only the access procedure would need to be trusted because only it would be able to achieve or grant access to sensitive data. The structure of such a system is shown in Figure 6-10.

Figure 6-10. Trusted Database Manager.



The efficiency of integrity locks is a serious drawback. The space needed for storing an element must be expanded to contain the sensitivity label. Because there are several pieces in the label and one label for every element, the space required is significant.

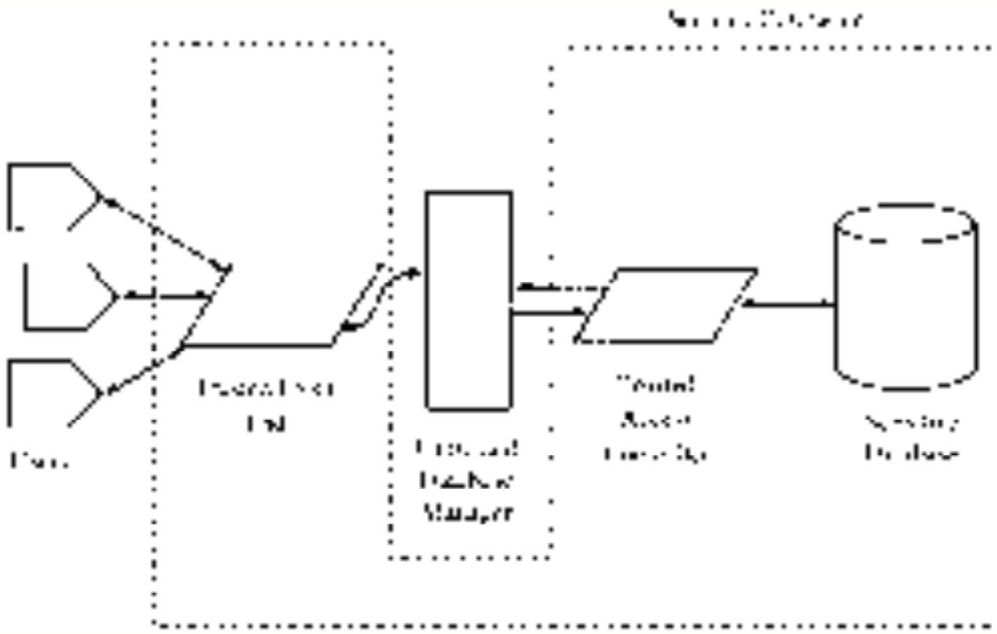
Problematic, too, is the processing time efficiency of an integrity lock. The sensitivity label must be decoded every time a data element is passed to the user to verify that the user's access is allowable. Also, each time a value is written or modified, the label must be recomputed. Thus, substantial processing time is consumed. If the database file can be sufficiently protected, the data values of the individual elements can be left in plaintext. That approach benefits select and project queries across sensitive fields because an element need not be decrypted just to determine whether it should be selected.

A final difficulty with this approach is that the untrusted database manager sees all data, so it is subject to Trojan horse attacks by which data can be leaked through covert channels.

Trusted Front End

The model of a **trusted front-end** process is shown in Figure 6-11. A trusted front end is also known as a **guard** and operates much like the reference monitor of Chapter 5. This approach, originated by Hinke and Schaefer [HIN75], recognizes that many DBMSs have been built and put into use without consideration of multilevel security. Staff members are already trained in using these DBMSs, and they may in fact use them frequently. The front-end concept takes advantage of existing tools and expertise, enhancing the security of these existing systems with minimal change to the system. The interaction between a user, a trusted front end, and a DBMS involves the following steps.

Figure 6-11. Trusted Front End.



1. A user identifies himself or herself to the front end; the front end authenticates the user's identity.

2. The user issues a query to the front end.

3. The front end verifies the user's authorization to data.

4. The front end issues a query to the database manager.

5. The database manager performs I/O access, interacting with low-level access control to achieve access to actual data.

6. The database manager returns the result of the query to the trusted front end.

7. The front end analyzes the sensitivity levels of the data items in the result and selects those items consistent with the user's security level.

8. The front end transmits selected data to the untrusted front end for formatting.

9. The untrusted front end transmits formatted data to the user.

The trusted front end serves as a one-way filter, screening out results the user should not be able to access. But the scheme is inefficient because potentially much data is retrieved and then discarded as inappropriate for the user.

Commutative Filters

The notion of a commutative filter was proposed by Denning [DEN85] as a simplification of the trusted interface to the DBMS. Essentially, the filter screens the user's request, reformatting it if necessary, so that only data of an appropriate sensitivity level are returned to the user.

A **commutative filter** is a process that forms an interface between the user and a DBMS. However, unlike the trusted front end, the filter tries to capitalize on the efficiency of most DBMSs. The filter reformats the query so that the database manager does as much of the

work as possible, screening out many unacceptable records. The filter then provides a second screening to select only data to which the user has access.

Filters can be used for security at the record, attribute, or element level.

- When used at the record level, the filter requests desired data plus cryptographic checksum information; it then verifies the accuracy and accessibility of data to be passed to the user.
- At the attribute level, the filter checks whether all attributes in the user's query are accessible to the user and, if so, passes the query to the database manager. On return, it deletes all fields to which the user has no access rights.
- At the element level, the system requests desired data plus cryptographic checksum information. When these are returned, it checks the classification level of every element of every record retrieved against the user's level.

Suppose a group of physicists in Washington works on very sensitive projects, so the current user should not be allowed to access the physicists' names in the database. This restriction presents a problem with this query:

```
retrieve NAME where ((OCCUP=PHYSICIST) ∧ (CITY=WASHDC))
```

Suppose, too, that the current user is prohibited from knowing anything about any people in Moscow. Using a conventional DBMS, the query might access all records, and the DBMS would then pass the results on to the user. However, as we have seen, the user might be able to infer things about Moscow employees or Washington physicists working on secret projects without even accessing those fields directly.

The commutative filter re-forms the original query in a trustable way so that sensitive information is never extracted from the database. Our sample query would become

```
retrieve NAME where ((OCCUP=PHYSICIST) ∧ (CITY=WASHDC))
from all records R where
      (NAME-SECRECY-LEVEL (R) ≤ USER-SECRECY-LEVEL) ∧
      (OCCUP-SECRECY-LEVEL (R) ≤ USER-SECRECY-LEVEL) ∧
      (CITY-SECRECY-LEVEL (R) ≤ USER-SECRECY-LEVEL))
```

The filter works by restricting the query to the DBMS and then restricting the results before they are returned to the user. In this instance, the filter would request NAME, NAME-SECRECY-LEVEL, OCCUP, OCCUP-SECRECY-LEVEL, CITY, and CITY-SECRECY-LEVEL values and would then filter and return to the user only those fields and items that are of a secrecy level acceptable for the user. Although even this simple query becomes complicated because of the added terms, these terms are all added by the front-end filter, invisible to the user.

An example of this query filtering in operation is shown in Figure 6-12. The advantage of the commutative filter is that it allows query selection, some optimization, and some subquery handling to be done by the DBMS. This delegation of duties keeps the size of the security filter small, reduces redundancy between it and the DBMS, and improves the overall efficiency of the system.

Figure 6-12. Commutative Filters.