# 4 Access Controls

Access controls ensure that all direct accesses to objects are authorized. By regulating the reading, changing, and deletion of data and programs, access controls protect against accidental and malicious threats to secrecy, authenticity, and system availability (see Chapter 1).

Many access controls incorporate a concept of **ownership**—that is, users may dispense and revoke privileges for objects they own. This is common in file systems intended for the long-term storage of user data sets and programs. Not all applications include this concept; for example, patients do not own their records in a medical information system.

The effectiveness of access controls rests on two premises. The first is proper user identification: no one should be able to acquire the access rights of another. This premise is met through authentication procedures at login as described in Chapter 3. The second premise is that information specifying the access rights of each user or program is protected from unauthorized modification. This premise is met by controlling access to system objects as well as to user objects.

In studying access controls, it is useful to separate policy and mechanism. An access control **policy** specifies the authorized accesses of a system; an access control **mechanism** implements or enforces the policy. This separation is useful for three reasons. First, it allows us to discuss the access requirements of systems independent of how these requirements may be implemented. Second, it allows us to compare and contrast different access control policies as well as different mechanisms that enforce the same policies. Third, it allows us to design mechanisms capable of enforcing a wide range of policies. These mechanisms can be integrated into the hardware features of the system without impinging on the flexibility of the system to adapt to different or changing policies.

## 4.1 ACCESS-MATRIX MODEL

The access-matrix model provides a framework for describing protection systems. The model was independently developed by researchers in both the operating systems area and the database area. The operating systems version of the model was formulated by Lampson [Lamp71] in conjunction with the COSINE Task Force on Operating Systems [DenP71a]. The model was subsequently refined by Graham and Denning [GrDe72,DenP71b]. Harrison, Ruzzo, and Ullman [Harr76] later developed a more formal version of the model as a framework for proving properties about protection systems. At about the same time the access matrix was introduced as a model of protection in operating systems, Conway, Maxwell, and Morgan [Conw72] of the ASAP project at Cornell University independently modeled protection in database systems with a security matrix.

The model is defined in terms of states and state transitions, where the state of a protection system is represented by a matrix, and the state transitions are described by commands.

### 4.1.1 The Protection State

The state of a system is defined by a triple $(S, O, A)$, where:

1. $S$ is a set of **subjects**, which are the active entities of the model. We will assume that subjects are also considered to be objects; thus $S \subseteq O$.
2. $O$ is a set of **objects**, which are the protected entities of the system. Each object is uniquely identified by a name.
3. $A$ is an **access matrix**, with rows corresponding to subjects and columns to objects. An entry $A[s, o]$ lists the access **rights** (or privileges) of subject $s$ for object $o$.

In operating systems, the objects typically include files, segments of memory, and processes (i.e., activations of programs). The subjects may be users, processes, or domains; a **domain** is a protection environment in which a process executes. A process may change domains during its execution.

The access rights specify the kinds of accesses that may be performed on different types of objects. The rights for segments and files usually include $R$ (read), $W$ (write), and $E$ (execute). (Some systems have an append right, which allows subjects to add data to the end of an object but not overwrite existing data.) Some rights may be **generic**, applying to more than one type of object; $R$, $W$, $E$, and *Own* (ownership) are examples. The number of generic rights is finite.

***Example:***
Figure 4.1 illustrates the state of a simple system having two processes (subjects), two memory segments, and two files. Each process has its own private segment and owns one file. Neither process can control the other process. ■

Previous Chapter Previous Page Home Next Page Next Chapter

FIGURE 4.1 Access matrix.

| | | Objects | | | | | |
|---|---|---|---|---|---|---|---|
| | | *M*1 | *M*2 | *F*1 | *F*2 | *P*1 | *P*2 |
| Subjects | *P*1 | *R* *W* *E* | | *Own* *R* *W* | | | |
| | *P*2 | | *R* *W* *E* | | *Own* *R* *E* | | |

Graham and Denning associate with each type of object a **monitor** that controls access to the object. The monitor for an object *o* prevents a subject *s* from accessing *o* when *A*[*s*, *o*] does not contain the requisite right. A monitor can be implemented in hardware, software, or some combination of hardware and software. An example of a monitor is the hardware that checks an address to determine if it is within the memory bounds associated with a process. Another example is the file system monitor that validates requests for file accesses.

Protection within a program is modeled with a more refined matrix in which the subjects are procedures (or activations of procedures), and the objects are data structures and procedures. The access rights for a data object are determined by the operations and procedures that may be applied to objects of that type. For example, the access rights for an **integer** variable consist of the arithmetic and relational operators (+, *, <, etc.) and assignment (:=); the rights for an **integer** constant (or variable passed by value) exclude assignment. Note that the right to apply the assignment operator is granted by *W*-access to an object, whereas the right to apply the remaining operators is granted by *R*-access to the object.

The access rights for a programmer-defined object of type stack may be restricted to *push* and *pop* procedures, so the program cannot manipulate the underlying representation of the stack (e.g., to delete an element from the middle of the stack). The *pop* and *push* procedures, however, have access to the representation of stacks.

The protection provided by language structures must be enforced by both the language processor and the run-time system. The language processor is responsible for screening out **specification errors** (e.g., violations of syntax, type, and scope rules). The run-time system is responsible for screening out **execution errors** (e.g., linkage errors, binding errors, addressing errors, or I/O errors). The system also provides protection in case of changes to compiled programs. If access checking is implemented in hardware, it can be performed in parallel with program execution, and need not degrade the performance of the system. The strongest protection comes from a combination of compiler and system support: the compiler prevents programs with detectable specification errors from executing; the system ensures that programs execute according to their specifications and to the current protection state.

In database systems, the subjects correspond to users and the objects to files, relations, records, or fields within records. Each entry $A[s, o]$ is a **decision rule**, specifying the conditions under which user $s$ may access data object $o$, and the operations that $s$ is permitted to perform on $o$.

The decision rules are a generalization of the concept of access rights. The rules may specify both **data-independent** conditions, which are analogous to the rights in operating systems, and **data-dependent** conditions, which are a function of the current values of the data being accessed. For example, permitting a user to alter the contents of only those student records for which the *Department* field is *Physics* is an example of a data-dependent condition. The concept of dependency may be extended to include **time-dependent** conditions, which are functions of the clock (e.g., a user may be permitted to access the payroll records between 9 and 5 only), **context-dependent** conditions, which are functions of combinations of data (e.g., a user may be permitted to see student grades or student names, but not pairs of names and grades), and **history-dependent** conditions, which are functions of previous states of the system (e.g., a process may not be allowed to write into an unclassified file if it previously processed classified data) [Hart76]. In general, a decision may depend on any available information describing the present or past state of the system.

The decision rules are similar to the control procedures described in Hoffman's [Hoff71] **formulary model**. In Hoffman's system, all security is enforced by a set of database access procedures called **formularies**. A formulary consists of control procedures that determine whether to grant an access request, addressing procedures that map logical names into virtual addresses, and encryption and decryption procedures. Hoffman's formularies are like the object monitors described by Graham and Denning.

### 4.1.2 State Transitions

Changes to the state of a system are modeled by a set of **commands**. Commands are specified by a sequence of primitive operations that change the access matrix. These operations are conditioned on the presence of certain rights in the access matrix and are controlled by a monitor responsible for managing the protection state. Harrison, Ruzzo, and Ullman identified six **primitive operations**:

**enter** $r$ into $A[s, o]$
**delete** $r$ from $A[s, o]$
**create subject** $s$
**create object** $o$
**destroy subject** $s$
**destroy object** $o$ .

Their effect on the access matrix is formally defined in Table 4.1. Let $op$ be a primitive operator, and let $Q = (S, O, A)$ be a system state. Then execution of $op$ in state $Q$ causes a **transition** from $Q$ to the state $Q' = (S', O', A')$ under the conditions defined in Table 4.1. This is written as

TABLE 4.1 Primitive operations.

| op | conditions | new state |
|---|---|---|
| **enter** $r$ into $A[s, o]$ | $s \in S$<br>$o \in O$ | $S' = S$<br>$O' = O$<br>$A'[s, o] = A[s, o] \cup \{r\}$<br>$A'[s_1, o_1] = A[s_1, o_1]$ $(s_1, o_1) \neq (s, o)$ |
| **delete** $r$ from $A[s, o]$ | $s \in S$<br>$o \in O$ | $S' = S$<br>$O' = O$<br>$A'[s, o] = A[s, o] - \{r\}$<br>$A'[s_1, o_1] = A[s_1, o_1]$ $(s_1, o_1) \neq (s, o)$ |
| **create subject** $s'$ | $s' \notin O$ | $S' = S \cup \{s'\}$<br>$O' = O \cup \{s'\}$<br>$A'[s, o] = A[s, o]$ $s \in S, o \in O$<br>$A'[s', o] = \emptyset, o \in O'$<br>$A'[s, s'] = \emptyset, s \in S'$ |
| **create object** $o'$ | $o' \notin O$ | $S' = S$<br>$O' = O \cup \{o'\}$<br>$A'[s, o] = A[s, o]$ $s \in S, o \in O$<br>$A'[s, o'] = \emptyset, s \in S'$ |
| **destroy subject** $s'$ | $s' \in S$ | $S' = S - \{s'\}$<br>$O' = O - \{s'\}$<br>$A'[s, o] = A[s, o]$ $s \in S', o \in O'$ |
| **destroy object** $o'$ | $o' \in O$<br>$o' \notin S$ | $S' = S$<br>$O' = O - \{o'\}$<br>$A'[s, o] = A[s, o]$ $s \in S', o \in O'$ |

$$Q \vdash_{op} Q'$$

(read "$Q$ derives $Q'$ under $op$") .

Harrison, Ruzzo, and Ullman consider commands of the following form:

```
command $c(x_1, \ldots, x_k)$
  if $r_1$ in $A[x_{s1}, x_{o1}]$ and
     $r_2$ in $A[x_{s2}, x_{o2}]$ and
        .
        .
        .
     $r_m$ in $A[x_{sm}, x_{om}]$
  then
     $op_1$;
     $op_2$;
```

```
        .
        .
        .
        $op_n$
    end ,
```

where $r_1, \ldots, r_m$ are rights, and $s1, \ldots, sm$ and $o1, \ldots, om$ are integers between 1 and $k$. A command may have an empty set of conditions (i.e., $m = 0$). We assume, however, that each command performs at least one operation.

The effect of a command on the state of a protection system is as follows. Let $c(a_1, \ldots, a_k)$ be a command with actual parameters $a_1, \ldots, a_k$, and let $Q = (S, O, A)$ be a state of a protection system. Then $Q$ yields state $Q'$ under $c$, written

$$Q \vdash_{c(a_1, \ldots, a_k)} Q' ,$$

provided

1. $Q' = Q$ if one of the conditions of $c$ is not satisfied, and
2. $Q' = Q_n$ otherwise, where there exist states $Q_0, Q_1, \ldots, Q_n$ such that

   $$Q = Q_0 \vdash_{op_1^*} Q_1 \vdash_{op_2^*} \ldots \vdash_{op_n^*} Q_n ,$$

   where $op_i^*$ denotes the primitive operation $op_i$, substituting the actual parameters $a_i$ for the formal parameters $x_i$.

We shall write $Q \vdash_c Q'$ if there exist actual parameters $a_1, \ldots, a_k$ such that $Q \vdash_{c(a_1, \ldots, a_k)} Q'$; and $Q \vdash Q'$ if there exists a command $c$ such that $Q \vdash_c Q'$. Finally, we shall write $Q \vdash^* Q'$ if there exists a sequence of length $n \geq 0$: $Q = Q_0 \vdash Q_1 \vdash \ldots \vdash Q_n = Q'$.

The access-matrix model is an abstract representation of the protection policies and mechanisms found in real systems. As such, it provides a conceptual aid to understanding and describing protection systems, a common framework for comparing different protection systems, and a formal model for studying the inherent properties of protection systems. Some mechanisms are more easily described by other models, however. We shall later use graphs to describe mechanisms that involve the transfer of rights.

Protection systems should not be implemented using the matrix and associated commands. One reason is there are much better ways of representing the access state of a system than with a matrix that is likely to be large and sparse. Another is that the commands are too primitive for most applications.

***Example:***

We shall show how the model can be used to describe the transfer of rights in operating systems with processes, segments, and owned files, as illustrated by Figure 4.1. Our example is similar to the one given by Harrison, Ruzzo, and Ullman.

Any process may create a new file. The process creating a file is automatically given ownership of the file and *RW*-access to the file. This is represented by the command:

```
command create.file(p, f)
  create object f;
  enter Own into A[p, f];
  enter R into A[p, f];
  enter W into A[p, f]
end .
```

The process owning a file may change its rights to the file. For example, a process $p$ owning a file $f$ can protect $f$ from inadvertent modification by removing its $W$-access to $f$. It may also confer any right to the file (except ownership) on other processes. For example, $R$-access may be conferred by process $p$ on process $q$ with the command:

```
command confer.read(p, q, f)
  if Own in A[p, f]
  then enter R into A[q, f]
end .
```

(Similar commands confer $W$- and $E$-access.)

The preceding command states that the owner of an object can grant a right to the object it does not have. In particular, it can grant this right to itself. This allows it to revoke its $W$-access to an object, and later restore the right to modify the object. This violates the principle of **attenuation of privilege**, which states that a process can never increase its rights, or transfer rights it does not have. We shall apply this principle only to nonowners, as is done in most systems.

Removal of access rights is a subject of much controversy. Here we shall assume that the owner of an object may revoke access rights to the object at any time. Commands for removing access rights from the access matrix are similar to those for conferring rights; for example, process $p$ may revoke $R$-access from process $q$ with the command:

```
command revoke.read(p, q, f)
  if Own in A[p, f]
  then delete R from A[q, f]
end .
```

Some systems permit subjects to transfer an access right $r$ to an unowned object. This is modeled with a **copy flag** (denoted by an asterisk placed after $r$). Following the principle of attenuation of privilege, a process may transfer any access right it holds for an object provided the copy flag of the attribute is set. The following command transfers $R$-access from process $p$ to $q$, but does not give $q$ the ability to transfer the right further:

```
command transfer.read(p, q, f)
  if R* in A[p, f]
  then enter R into A[q, f]
end .
```

Alternatively, a process may be able to transfer an access right, but forfeit the right in so doing. This is modeled with a **transfer-only flag** (denoted by +). The following command transfers *R*-access under these conditions:

```
command transfer-only.read (p, q, f)
  if R+ in A[p, f]
  then delete R+ from A[p, f];
       enter R+ into A[q, f]
end .
```

The owner of an object would be able to confer rights with or without either of the flags.

Some systems permit a process to spawn subordinate processes and control the access rights of its subordinates. A process *p* can create a subordinate process *q* having memory segment *m* with the command:

```
command create.subordinate(p, q, m)
  create subject q;
  create object m;
  enter Ctrl into A[p, q]
  enter R into A[q, m]
  enter W into A[q, m]
  enter E into A[q, m]
end .
```

Process *p* is given the *Ctrl* right to *q*, allowing it to take or revoke any of *q*'s rights, including those conferred on *q* by other processes. The following command gives *p* *R*-access to *q*'s memory segment:

```
command take.subordinate.read(p, q, m)
  if Ctrl in A[p, q] and
      R in A[q, m]
  then enter R in A[p, m]
end .
```

The following command revokes *q*'s *R*-access to file *f*:

```
command revoke.subordinate.read(p, q, f)
  if Ctrl in A[p, q]
  then delete R from A[q, f]
end .
```

If our command format permitted disjunctions, we could combine *revoke.subordinate.read* with *revoke.read:*

```
command revoke.read(p, q, f)
  if Own in A[p, f] or
     Ctrl in A[p, q]
  then delete R from A[q, f]
end .
```

Previous Chapter Previous Page Home Next Page Next Chapter

FIGURE 4.2 Access matrix after command sequence.

| | | Objects | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | *M*1 | *M*2 | *M*3 | *F*1 | *F*2 | *P*1 | *P*2 | *P*3 |
| | *P*1 | *R* *W* *E* | | | *Own* *R* *W* | | | | |
| Subjects | *P*2 | | *R* *W* *E* | *R* *W* | | *Own* *R* *W* | | | *Ctrl* |
| | *P*3 | | | *R* *W* *E* | | *R* | | | |

Because any command with disjunctions is equivalent to a list of separate commands, we have restricted commands to the simpler format.

Figure 4.2 shows the effect of executing the following commands on the initial state shown in Figure 4.1:

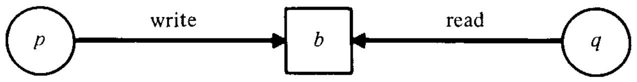
*create.subordinate*(*P*2, *P*3, *M*3)
*take.subordinate.read*(*P*2, *P*3, *M*3)
*take.subordinate.write*(*P*2, *P*3, *M*3)
*confer.read*(*P*2, *P*3, *F*2) . ■

### 4.1.3. Protection Policies

A configuration of the access matrix describes what subjects can do—not necessarily what they are authorized to do. A **protection policy** (or **security policy**) partitions the set of all possible states into authorized versus unauthorized states. The next example shows how a simple policy regulating message buffers shared by two communicating processes can be formally described in terms of authorized states.

***Example:***
The following specifies that for every message buffer *b*, there exists exactly one process *p* that can write into the buffer and one process *q* that can read from the buffer (see Figure 4.3).

FIGURE 4.3 Shared message buffer.



A state $Q = (S, O, A)$ is authorized if and only if for every buffer $b \in O$, there exists exactly one process $p \in S$ such that $W \in A[p, b]$ and exactly one process $q \neq p$ such that $R \in A[q, b]$. ■

Whether a state is authorized can depend on the previous state and the command causing the state transition:

***Example:***
Consider the system described in the previous section. The following specifies that no subject can acquire access to a file unless that right has been explicitly granted by the file's owner:

Let $Q = (S, O, A)$ be an authorized state such that $Own \in A[p, f]$ for subject $p$ and file $f$, but $r \notin A[q, f]$ for subject $q$ and right $r$. Let $Q' = (S', O', A')$ be a state such that $r \in A'[q, f]$ and $Q \vdash_c Q'$. Then $Q'$ is authorized if and only if

$$c = \mathit{confer.read}(p, q, f)\ . \quad ■$$

(See also [Krei80] for descriptions of protection policies).

## 4.2 ACCESS CONTROL MECHANISMS

### 4.2.1 Security and Precision

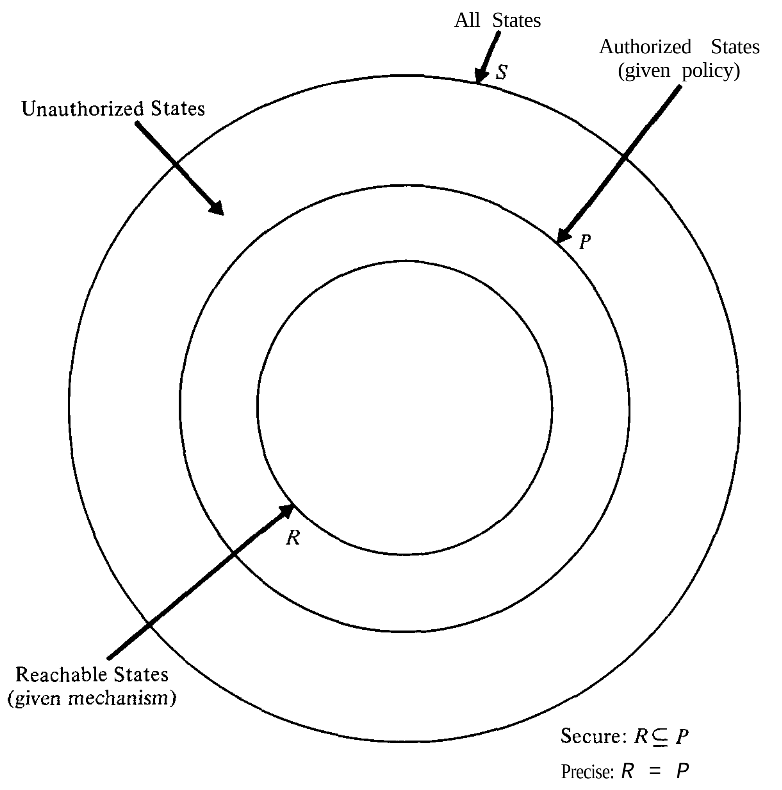
The access control mechanisms of a system enforce the system's security policies by ensuring that the physical states of the system correspond to authorized states of the abstract model. These mechanisms must monitor all accesses to objects as well as commands that explicitly transfer or revoke rights. Otherwise, the system could enter a physical state that does not correspond to an authorized state of the model. For example, a subject could bypass a file system and issue a read request directly to the physical location of a file on disk. The security mechanisms must also ensure that the physical resource states correspond to logical object states of the model. Systems that do not clear memory between use, for example, may expose data to unauthorized subjects.

The security mechanisms of a system may be overprotective in the sense of preventing entry to an authorized state or of denying an authorized access. For example, a system might never allow a subject access to a file unless the subject owns the file. Systems that are overprotective are secure, but may not satisfy other requirements of the system.

The preceding requirements are summarized with the aid of Figure 4.4. Let $S$ be the set of all possible states, and let $P$ be the subset of $S$ authorized by the protection policies of the system. Let $R$ be the subset of $S$ that is reachable with the security mechanisms in operation. The system is **secure** if $R \subseteq P$; that is, if all reachable states are authorized. The system is **precise** (not overprotective) if $R = P$; that is, if all authorized states are reachable (see Figure 4.4).

After discussing the general requirements of security mechanisms, we shall describe particular mechanisms. We shall then turn to the problem of designing

FIGURE 4.4 Security and precision.



systems whose security can be verified. Finally, we shall examine theoretical questions related to proving properties about the transfer of rights in abstract models.

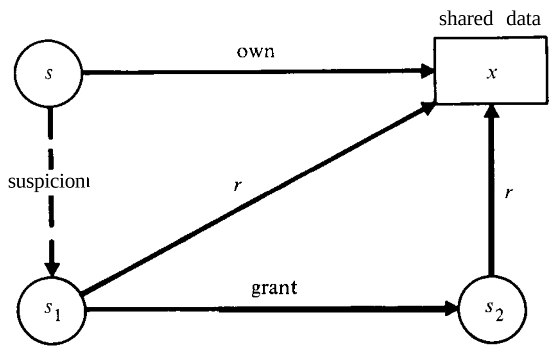
## 4.2.2 Reliability and Sharing

Protection mechanisms enforce policies of controlled sharing and reliability. Several levels of sharing are possible:

1. No sharing at all (complete isolation).
2. Sharing copies of data objects.
3. Sharing originals of data objects.
4. Sharing untrusted programs.

Each level of sharing imposes additional requirements on the security mechanisms of a system. The sharing of data introduces a problem of **suspicion,** where the owner s of a data object x may not trust another subject $s_1$ with whom s shares access. For example, s may fear that $s_1$ will grant its access rights to x to

FIGURE 4.5 Suspicion.



another subject $s_2$ (see Figure 4.5). Alternatively, s may fear that $s_1$ will simply misuse its privileges-for example, $s_1$ could read x and broadcast its contents to other subjects (see Figure 4.6). The second problem does not fall under the scope of an access control policy, however, because it involves the transfer of information rather than the transfer of rights; controls for information flow are studied in the next chapter.

Most systems have facilities for sharing originals of programs or data. On-line database systems often allow multiple users simultaneous access to a common database. Most operating systems allow concurrent processes to execute the same code and access global system tables or file directories.

There are several reasons for sharing originals of data (or programs) rather than just copies. A principal reason is to save space. Another is to save time that

FIGURE 4.6 Misuse of privilege.

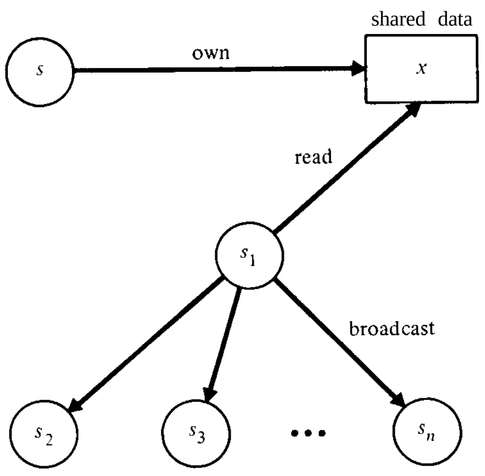
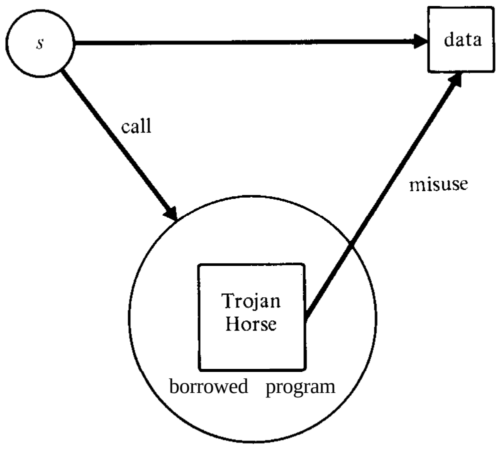


FIGURE 4.7 Trojan Horse.



would otherwise be needed to make duplicates or transfer updates to a master copy. A third reason is to ensure each subject has a consistent, up-to-date view of the data.

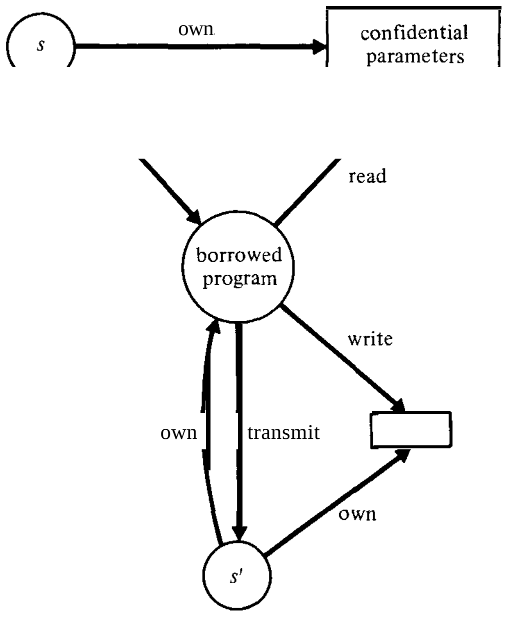
Protecting originals of data imposes additional requirements on the security mechanisms, however, because a subject with write-access to the original might destroy the data. Concurrent accesses to the data must be controlled to prevent writing subjects from interfering with other subjects.

If programs are shared, the protection problem is considerably more complex than if only data is shared. One reason is a **Trojan Horse** may be lurking inside a borrowed program. A Trojan Horse performs functions not described in the program specifications, taking advantage of rights belonging to the calling environment to copy, misuse, or destroy data not relevant to its stated purpose (see Figure 4.7). A Trojan Horse in a text editor, for example, might copy confidential information in a file being edited to a file accessible to another user. The attack was first identified by D. Edwards and described in the Anderson report [Ande72]. The term Trojan Horse is often used to refer to an entry point or "trapdoor" planted in a system program for gaining unauthorized access to the system, or to any unexpected and malicious side effect [Lind75]. Protection from Trojan Horses requires encapsulating programs in small domains with only the rights needed for the task, and no more.

Even if encapsulated, a borrowed program may have access to confidential parameters passed by the calling subject s. The program could transmit the data to the program's owner $s'$, or retain it in objects owned by $s'$ for later use (see Figure 4.8). For example, a compiler might make a copy of a proprietary software program. A program that cannot retain or leak its parameters is said to be **memoryless** or **confined.**

Lampson [Lamp731 was the first to discuss the many subtleties of the confinement problem. As long as a borrowed program does not have to retain any

FIGURE 4.8 The confinement problem.



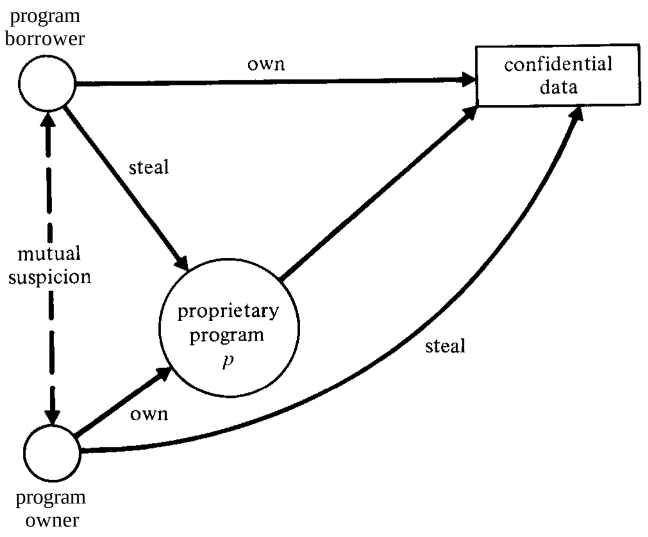
information, confinement can be implemented by restricting the access rights of the program (and programs called by it). But as soon as the program must retain nonconfidential information, access controls alone are insufficient, because there is no way of ensuring the program does not also retain confidential information. This is called the **selective confinement** problem, and is treated in the next chapter.

Protecting proprietary programs is also difficult. The owner needs assurances that borrowers can execute the programs, but cannot read or copy them. We saw in Chapter 3 how a form of encryption can be used to protect proprietary programs run on the customer's system. Access controls can be used to protect programs run on the owner's system, because the borrower cannot copy a program if it is only given access rights to execute (but not read) the program.

The sharing of software, therefore, introduces a problem of **mutual suspicion** between the owner and the borrower: the owner of a program *p* may be concerned the borrower will steal *p*; the borrower may be concerned the owner will steal confidential input to *p* (see Figure 4.9).

Protection mechanisms are needed for reliability as much as sharing. They prevent malfunctioning programs from writing into segments of memory belonging to the supervisor or to other programs. They prevent undebugged software from disrupting the system. They prevent user programs from writing directly on a disk, destroying files and directories. They provide backup copies of files in case of hardware error or inadvertent destruction. By limiting the damage caused by mal-

FIGURE 4.9 Mutual suspicion.



functioning programs, they provide error confinement and an environment for recovery.

Protection was originally motivated by time-sharing systems serving multiple users simultaneously and providing long-term storage for their programs and data. Protection was essential both for reliability and controlled sharing.

The recent trend toward distributed computing through networks of personal computers mitigates the need for special mechanisms to isolate users and programs-each user has a private machine. It is often unnecessary for users to share originals of programs in this environment, because it is more economical to run a separate copy on each user's machine. Controlled sharing is still essential, however; the difference is that it must now be provided by the network rather than by the users' personal computers. Users may wish to exchange data and programs on the network, store large files in central storage facilities, access database systems, or run programs on computers having resources not provided by their own computers.

Network security requires a combination of encryption and other security controls. Encryption is needed to protect data whose transmission over the network is authorized. Information flow controls are needed to prevent the unauthorized dissemination of confidential data, classified military data, and proprietary software over the network. Access controls are needed to prevent unauthorized access to the key management facilities, to shared databases, and to computing facilities. They are needed for reliability; loss of a single node on the network should not bring down the entire network, or even make it impossible to transmit messages that would have been routed through that node.

### 4.2.3 Design Principles

Saltzer and Schroeder identified several design principles for protection mechanisms [Salt75]:

1. **Least privilege:** Every user and process should have the least set of access rights necessary. This principle limits the damage that can result from error or malicious attack. It implies processes should execute in **small protection domains,** consisting of only those rights needed to complete their tasks. When the access needs of a process change, the process should switch domains. Furthermore, access rights should be acquired by explicit permission only; the default should be lack of access (Saltzer and Schroeder called this a "fail-safe" default). This principle is fundamental in containing Trojan Horses and implementing reliable programs; a program cannot damage an object it cannot access.
2. **Economy of mechanism:** The design should be sufficiently small and simple that it can be verified and correctly implemented. Implementing security mechanisms in the lowest levels of the system (hardware and software) goes a long way toward achieving this objective, because the higher levels are then supported by a secure base. This means, however, security must be an integral part of the design. Attempting to augment an existing system with security mechanisms usually results in a proliferation of complex mechanisms that never quite work. Although the system must be sufficiently flexible to handle a variety of protection policies, it is better to implement a simple mechanism that meets the requirements of the system than it is to implement one with complicated features that are seldom used.
3. **Complete mediation:** Every access should be checked for authorization. The mechanism must be efficient, or users will find means of circumventing it.
4. **Open design:** Security should not depend on the design being secret or on the ignorance of the attackers [Bara64]. This principle underlies cryptographic systems, where the enciphering and deciphering algorithms are assumed known.
5. **Separation of privilege:** Where possible, access to objects should depend on more than one condition being satisfied. The key threshold schemes discussed in Section 3.8 illustrate this principle; here more than one shadow is needed to restore a key.
6. **Least common mechanism:** Mechanisms shared by multiple users provide potential information channels and, therefore, should be minimized. This principle leads to mechanisms that provide user isolation through physically separate hardware (distributed systems) or through logically separate virtual machines [Pope74,Rush81].
7. **Psychological acceptability:** The mechanisms must be easy to use so that they will be applied correctly and not bypassed. In particular, it must not be substantially more difficult for users to restrict access to their objects than it is to leave access to them unrestricted.

Access control mechanisms are based on three general concepts:

1. **Access Hierarchies,** which automatically give privileged subjects a superset of the rights of less privileged subjects.
2. **Authorization Lists,** which are lists of subjects having access rights to some particular object.
3. **Capabilities,** which are like "tickets" for objects; possession of a capability unconditionally authorizes the holder access to the object.

Examples of mechanisms based on one or more of these concepts are discussed in Sections 4.3-4.5.

## 4.3 ACCESS HIERARCHIES

We shall describe two kinds of mechanisms based on access hierarchies: privileged modes and nested program units.

### 4.3.1 Privileged Modes

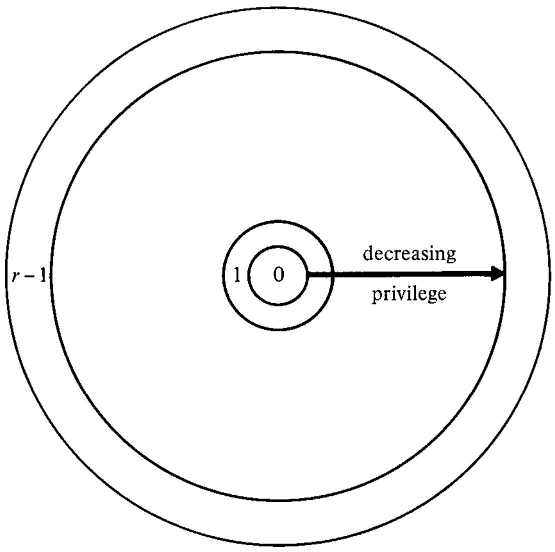
Most existing systems implement some form of **privileged mode** (also called **supervisor state)** that gives supervisor programs an access domain consisting of every object in the system. The state word of a process has a l-bit flag indicating whether the process is running in privileged mode or in nonprivileged (user) mode. A process running in privileged mode can create and destroy objects, initiate and terminate processes, access restricted regions of memory containing system tables, and execute privileged instructions that are not available to user programs (e.g., execute certain I/O operations and change process statewords). This concept of privileged mode is extended to users in UNIX, where a "super user" is allowed access to any object in the system.

The **protection rings** in MULTICS are a generalization of the concept of supervisor state [Grah68,Schr72,0rga72]. The state word of a process *p* specifies an integer ring number in the range [0, r - 1. Each ring defines a domain of access, where the access privileges of ring j are a subset of those for ring ***i,*** for all $0 \le i < j \le r - 1$ (see Figure 4.10). The supervisor state has $r$ = 2.

Supervisor states and ring structures are contrary to the principle of least privilege. Systems programs typically run with considerably more privilege than they require for their tasks, and ring 0 programs have full access to the whole system. A single bug or Trojan Horse in one of these programs could do considerable damage to data in main memory or on disk.

There are numerous examples of users who have exploited a design flaw that enabled them to run their programs in supervisor state or plant a Trojan Horse in some system module. For example, Popek and Farber [Pope781 describe an "ad-

FIGURE 4.10 MULTICS rings.



dress wraparound" problem in the PDP-10 TENEX system: under certain conditions, a user could force the program counter to overflow while executing a supervisor call, causing the privileged-mode bit in the process state word to be turned on, and control to return to the user's program in privileged mode.

This does not mean supervisor states are inherently bad. They have strengthened the security of many systems at low cost. But systems requiring a high level of security need additional mechanisms to limit the access rights of programs running in supervisor state. Moreover, these systems require verifying that there are no trapdoors whereby a user can run programs in supervisor state.

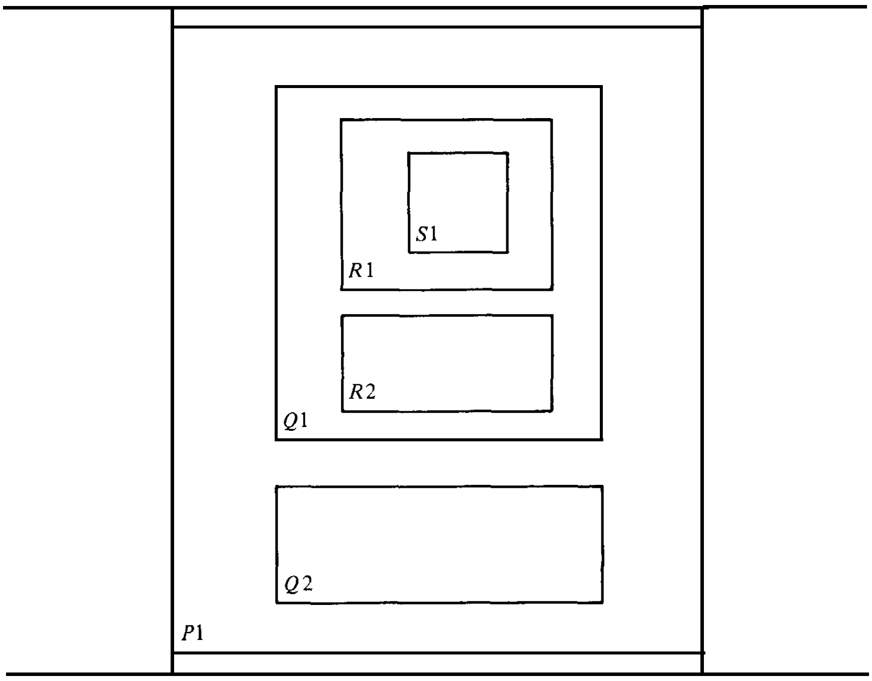
### 4.3.2 Nested Program Units

The scope rules of languages such as ALGOL, PL/I, and Pascal automatically give inner program units (e.g., procedures and blocks) access to objects declared in enclosing units-even if they do not require access to these objects. The inner program units are, therefore, much like the inner rings of MULTICS. Whereas objects declared in the inner units are hidden from the rest of the program, objects declared in the outer units may be accessible to most of the program.

***Example:***

Consider the program structure shown in Figure 4.11. The inner program unit *Sl* has access to its own local objects as well as global objects declared in the enclosing units *R*1, Ql, and *P*1 (as long as there are no name conflicts). No other unit can access objects local to *S*1 unless they are explicitly passed as parameters by *S*1. The outermost unit *P*1 has access only to the

FIGURE 4.11 Block structured program.



objects declared in $P1$, though these objects are global to the entire program. ■

Programs that exploit the full capabilities of nested scopes are often difficult to maintain. An inner unit that modifies a global data object has side effects that can spread into other units sharing the same global objects. Changes to the code in the innermost units can affect code in the other units. Changes to the structure of a global object can affect code in the innermost units of the program.

Some recent languages have facilities for restricting access to objects in enclosing program units. In Euclid [Lamp76a], for example, a program unit can access only those objects that are accessible in the immediately enclosing unit and either explicitly **imported** into the unit (through an **imports list)** or declared **pervasive** (global) in some enclosing block. Ada† has a similar facility, where a **restricted** program unit can access only those objects declared in an enclosing unit and included in the **visibility list** for the unit.

## 4.4 AUTHORIZATION LISTS

**An authorization list** (also called an **access-control list)** is a list of $n \geq 0$ subjects who are authorized to access some particular object $x$. The *ith* entry in the list gives the name of a subect $s_i$ and the rights $r_i$ in $A[s_i, x]$ of the access matrix:

†Ada is a trademark of the Department of Defense.

Authorization List

$s_1, r_1$

$s_2, r_2$

$s_n, r_n$ .

An authorization list, therefore, represents the nonempty entries in column x of the access matrix.

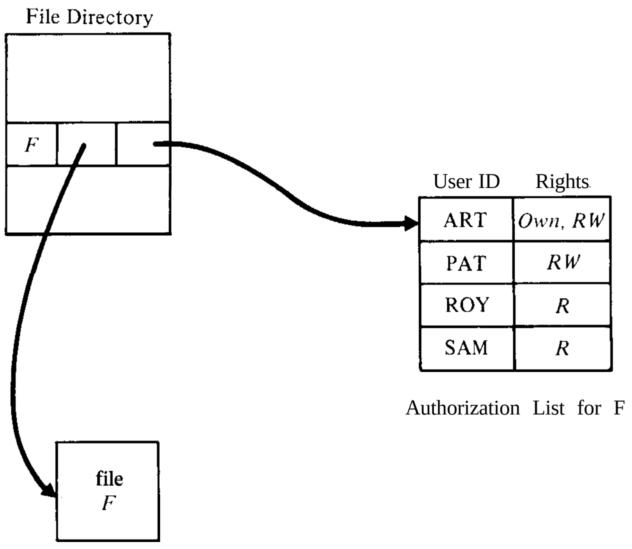
## 4.4.1 Owned Objects

Authorization lists are typically used to protect owned objects such as files. Each file has an authorization list specifying the names (or IDs) of users or user groups, and the access rights permitted each. Figure 4.12 illustrates.

The owner of a file has the sole authority to grant access rights to the file to other users; no other user with access to the file can confer these rights on another user (in terms of the abstract model, the copy flag is off). The owner can revoke (or decrease) the access rights of any user simply by deleting (or modifying) the user's entry in the authorization list.

MULTICS uses authorization lists to protect segments in long-term storage [Dale65,Bens72,0rga72]. Each segment has an access-control list -with an entry for each user permitted access to the segment. Each entry in the list indicates the

FIGURE 4.12 Authorization list for file.



type of access (read, write, or execute) permitted, together with the range of rings (bracket) over which this permission is granted:

$(r_1, r_2)$—the read bracket, $r_1 \leq r_2$
$(w_1, w_2)$—the write bracket, $w_1 \leq w_2$
$(e_1, e_2)$—the execute bracket, $e_1 \leq e_2$ .

A process executing in ring $i$ (see previous section) is not allowed access to a segment unless the user associated with the process is listed in the access-control list, and $i$ falls within the range corresponding to the type of access desired.

In the SWARD system, authorization lists are associated with **access sets** for objects [Buck80]. Users can group objects together into object sets, and then define access sets over objects, object sets, and other access sets.

***Example:***
Figure 4.13 illustrates an access set for the object sets $X$, $Y$, and $Z$. The access set $A1$ is defined by the expression $X + Y - Z$, which evaluates left-to-right to the set of objects $\{a_2, c\}$ (duplicate names in $Y$ are removed; thus $a_1$, the first version of $a$, is not included). Because the set $A1$ is defined by an expression, its constituency can change as objects are added to or deleted from the object sets $X$, $Y$, and $Z$. ■

To access an object, a user must specify the name of the object and an access

FIGURE 4.13 Access sets in SWARD.

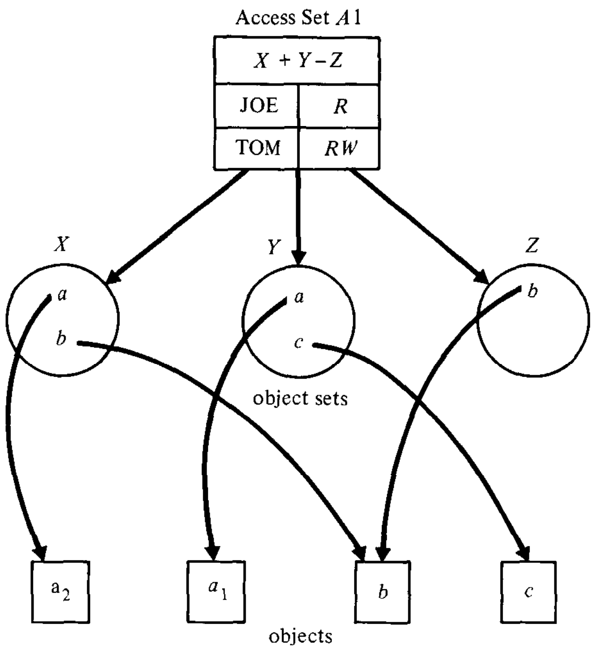
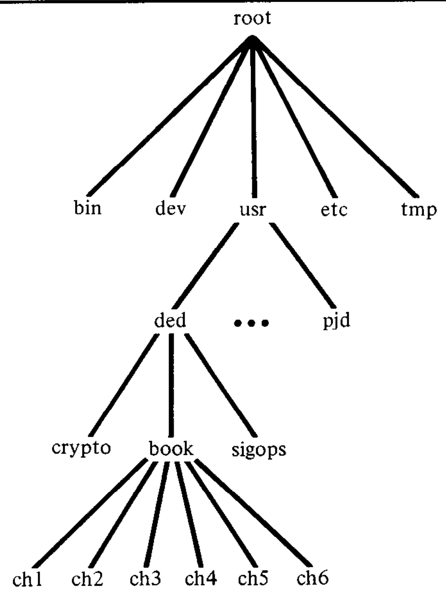


FIGURE 4.14 UNIX file directory tree.



set. The user is granted access to the object only if it is included in the access set; in addition, the authorization list for the set must have an entry for the user with the appropriate access rights.

In UNIX, each file has an authorization list with three entries: one specifying the owner's access rights; a second specifying the access rights of all users in the owner's group (e.g., faculty, students, etc.), and a third specifying the access rights of all others [Ritc74]. The access rights for each entry include *R*, *W*, and *E*, (*E* is interpreted as "directory search" for directory files). The UNIX file system has a tree-structured directory, and files are named by paths in the tree.

***Example:***
Figure 4.14 shows a portion of the tree structure for the UNIX file system that contains the working version of this book. A user (other than ded) wishing to read Chapter 4 must specify the path name /usr/ded/book/ch4. Access is granted provided the user has *E*-access to the directories usr, ded, and book, and *R*-access to the file ch4. ■

Many systems use authorization lists with only two entries: one specifying the owner's access rights, and the other specifying the access rights of all others. The access rights in these systems are usually limited to *R* and *W*.

These degenerate forms of authorization lists do not meet the objective of least privilege. Nevertheless, they are efficient to implement and search, and adequate for many applications.

Because an authorization list can be expensive to search, many systems do not check the authorization list for every access. A file system monitor, for example, might check a file's authorization list when the file is opened, but not for each read or write operation. Consequently, if a right is revoked after a file is opened, the revocation does not take effect until the file is closed. Because of their inefficiencies, authorization lists are not suitable for protecting segments of memory, where address bounds must be checked for every reference.

We saw in Section 3.7.1 (see Figure 3.19) how access to a file $F$ encrypted under a key $K$ (or keys if separate read and write keys are used) could be controlled by a "keys record", where every user allowed access to $F$ has an entry in the record containing a copy of $K$ enciphered under the user's private transformation. The keys record is like an authorization list, and is impossible to forge or bypass (the key is needed to access the file). Standard authorization lists for encrypted or unencrypted files could also be protected from unauthorized modification by encrypting them under a file monitor key.

## 4.4.2 Revocation

The authorization lists we have described so far have the advantage of giving the owner of an object complete control over who has access to the object; the owner can revoke any other user's access to the object by deleting the user's entry in the list (though the revocation might not take effect immediately as noted earlier). This advantage is partially lost in systems that use authorization lists to support nonowned objects, and allow any user to grant and revoke access rights to an object.

An example of such a system is System R [Grif76], a relational database system [Codd70,Codd79] developed at the IBM Research Laboratory in San Jose. The protected data objects of System R consist of **relations**, which are sets (tables) of $n$-tuples (rows or records), where each $n$-tuple has $n$ attributes (columns). A relation can be either a **base relation**, which is a physical table stored in memory, or a **view**, which is a logical subset, summary, or join of other relations.

***Example:***
An example of a relation is

*Student (Name, Sex, Major, Class, SAT, GP).*

Each tuple in the *Student* relation gives the name, sex, major, class, SAT score, and grade-point of some student in the database (see Table 6.1 in Chapter 6). ■

The access rights for a relation (table) include:

| | |
|---|---|
| *Read:* | for reading rows of the table, using the relation in queries, or defining views based on the relation. |
| *Insert:* | for adding new rows to a table. |

|  |  |
|---|---|
| *Delete:* | for deleting rows in a table. |
| *Update:* | for modifying data in a table or in certain columns of a table. |
| *Drop:* | for deleting a table. |

Any user *A* with access rights to a table can grant these rights to another user *B* (provided the copy flag is set). *A* can later revoke these rights, and the state of the system following a revocation should be as if the rights had never been granted. This means that if *B* has subsequently granted these rights to another user *C*, then *C*'s rights must also be lost. In general, any user who could not have obtained the rights without the grant from *A* to *B* must lose these rights.

To implement this, each access right granted to a user is recorded in an access list called *Sysauth*. *Sysauth* is a relation, where each tuple specifies the user receiving the right, the name of the table the user is authorized to access, the type of table (view or base relation), the grantor making the authorization, the access rights granted to the user, and a copy flag (called a *Grant* option). Each access right (except for *Update*) is represented by a column of *Sysauth* and indicates the time of the grant (a time of 0 means the right was not granted). The timestamps are used by the revocation mechanism to determine the path along which a right has been disseminated.

The column for *Update* specifies *All*, *None*, or *Some*. If *Some* is specified, then for each updatable column, a tuple is placed in a second authorization table called *Syscolauth*. It is unnecessary to provide a similar feature for *Read*, because any subset of columns can be restricted through the view mechanism. For example, access only to *Name*, *Sex*, *Major*, and *Class* for the *Student* relation can be granted by defining a view over these attributes only (i.e., excluding *SAT* and *GP*).

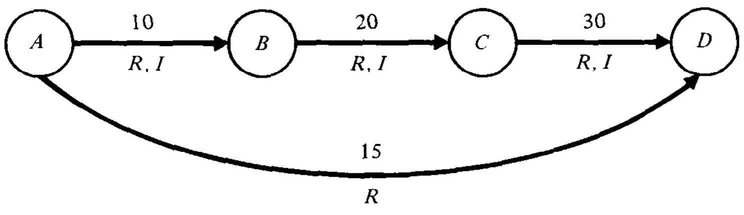
Note that the authorization list for a particular relation *X* can be obtained from *Sysauth* by selecting all tuples such that *Table* = *X*.

***Example:***
The following shows the tuples in *Sysauth* that result from a sequence of grants for a relation *X* created by user *A*. (Columns for table type and the rights *Delete*, *Update*, and *Drop* are not shown.)

| User | Table | Grantor | Read | Insert . . . | Copy |
|---|---|---|---|---|---|
| *B* | *X* | *A* | 10 | 10 | yes |
| *D* | *X* | *A* | 15 | 0 | no |
| *C* | *X* | *B* | 20 | 20 | yes |
| *D* | *X* | *C* | 30 | 30 | yes . |

User *B* obtained both *Read* and *Insert* access to *X* from *A* at time $t = 10$, and passed both rights to *C* at time $t = 20$. User *D* obtained *Read* access to *X* from *A* at time $t = 15$, and both *Read* and *Insert* access to *X* from *C* at time $t = 30$. Whereas *D* can grant the rights received from *C*, *D* cannot grant the *Read* right received from *A*. ■

FIGURE 4.15 Transfer of rights for relation *X*.



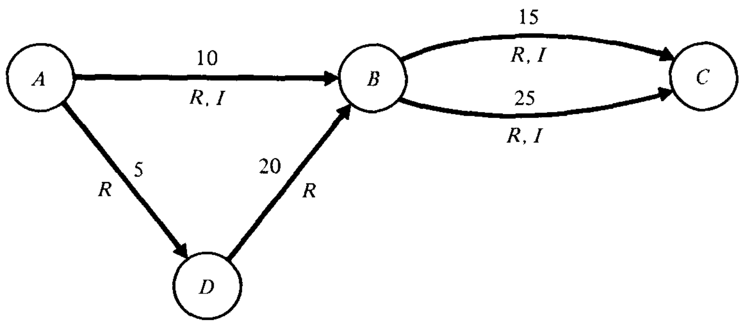
Griffiths and Wade [Grif76] use directed graphs to illustrate the transfer and revocation of rights for a particular relation. Each node of a graph represents a user with access to the relation. Each edge represents a grant and is labeled with the time of the grant; we shall also label an edge with the rights transferred.

***Example:***
Figure 4.15 shows a graph for the relation *X* of the previous example. Suppose that at time *t* = 40, *A* revokes the rights granted to *B*. Then the entries in *Sysauth* for both *B* and *C* must be removed because *C* received these rights from *B*. Although *D* is allowed to keep rights received directly from *A*, *D* must forfeit other rights received from *C*. The final state of *Sysauth* is thus:

| User | Table | Grantor | Read | Insert . . . Copy | |
|---|---|---|---|---|---|
| *D* | *X* | *A* | 15 | 0 | no . ■ |

***Example:***
Figure 4.16 illustrates a more complicated situation, where *B* first grants rights to *C* received from *A* (*t* = 15), and then later grants rights to *C* received from *D* (*t* = 25). The state of *Sysauth* is thus:

FIGURE 4.16 Transfer of rights for relation *Y*.



| User | Table | Grantor | Read | Insert . . . | Copy |
|---|---|---|---|---|---|
| *D* | *Y* | *A* | 5 | 0 | yes |
| *B* | *Y* | *A* | 10 | 10 | yes |
| *C* | *Y* | *B* | 15 | 15 | yes |
| *B* | *Y* | *D* | 20 | 0 | yes |
| *C* | *Y* | *B* | 25 | 25 | yes . ■ |

In the preceding example, we recorded in *Sysauth* the duplicate grants from *B* to *C* (at $t = 15$ and $t = 25$). This differs from the procedure given by Griffiths and Wade, which records only the earliest instance of a grant and, therefore, would not have recorded the one at $t = 25$. Fagin [Fagi78] observed that unless all grants are recorded, the revocation procedure may remove more rights than necessary.

***Example:***
Suppose *A* revokes the *Read* and *Insert* rights given to *B*. *B* should be allowed to keep the *Read* right received from *D*, and *C* should be allowed to keep the *Read* right received from *B* at time $t = 25$ that was passed along the path (*A*, *D*, *B*, *C*). Both *B* and *C*, however, must forfeit their *Insert* rights. The final state of *Sysauth* should therefore be:

| User | Table | Grantor | Read | Insert . . . | Copy |
|---|---|---|---|---|---|
| *D* | *Y* | *A* | 5 | 0 | yes |
| *B* | *Y* | *D* | 20 | 0 | yes |
| *C* | *Y* | *B* | 25 | 0 | yes . |

If the duplicate entry at time $t = 25$ had not been recorded in *Sysauth*, *C*'s *Read* right for *Y* would have been lost. ■

## 4.5 CAPABILITIES

A **capability** is a pair $(x, r)$ specifying the unique name (logical address) of an object $x$ and a set of access rights $r$ for $x$ (some capabilities also specify an object's type). The capability is a ticket in that possession unconditionally authorizes the holder $r$-access to $x$. Once the capability is granted, no further validation of access is required. Without the capability mechanism, validation would be required on each access by searching an authorization list.

The concept of capability has its roots in Iliffe's "codewords", which were implemented in the Rice University Computer [Ilif62] in 1958, and generalized in the Basic Language Machine [Ilif72] in the early 1960s. A codeword is a descriptor specifying the type of an object and either its value (if the object is a single element such as an integer) or its length and location (if the object is a structure of elements such as an array). A codeword also has a special tag that allows it to be recognized and interpreted by the hardware. A similar concept was embodied in the "descriptors" of the Burroughs B5000 computer in 1961.

Dennis and VanHorn [DeVH66] introduced the term "capability" in 1966. They proposed a model of a multiprogramming system that used capabilities to control access to objects that could either be in main memory or in secondary (long-term) storage. In their model, each process executes in a domain called a "sphere of protection", which is defined by a **capability list**, or **C-list** for short. The C-list for a domain $s$ is a list of $n \geq 0$ capabilities for the objects permitted to $s$:

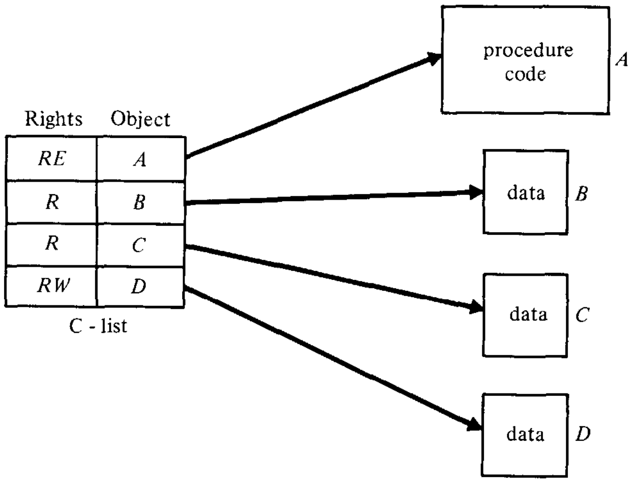
$$\begin{array}{l} \underline{\text{C-List}} \\ x_1, r_1 \\ x_2, r_2 \\ \quad\cdot \\ \quad\cdot \\ \quad\cdot \\ x_n, r_n \end{array}$$

where $r_i$ gives the rights in $A[s, x_i]$ of the access matrix. The C-list for $s$, therefore, represents the nonempty entries in row $s$ of the access matrix.

**Example:**

Figure 4.17 illustrates a C-list that provides read/execute-access ($RE$) to the code for procedure $A$, read-only-access ($R$) to data objects $B$ and $C$, and read/write-access ($RW$) to data object $D$. The diagram shows each capability pointing directly to an object; the mapping from capabilities to object locations is described in Section 4.5.3. ■

FIGURE 4.17 Capability list.

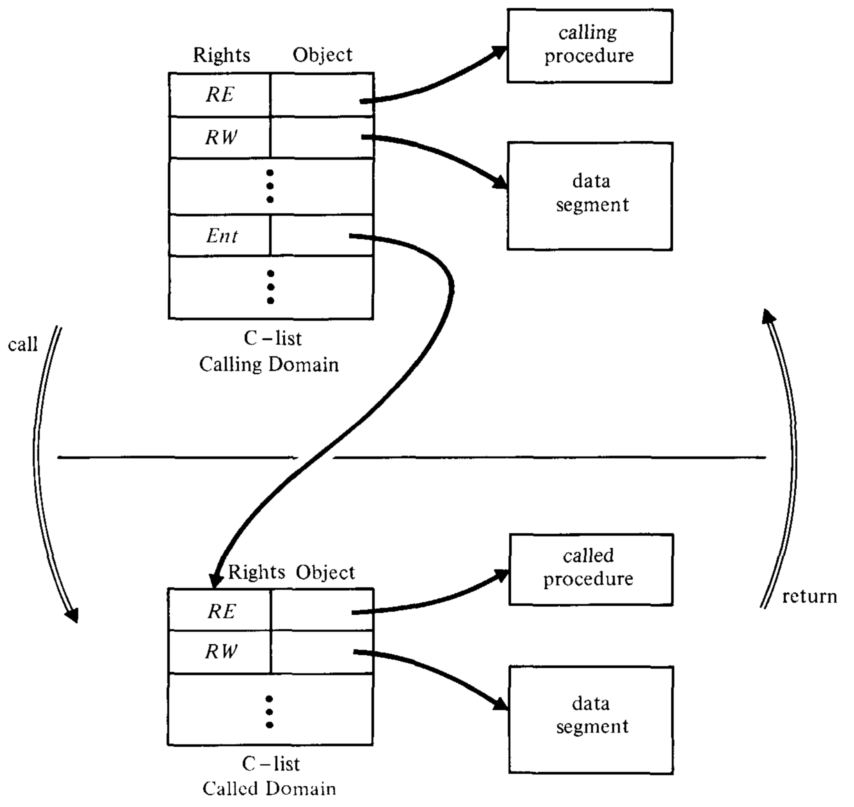


Capabilities have been the underlying protection mechanism in several systems, including the Chicago Magic Number Computer [Fabr71a], the BCC model I system [Lamp69], the SUE system [Sevc74], the Cal system [Lamp76b], the Plessey System 250 [Engl74], HYDRA [Wulf74,Cohe75], the CAP system [Need77], StarOS for CM* [Jone79], UCLA Secure UNIX [Pope79], iMAX for the INTEL iAPX 432 [Kahn81], the SWARD system [Myer80,Myer78], and PSOS [Feie79,Neum80].

## 4.5.1 Domain Switching with Protected Entry Points

Dennis and VanHorn envisaged a mechanism for supporting small protection domains and abstract data types. A principal feature of their mechanism was an enter capability (denoted by the access right *Ent*). An enter capability points to a C-list and gives the right to transfer into the domain defined by the C-list. Enter capabilities provide a mechanism for implementing **protected entry points** into

FIGURE 4.18 Domain switch with protected entry points.



procedures, or **protected subsystems**. When a process calls a protected procedure, its C-list (and therefore domain) is changed. When the procedure returns, the former C-list is restored (see Figure 4.18).

This concept of giving each procedure its own set of capabilities supports the principle of least privilege. Each capability list need contain entries only for the objects required to carry out its task. Damage is confined in case the program contains an error. An untrusted program can be encapsulated in an **inferior sphere of protection** where it cannot endanger unrelated programs. Data can be hidden away in a domain accessible only to the programs allowed to manipulate it. This provides a natural environment for implementing abstract data types (see also [Linn76]).

### 4.5.2 Abstract Data Types

An **abstract data type** is a collection of objects and operations that manipulate the objects. Programming languages that support abstract data types have facilities for defining a **type module** that specifies:

1. The **name** *t* of an object type.
2. The **operations** (procedures, functions) that may be performed on objects of type *t*. There are two categories of operations: external operations, which provide an interface to the module from the outside, and internal operations, which are available within the module only. The semantics of the operations may be defined by a set of axioms.
3. The **representation** or implementation of objects of type *t* in terms of more primitive objects. This representation is hidden; it is not available to procedures outside the module.

Abstract data types are also called **extended-type objects**, because they extend the basic built-in types of a programming language. Languages that support abstract data types include Simula 67 [Dahl72], CLU [Lisk77], Alphard [Wulf76], MOD-EL [Morr78], and Ada. These languages are sometimes called "object-oriented" languages.

> ***Example:***
> Figure 4.19 shows the principal components of a type module for a stack of integer elements, implemented as a sequential vector; all procedures are available outside the module. ■

A type module encapsulates objects in a small protection domain. These objects can be accessed only through external procedures that serve as protected entry points into the module. The module provides an environment for **information hiding**, where the low-level representation of an object is hidden from outside procedures that perform high-level operations on the data.

The principal motivation for abstract data types is program reliability and

FIGURE 4.19 Type module for stacks.

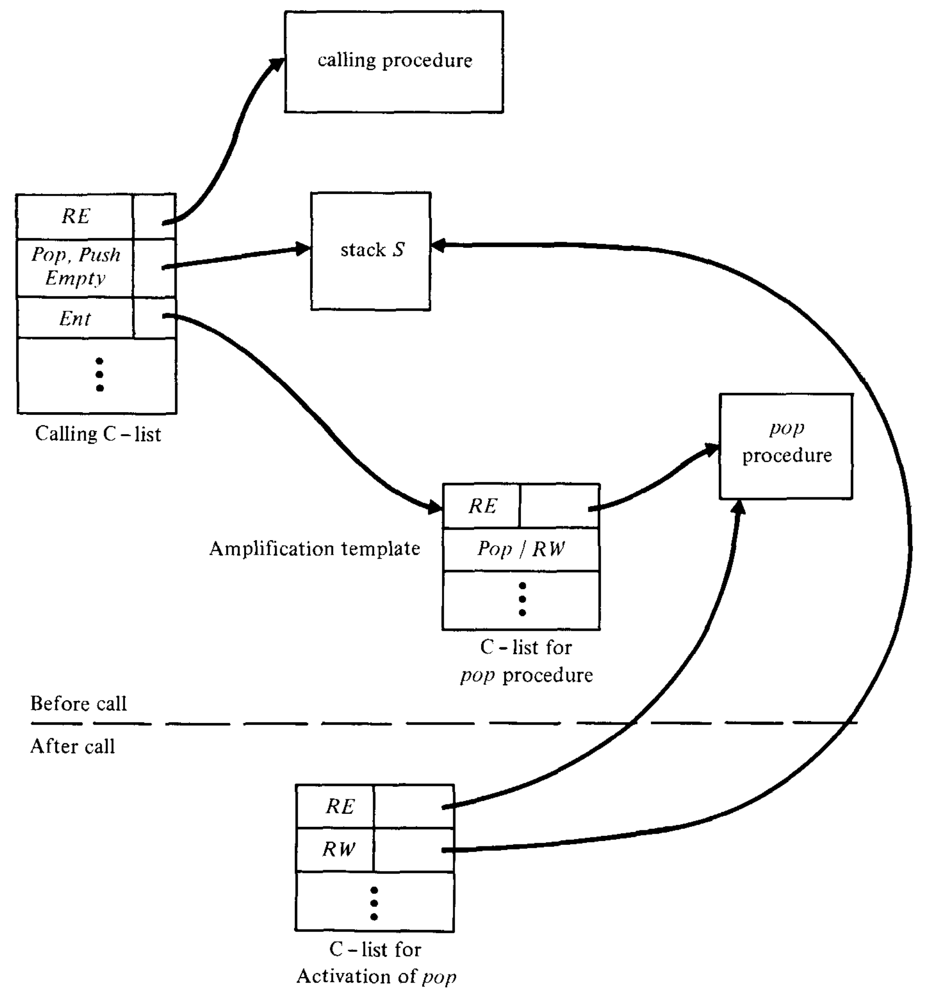
```
module stack
   constant size = 100;
   type stack =
     record of
       top: integer, init 0;
       data: array[1 . . size] of integer;
     end;
   procedure push(var s: stack; x: integer);
     begin
       s.top := s.top + 1;
       if s.top > size
         then "stack overflow"
         else s.data[s.top] := x
     end;
   procedure pop(var s: stack): integer;
     begin
       if s.top = 0
         then "stack underflow"
         else begin
           pop := s.data[s.top];
           s.top := s.top - 1;
           end
     end;
   procedure empty(var s: stack): boolean;
     begin
       if s.top = 0
         then empty := true
         else empty := false
     end
end stack
```

maintenance. Because the representation of an object is confined to a single module, changes to the representation should have little effect outside the module.

Capability-based systems provide an attractive environment for implementing abstract data types because the objects required by a program can be linked to the program by capabilities, and capabilities provide a uniform mechanism for accessing all types of objects. Iliffe's codewords, in fact, were devised to support data abstraction.

In HYDRA, for example, each procedure in a type module is represented by a C-list that includes capabilities for the procedure's code and local objects, and templates for the parameters passed to the procedure. If a process creates an instance $x$ of the object type defined by the module, it is given a capability $C$ for $x$ with rights to pass $x$ to the procedures of the module (or a subset of the procedures), but it is not given $RW$ rights to the representation of $x$. Because the procedures of the type module require $RW$ rights to the internal representation of $x$, they are permitted to **amplify** the rights in $C$ to those specified in an "amplification template".

FIGURE 4.20 Abstract data type and rights amplification.



***Example:***
Let *S* be an instance of type *stack* as defined in Figure 4.19. Figure 4.20 shows a calling C-list with *Pop, Push,* and *Empty* rights to *S*, and an *Ent* right for the *pop* procedure. The C-list for the *pop* procedure has an amplification template that allows it to amplify the *pop* right in a capability for a stack to *RW*. When the *pop* procedure is called with parameter *S*, a C-list for the activation of *pop* is created with *RW* rights for *S* (activation C-lists are analogous to procedure activation records). ■

FIGURE 4.21 Object encapsulated in private domain.

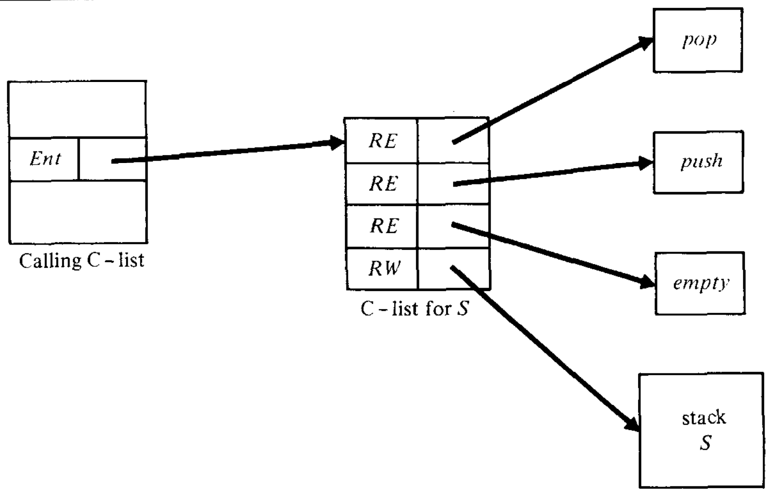
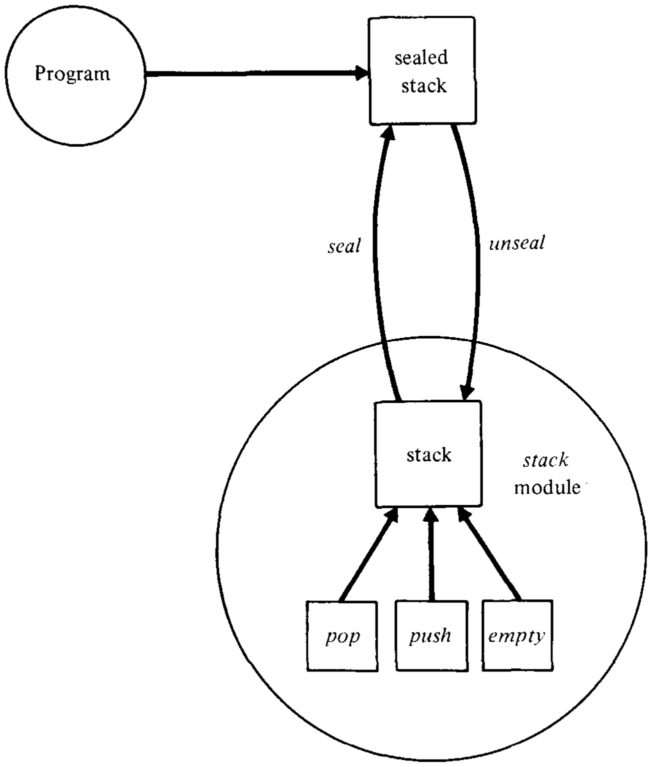


FIGURE 4.22 Sealed objects.



The capability mechanisms of HYDRA are implemented completely in software. Many of the features in HYDRA have been implemented in hardware in the INTEL iAPX 432. Its operating system, iMAX, is implemented in Ada. An Ada type module (called a "package") is represented by a capability list for the procedures and data objects of the module. If objects are passed to the module as parameters, their capabilities are represented by templates (called descriptor-control objects), and rights amplification is used as in HYDRA.

Objects can also be encapsulated in private domains, in which case rights amplification is not needed (see Figure 4.21). The private domain contains capabilities for the object plus the procedures for manipulating it. This strategy is also used in CAP.

Rights amplification is one way of implementing the *unseal* operation described by Morris [Mors73]. Morris suggested that an object of type *t* be *sealed* by the type manager for *t* such that only the type manager can *seal* and *unseal* it, and operations applied to the sealed object cause an error. The seal, therefore, serves to authenticate the internal structure of the object to the type manager, while hiding the structure from the remainder of the program. Figure 4.22 shows a sealed stack.

*Seal* and *unseal* could also be implemented with encryption. The type manager would encrypt an object before releasing it to the program, and only the type manager would be able to decrypt it for processing. This could degrade performance if the time required to encrypt and decrypt the object exceeds the processing time. When abstract data types are used primarily for program reliability and maintenance rather than for protection against malicious attacks, more efficient mechanisms are preferable. (A more general form of cryptographic sealing is described in Section 4.5.5.)

The SWARD machine designed by Myers effectively seals objects through its tagged memory. Every object in memory is tagged with a descriptor that identifies its type and size. If an object is a nonhomogeneous structure (e.g., a record structure), its elements are also tagged. A program must know the internal structure of an object to access it in memory. This means a program can hold a capability with *RW*-access to an object without being able to exercise these rights—only the type module will know the representation of the object and, therefore, be able to access it. This does not guarantee that a program cannot guess the internal representation of its objects and thereby access them, but it does prevent programs from accidentally modifying their objects. Myers calls this a "second level of protection"—the capabilities providing the "first level".

Although object-oriented languages have facilities for specifying access constraints on object types, they do not have facilities for specifying additional constraints on instances of a type. For example, it is not possible in these languages to specify that a procedure is to have access only to the *push* procedure for a stack $S_1$ and the *pop* and *empty* procedures for another stack $S_2$. Jones and Liskov [Jone76a] proposed a language extension that permits the specification of **qualified types** that constrain an object type to a subset of the operations defined on that type. The constraints imposed by qualified types can be enforced by the compiler or by the underlying capability mechanism.

Minsky [MinN78] has extended the concept of capability to distinguish between **tickets** for operands (data objects) and **activators** for operators (functions or procedures). A ticket is like a capability. An activator $A$ has the following structure:

$$A = (o, p_1, \ldots, p_k \mid G) \rightarrow p_o$$

where $o$ is an operator-identifier, $p_i$ is an access constraint on the $i$th operand of $o(1 \leq i \leq k)$, $G$ is a global constraint defined on all operands, and $p_o$ is a constraint on the result of the operator. The constraints $p_i$ may be data-dependent or state-dependent; the global constraint $G$ allows for the specification of context dependent conditions. A subject can apply the operator $o$ of an activator $A$ only if the operands satisfy the constraints of $A$.

The scheme could be implemented using a capability-based system that supports enter capabilities for protected modules. An activator $A$ for operator $o$ would be implemented as a capability to enter a protected module. The module would check the operands to determine if they satisfy the constraints of $A$; if so, it would execute the operator $o$.

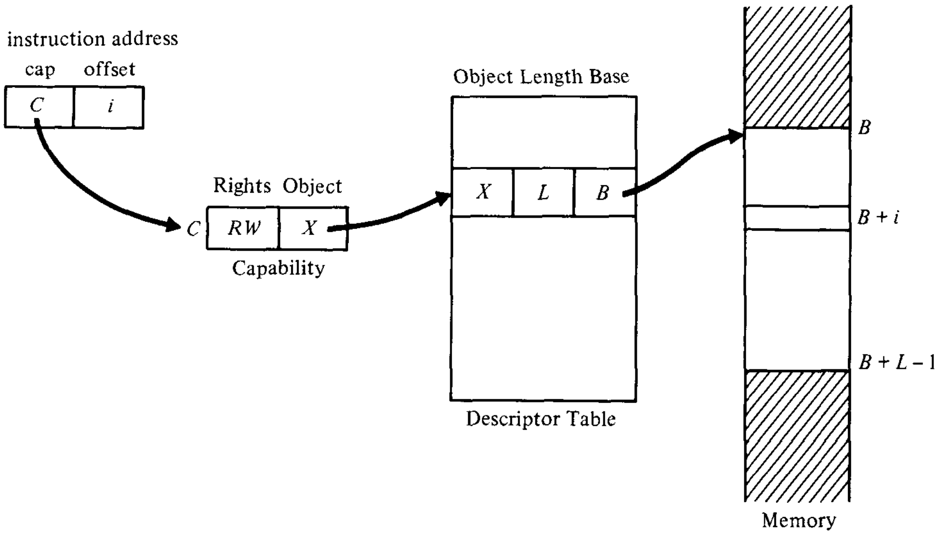
Minsky believes, however, that an underlying mechanism that implements both tickets and activators is preferable. He shows that such a scheme could support abstract data types without using rights amplification.

***Example:***
A process owning a stack $S$ would be given a ticket for $S$ and activators for popping and pushing arbitrary stacks. The type module, on the other hand, would be given activators for reading and writing the representation of stacks; these activators can be applied to the stack $S$ if the process owning $S$ passes the ticket for $S$ to the type module. The activators within the type module serve the same purpose as the templates used in HYDRA. ■

## 4.5.3 Capability-Based Addressing

Capabilities provide an efficient protection mechanism that can be integrated with a computer's addressing mechanism. This is called **capability-based addressing**. Figure 4.23 illustrates how the $i$th word of memory in a segment $X$ is addressed with a capability. The instruction address specifies a capability C for $X$ and the offset $i$ (the capability may be loaded on a hardware stack or in a register). The logical address of $X$ is mapped to a physical address through a **descriptor** in a mapping table: the descriptor gives the base $B$ and length $L$ of the memory segment containing the object. The base address $B$ could be an address in either primary or secondary memory (a flag, called the presence bit, indicates which); if the object is in secondary memory, it is moved to primary memory and $B$ updated. The process is given access to the memory address $B + i$ only if the offset is in range; that is, if $0 \leq i < L$. With the descriptor table, an object can be relocated without changing the capability; only the entry in the descriptor table requires updating.

FIGURE 4.23 Capability-based addressing.



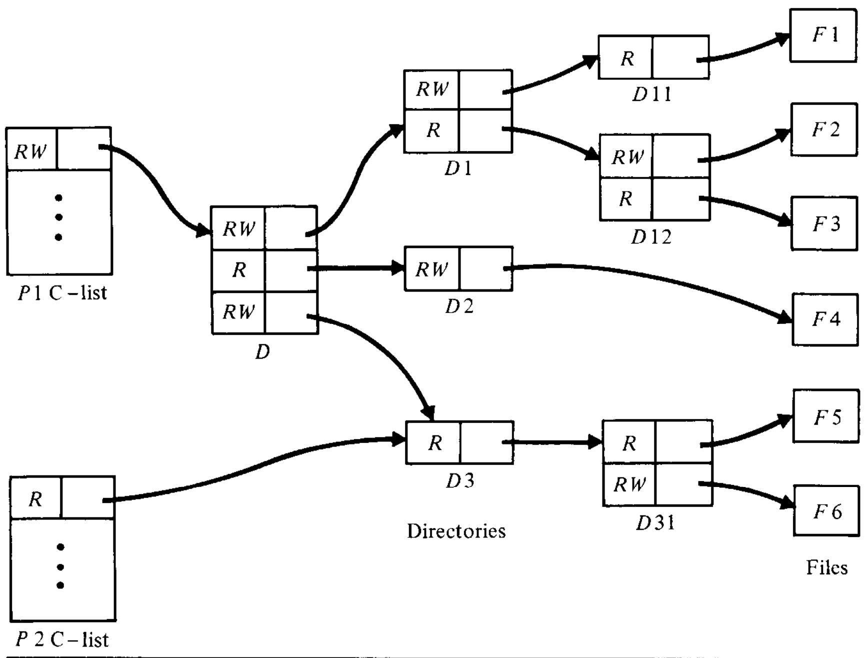
A capability does not usually index the mapping table directly. Instead it gives the unique name of the object, and this name is hashed to a slot in the table. This has special advantages when programs are shared or saved in long-term storage. Because the capabilities are invariant, the program can run in any domain at any time without modification. All variant information describing the location of the program and the objects referenced by it is kept in a central descriptor table under system control.

This property of invariance distinguishes capability systems from segmented virtual memory systems based on codewords and descriptors. In a descriptor-addressed system, a program accesses an object through a local address that points directly to an entry in the descriptor table. (Access rights are stored directly in the descriptors.) If local addresses refer to fixed positions in the descriptor table, sharing requires the participants to prearrange definitions (bindings) of local addresses. (See [Fabr71b,Fabr74] for a detailed discussion of capability-based addressing.)

The capabilities and hashed descriptor table need not significantly degrade addressing speed. Information in the descriptor table can be stored in high-speed associative registers, as is done for virtual memories. Table lookup time can be reduced by picking object names that hash to unique slots in the table (in SWARD, this is done by incrementing a counter until the value hashes to an empty slot).

With capability-based addressing, subjects sharing an object need only store a capability to an entry point in the structure, and different subjects can have different entry points and different access rights.

FIGURE 4.24 Shared subdirectory.



***Example:***

Figure 4.24 shows a process *P*1 with an *RW*-capability for the root *D* of a file directory. To read file *F*1, *P*1 would first get the *RW*-capability for directory *D*1 from *D*; it would then get the *RW*-capability for *D*11 from *D*1, and finally get an *R*-capability for *F*1 from *D*11. Similarly, *P*1 can acquire capabilities to access other files in the directory or to modify directories addressed by *RW*-capabilities (namely, *D*, *D*1, *D*11, and *D*3). Process *P*1 can share the subdirectory *D*3 with a process *P*2 by giving it an *R*-capability for the subdirectory *D*3; thus *P*2 can only access files *F*5 and *F*6; it cannot modify any of the directories. ■

With capability-based addressing, the protection state of a system is more naturally described with a directed graph such as shown in Figure 4.24 than with a matrix. The nodes of a graph correspond to the subjects and objects of the matrix; the edges to rights.

If the memory of the machine is not tagged, then the capabilities associated with a process or object must be stored in capability lists that are managed separately from other types of data. This means any object that has both capabilities and other types of data must be partitioned into two parts: a capability part and a data part. The system must keep these parts separate to protect the capabilities from unauthorized modification. This approach is used in most capability systems.

If the memory is tagged, capabilities can be stored anywhere in an object and used like addresses. Because their tags identify them as capabilities, the system can protect them from unauthorized modification. To address an object with a capability, the capability could be loaded either onto a hardware stack (in a stack architecture) or into a register (in a register architecture). The tagged memory approach simplifies addressing, domain switching, and storage management relative to partitioning (e.g., see [Dens80,Myer78]). Tagged memory, like capabilities, has its origins in the Rice Computer, the Basic Language Machine, and the Burroughs B5000 [Ilif72]. The SWARD machine and PSOS operating systems use tagged memories, as well as the capability-based machine designs of Dennis [Dens80] and Gehringer [Gehr79]. In SWARD, for example, a program can address local objects directly; capabilities are only used to address nonlocal objects. A capability can refer either to an entire object or to an element within an object. Indirect capabilities can be created to set up indirect address chains.

A system can use both capabilities and authorization lists—capabilities for currently active objects and authorization lists for inactive ones. Both MULTICS and SWARD, for example, provide a segmented name space through a single-level logical store. To access a segment in MULTICS, a process requests a descriptor (capability in SWARD) for it; this is granted provided the user associated with the process is listed in the authorization list for the target segment (access set in SWARD). Thereafter, the process can access the segment directly through the capability.

## 4.5.4 Revocation

Capabilities are easy to copy and disseminate, especially when they are stored in a tagged memory rather than in C-lists. This facilitates sharing among procedures or processes. There is, however, a drawback to this if objects are owned, and the owners can revoke privileges. If all access rights to an object are stored in a single

FIGURE 4.25 Revocation of rights with indirection.

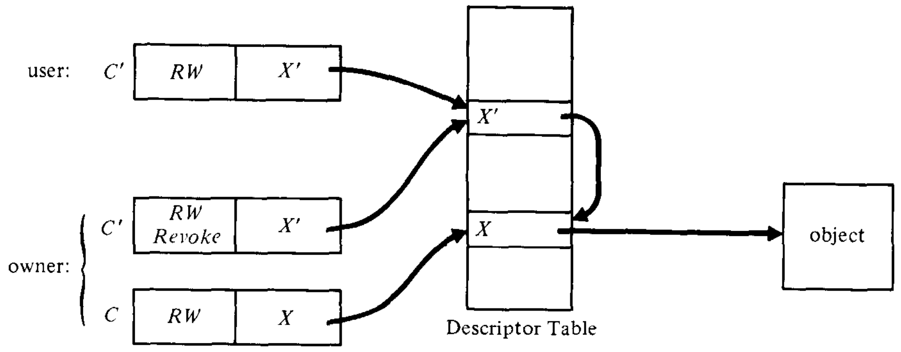
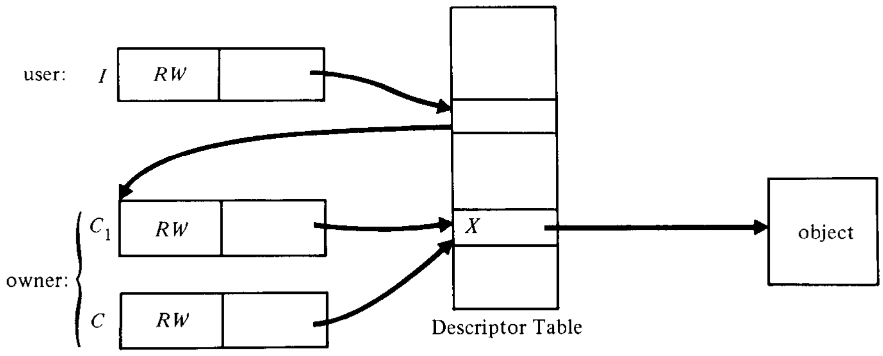


FIGURE 4.26 Revocation of rights with indirect capability in SWARD.



authorization list, it is relatively simple to purge them. But if they are scattered throughout the system, revocation could be more difficult.

Redell [Rede74] proposed a simple solution to this problem based on indirect addressing. The owner of an object $X$ with capability $C$ creates a capability $C'$ with name $X'$. Rather than pointing to the object directly, the entry for $X'$ in the descriptor table points to the entry for $X$. The owner grants access to $X$ by giving out copies of the capability $C'$ (see Figure 4.25). If the owner later revokes $C'$, the entry for $X'$ is removed from the descriptor table, breaking the link to $X$. The indirect capabilities of SWARD can also be used for revocation. Here the user is given an indirect capability $I$ that points to a copy $C_1$ of the owner's capability for $X$. The owner revokes access to $X$ by changing $C_1$ (see Figure 4.26).

## 4.5.5 Locks and Keys

Locks and keys combine aspects of list-oriented and ticket-oriented mechanisms. Associated with each object $x$ is a list of locks and access rights:

$$\begin{array}{l} L_1, r_1 \\ L_2, r_2 \\ \cdot \\ \cdot \\ \cdot \\ L_n, r_n\,. \end{array}$$

A subject $s$ is given a key $K_i$ to lock $L_i$ if $s$ has $r_i$-access to $x$; that is, if $A[s, x] = r_i$. A lock list, therefore, represents a column of the access matrix, where identical (nonempty) entries of the column can be represented by a single pair $(L_i, r_i)$. A key for an object represents a form of capability in which access is granted only if the key matches one of the locks in the object's lock list. The owner of an object can revoke the access rights of all subjects sharing a key $K_i$ by deleting the entry for $L_i$ in the lock list. Typically $n = 1$; that is, an object contains a single lock. In this case, a key is like an indirect capability, because the owner can revoke the key by changing the lock.

This method of protection resembles the "storage keys" used in IBM System/360 [IBM68]; the program status word of a process specifies a 4-bit key that must match a lock on the region of memory addressed by the process.

The ASAP file maintenance system designed at Cornell uses locks and keys [Conw72]. Each field in a record has associated with it one of eight possible classes (locks). Users are assigned to one or more of the classes (keys), and can only access those fields for which they have a key. There is also associated with each user a list of operations (e.g., *Update, Print*) the user is allowed to perform, and a set of data-dependent access restrictions. Whereas the data-independent restrictions are enforced at compile-time, the data-dependent restrictions are enforced at run-time.

Encryption is another example of a lock and key mechanism. Encrypting data places a lock on it, which can be unlocked only with the decryption key. Gifford [Giff82] has devised a scheme for protecting objects with encryption, which he calls **cryptographic sealing**. Let $X$ be an object encrypted with a key $K$; access to $X$, therefore, requires $K$. Access to $K$ can be controlled by associating an **opener** $R$ with $X$. Openers provide different kinds of sharing, three of which are as follows:

1. **OR-Access**, where $K$ can be recovered with any $D_i$ in a list of $n$ deciphering transformations $D_1, \dots, D_n$. Here the opener $R$ is defined by the list
$$R = (E_1(K), E_2(K), \dots, E_n(K)),$$
where $E_i$ is the enciphering transformation corresponding to $D_i$. Because $K$ is separately enciphered under each of the $E_i$, a process with access to any one of the $D_i$ can present $D_i$ to obtain $K$. The opener is thus like the "keys record" in Gudes's scheme described in Section 3.7.1 (see Figure 3.19).
2. **AND-Access**, where every $D_i$ in a list of deciphering transformations $D_1, \dots,$ $D_n$ must be present to recover $K$. Here the opener $R$ is defined by
$$R = E_n\big(E_{n-1}(\dots E_2(E_1(K))\dots)\big).$$
Clearly, every $D_i$ must be present to obtain $K$ from the inverse function
$$D_1\big(D_2(\dots D_{n-1}(D_n(R))\dots)\big) = K.$$
3. **Quorum-Access**, where $K$ can be recovered from any subset of $t$ of the $D_i$ in a list $D_1, \dots, D_n$. Here $R$ is defined by the list
$$R = (E_1(K_1), E_2(K_2), \dots, E_n(K_n)),$$
where each $K_i$ is a shadow of $K$ in a $(t, n)$ threshold scheme (see Section 3.8).

The different types of access can be combined to give even more flexible forms of sharing. The scheme also provides mechanisms for constructing submaster keys and indirect keys (that allow keys to be changed), and providing checksums in encrypted objects (for authentication—see Section 3.4). It can be used with both single-key and public-key encryption.

### 4.5.6 Query Modification

A high-level approach to security may be taken in **query-processing systems** (also called transaction-processing systems). The commands (queries) issued by a user are calls on a small library of transaction programs that perform specific operations, such as retrieving and updating, on a database. The user is not allowed to write, compile, and run arbitrary programs. In such systems, the only programs allowed to run are the certified transaction programs.

A user accesses a set of records with a **query** of the form $(f, T, E)$, where $f$ is an operation, $T$ is the name of a table, and $E$ is a logical expression identifying a group of records in $T$.

***Example:***
An example of an expression is $E$ = "*Sex = Female*". A request to retrieve this group of records from a table *Student* is specified by the query:

*Retrieve, Student, (Sex = Female)* . ■

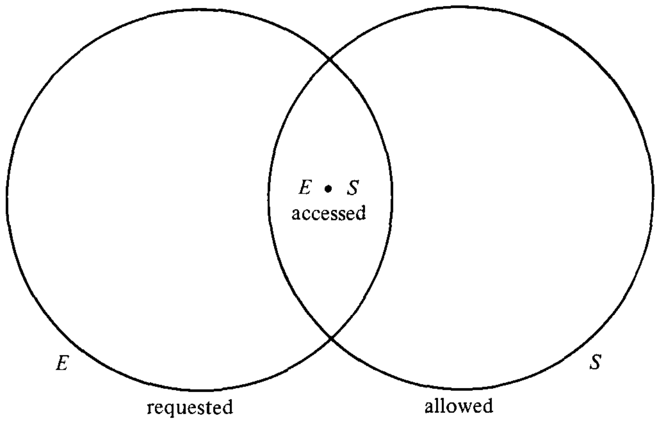
Stonebraker and Wong [Ston74] proposed an access control mechanism based on **query modification**. Associated with each user is a list with entries of the form $(T, R)$, where $T$ is the name of a table and $R$ is a set of access restrictions on $T$. The list is similar to a capability list in that it defines a user's access rights to the database. Each access restriction is of the form $(f, S)$, where $S$ is an expression identifying a subset of $T$; it authorizes the user to perform operation $f$ on the subset defined by $S$. If the user poses the query $(f, T, E)$, the transaction program modifies $E$ according to the expression $S$; it then proceeds as if the user had actually presented a formula $(E \cdot S)$, where "$\cdot$" denotes logical and (see Figure 4.27).

***Example:***
If a user is permitted to retrieve only the records of students in the department of computer science, then $S$ = "*Dept = CS*", and the preceding request would be transformed to:

*Retrieve, Student, (Sex = Female) · (Dept = CS)* . ■

Query modification may be used in systems with different underlying structures. Stonebraker and Wong developed it for the INGRES system, a relational database management system; it is also used in the GPLAN system [Cash76], a network-based database system designed around the CODASYL Data Base Task Group report.

FIGURE 4.27 Query modification.



The technique has the advantage of being conceptually simple and easy to implement, yet powerful enough to handle complex access constraints. It is, however, a high-level mechanism applicable only at the user interface. Lower-level mechanisms are needed to ensure the transaction programs do not violate their constraints, and to ensure users cannot circumvent the query processor.

## 4.6 VERIFIABLY SECURE SYSTEMS

The presence of protection mechanisms does not guarantee security. If there are errors or design flaws in the operating system, processes may still be able to acquire unauthorized access to objects or bypass the protection mechanisms. For example, a user may be able to bypass an authorization list for a file stored on disk by issuing I/O requests directly to the disk.

In the 1960s, the Systems Development Corporation (SDC) developed an approach for locating security flaws in operating systems [Lind75]. The methodology involves generating an inventory of suspected flaws called "flaw hypotheses", testing the hypotheses, and generalizing the findings to locate similar flaws. SDC applied the technique to locate and repair flaws in several major systems (see also [Hebb80] for a more recent application).

Most flaws satisfy certain general patterns; for example, a global variable used by the supervisor is tampered with between calls, and the supervisor does not check the variable before use. The University of Southern California Information Sciences Institute (ISI) has developed tools (some automatic) for finding these error patterns in operating systems [Carl75].

Penetration analysis (sometimes called a "tiger team" approach) has helped locate security weaknesses. But like program testing, it does not prove the absence of flaws.

In general, it is not possible to prove an arbitrary system is secure. The reason is similar to the reason we cannot prove programs halt, and is addressed in Section 4.7. But just as it is possible to write verifiably correct programs (e.g., the program "$i := 7 + 10$" always halts and satisfies the post-condition "$i = 17$"), it is possible to build provably secure systems. The key is to integrate the verification of a system into its specification, design, and implementation; that is described in the subsections that follow. Neumann [Neum78] believes the approach has led to systems with fewer flaws, but suggests combining it with penetration analysis to further strengthen the security of a system. Again, an analogy with program development holds; we would not put into production a verified but untested air traffic control program.

Even with advanced technology for developing and verifying systems, it is unlikely systems will be absolutely secure. Computer systems are extremely complex and vulnerable to many subtle forms of attack.

We shall first examine two techniques for structuring systems that aid verification, and then examine the verification process itself.

### 4.6.1 Security Kernels

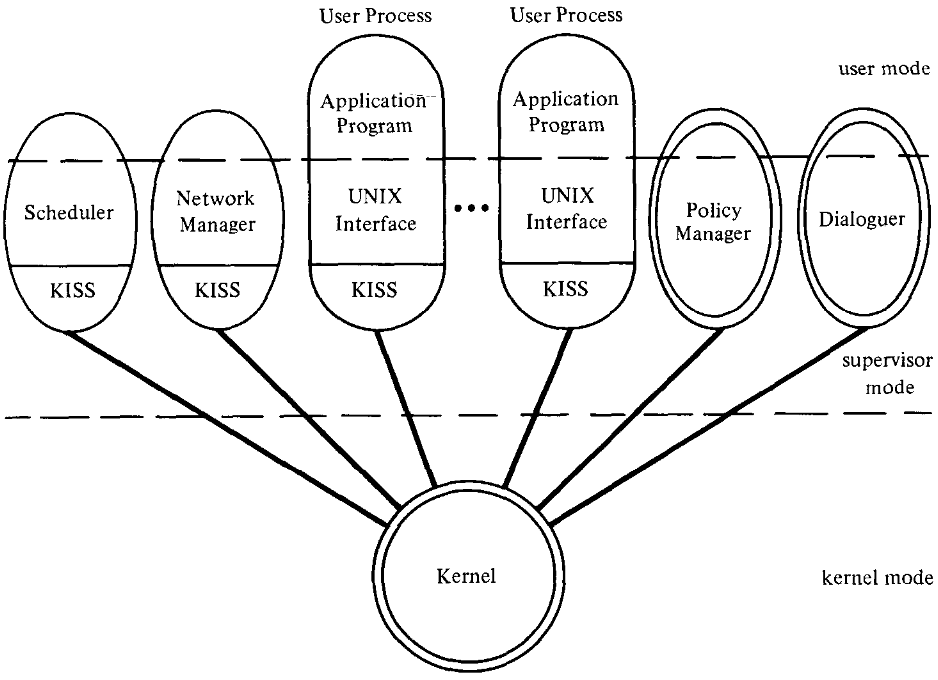
The objective is to isolate the access checking mechanisms in a small system nucleus responsible for enforcing security. The nucleus, called a **security kernel**, mediates all access requests to ensure they are permitted by the system's security policies. The security of the system is established by proving the protection policies meet the requirements of the system, and that the kernel correctly enforces the policies. If the kernel is small, the verification effort is considerably less than that required for a complete operating system.

The concept of security kernel evolved from the **reference monitor** concept described in the Anderson report [Ande72], and was suggested by Roger Schell. A reference monitor is an abstraction of the access checking function of object monitors [GrDe72] (see Section 4.1.1).

Several kernel-based systems have been designed or developed, including the MITRE security kernel for the DEC PDP-11/45 [Schi75,Mill76]; MULTICS with AIM [Schr77]; the MULTICS-based system designed at Case Western Reserve [Walt75]; the UCLA Data Secure UNIX system (DSU) for the PDP-11/45 and PDP-11/70 [Pope79]; the UNIX-based Kernelized Secure Operating System (KSOS) developed at Ford Aerospace for the PDP-11/70 (KSOS-11) [McCa79,Bers79] and at Honeywell for a Honeywell Level 6 machine (KSOS-6 or SCOMP) [Broa76]; and Kernelized VM/370 (KVM/370) developed at the System Development Corporation [Gold79].

With the exception of UCLA Secure UNIX, these systems were all developed to support the Department of Defense **multilevel security** policy described in the next chapter. Informally, this policy states that classified information must not be accessible to subjects with a lower security clearance. This means, for example, a user having a *Secret* clearance must not be able to read from *Top Secret* files

FIGURE 4.28 UCLA secure UNIX architecture.



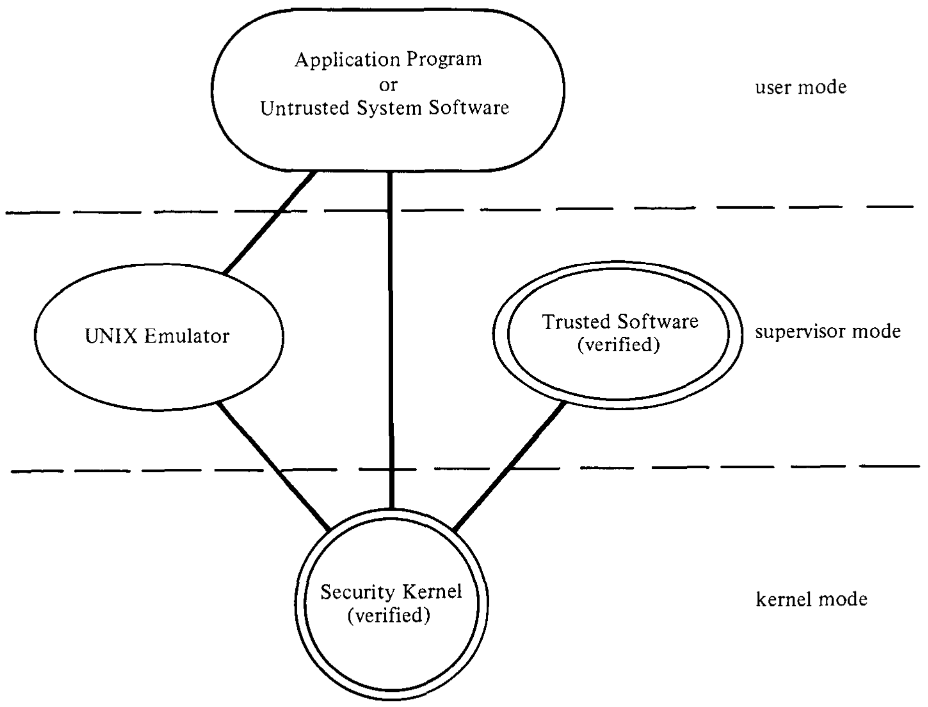
(i.e., "read up") or write *Secret* information in *Confidential* or *Unclassified* files (i.e., "write down").

We shall briefly describe UCLA Secure UNIX and KSOS-11. Both systems support a UNIX interface and run on PDP-11 hardware; both exploit the three execution modes of the PDP-11: kernel (highest privilege), supervisor, and user (least privilege). But different strategies have been used to structure and develop the two systems.

Figure 4.28 shows the architecture of the UCLA Secure UNIX system. Each user process runs in a separate protection domain, and is partitioned into two virtual address spaces. One address space contains the user (application) program and runs in user mode; the other contains a UNIX interface and runs in supervisor mode. The UNIX interface consists of a scaled down version of the standard UNIX operating system and a Kernel Interface SubSystem (KISS) that interfaces with the kernel.

The protection domain of a process is represented by a capability list managed by the kernel. The kernel enforces the protection policy represented by the capabilities, but does not alter the protection data. The power to grant and revoke capabilities is invested in a separate policy manager that manages the protection policies for shared files and kernel objects (processes, pages, and devices). Pro-

FIGURE 4.29 KSOS-11 architecture.



cesses cannot directly pass capabilities. The policy manager relies on a separate process (called a "dialoguer") to establish a secure connection between a user and terminal. The security policies and mechanisms of the system, therefore, are invested in the kernel, the policy manager, and the dialoguer (shown highlighted in Figure 4.28).

The kernel supports four types of objects: capabilities, processes, pages, and devices. It does not support type extensibility through abstract data types. Capabilities are implemented in software. The kernel is small (compared with other kernels), consisting of less than 2000 lines of code. The system architecture also includes a resource scheduler and network manager for the ARPANET.

Figure 4.29 shows the architecture of KSOS. Like the UCLA kernel, the KSOS kernel is at the lowest level and runs in kernel mode. But the KSOS kernel is substantially larger than the UCLA kernel, providing more operating system functions, file handling, and a form of type extension (the kernel is closer to a complete operating system than a simple reference monitor). The kernel enforces both an access control policy and the multilevel security policy.

A UNIX emulator and a "trusted" portion of nonkernel system software run in supervisor mode and interface directly with the kernel. The emulator translates system calls at the UNIX interface into kernel calls. Untrusted (nonverified) system software and application programs run in user mode and interface with the UNIX emulator or directly with the kernel. They can communicate with trusted processes through interprocess communication messages provided by the kernel. The trusted software consists of support services such as login, the terminal interface, and the telecommunications interface; and security mechanisms for applications with policies that conflict with the multilevel security policies of the kernel (see Section 5.6.3 for an example). Trusted processes are at least partially verified or audited.

Security kernels have also been used to structure database systems. Downs and Popek [Down77] describe a database system that uses two security kernels: a "kernel input controller", which processes user requests at the logical level, and a "base kernel", which accesses the physical representation of the data. All security related operations are confined to the kernels. Separate (nonverified) data management modules handle the usual data management functions, such as selecting access methods, following access paths, controlling concurrent accesses, and formatting data. The base kernel is the only module allowed to access the database.

### 4.6.2 Levels of Abstraction

The basic idea is to decompose a system (or security kernel) into a linear hierarchy of abstract machines, $M_0, \ldots, M_n$. Each abstract machine $M_i$ $(0 < i \leq n)$ is implemented by a set of abstract programs $P_{i-1}$ running on the next lower level machine $M_{i-1}$. Thus the programs at level $i$ depend only on the programs at levels $0, \ldots, i-1$. They are accessible at level $i+1$, but may be invisible at levels above that. The system is verified one level at a time, starting with level 0; thus verification of each level can proceed under the assumption that all lower levels are correct. The general approach originated with Dijkstra [Dijk68], who used it to structure the "THE" (Technische Hoogeschule Eindhoven) multiprogramming system.

The Provably Secure Operating System (PSOS), designed at SRI under the direction of Peter Neumann, has a design hierarchy with 17 levels of abstraction (see Figure 4.30) [Feie79,Neum80]. PSOS is a capability-based system supporting abstract data types. Because capabilities provide the basic addressing and protection mechanisms of the system, they are at the lowest level of abstraction. All levels below virtual memory (level 8) are invisible at the user interface, except for capabilities and basic operations (level 4). In the first implementation, levels 0 through 8 are expected to be implemented in hardware (or microcode) along with a few operations at the higher levels.

Imposing a loop-free dependency structure on a system design is not always straightforward (see [Schr77]). Some abstractions—for example, processes and memory—are seemingly interdependent. For example, the process manager depends on memory for the storage of process state information; the virtual memory manager depends on processes for page swaps. Because neither processes nor memory can be entirely below the other, these abstractions are split into "real"

FIGURE 4.30 PSOS design hierarchy.

| Level | Abstractions |
|---|---|
| 16 | Command interpreter |
| 15 | User environments and name space |
| 14 | User input/output |
| 13 | Procedure records |
| 12 | User processes and visible input/output |
| 11 | Creation and deletion of user objects |
| 10 | Directories |
| 9 | Abstract data types |
| 8 | Virtual memory (segmentation) |
| 7 | Paging |
| 6 | System processes and system input/output |
| 5 | Primitive input/output |
| 4 | Basic arithmetic and logical operations |
| 3 | Clocks |
| 2 | Interrupts |
| 1 | Real memory (registers and storage) |
| 0 | Capabilities |

and "virtual" components. In PSOS, real memory is below processes, but a few fixed (real) system processes with real memory are below virtual memory. User (virtual) processes are above virtual memory. (In PSOS, the design hierarchy is also separated from the implementation hierarchy.)

### 4.6.3 Verification

Verification involves developing formal specifications for a system, proving the specifications satisfy the security policies of the system, and proving the implementation satisfies the specifications.

SRI researchers have developed a Hierarchical Design Methodology (HDM) to support the development of verifiable systems [Neum80,Robi79]. HDM has been used in the design of PSOS and the development of KSOS.

HDM decomposes a system into a hierarchy of abstract machines as described in the preceding section. Each abstract machine is specified as a module in the language SPECIAL (SPECIfication and Assertion Language). A module is defined in terms of an internal state space (abstract data structures) and state transitions, using a specification technique introduced by Parnas [Parn72,Pric73]. The states and state transitions are specified by two types of functions:

1. ***V*-functions**, that give the value of a state variable (**primitive** *V*-function) or a value computed from the values of state variables (**derived** *V*-function). The initial value of a primitive *V*-function is specified in the module definition.

FIGURE 4.31 Specification of *stack* module.

```
module stack;
 Vfun top() : integer;
        "primitive V-function giving the index
          of the top of the stack"
     hidden;
     initially: top = 0;
     exceptions: none;
  Vfun data(i: integer) : integer;
        "primitive V-function giving the value
          of the ith element of the stack"
      hidden;
      initially: ∀i (data(i) = undefined);
      exceptions: (i < 0) or (i > size);
  Vfun empty() : boolean;
        "derived V-function giving status of stack"
      derived: if top = 0 then true else false;
  Ofun push(x: integer);
        "push element x onto top of stack"
      exceptions: top ≥ size;
      effects: 'top = top + 1;
               'data('top) = x;
               ∀j ≠ 'top, 'data(j) = data(j)
end stack
```

2. *O*-**functions**, that perform an operation changing the state. State transitions (called **effects**) are described by assertions relating new values of primitive *V*-functions to their prior values. A state variable cannot be modified unless its corresponding primitive *V*-function appears in the effects of an *O*-function.

A function can both perform an operation and return a value, in which case it is called an ***OV*-function**. Functions may be either **visible** or **invisible** (**hidden**) outside the module. Primitive *V*-functions are always hidden.

The specification of a function lists **exceptions** which state abnormal conditions under which the function is not defined. The implementation must ensure the code for the function is not executed when these conditions are satisfied (e.g., by making appropriate tests).

***Example:***
Figure 4.31 shows a specification of a module for a subset of the *stack* module shown in Figure 4.19. A *V*-function name preceded by a prime (′) refers to the value of the *V*-function after the transition caused by the effects of an *O*-function; the unprimed name refers to its original value. Specification of the *pop* operation is left as an exercise. ■

HDM structures the development of a system into five stages:

S0. **Interface Definition.** The system interface is defined and decomposed into a set of modules. Each module manages some type of system object (e.g., segments, directories, processes), and consists of a collection of *V*-, and *O*-functions. (Formal specifications for the functions are deferred to Stage S2.) The security requirements of the system are formulated. For PSOS, these requirements are described by two general principles:

a. **Detection Principle:** There shall be no unauthorized acquisition of information.

b. **Alteration Principle:** There shall be no unauthorized alteration of information.

These are low-level principles of the capability mechanism; each type manager uses capabilities to enforce its own high-level policy.

S1. **Hierarchical Decomposition.** The modules are arranged into a linear hierarchy of abstract machines $M_0, \dots, M_n$ as described in Section 4.6.1. The consistency of the structure and of the function names is verified.

S2. **Module Specification.** Formal specifications for each module are developed as described. Each module is verified to determine if it is self-consistent and satisfies certain global assertions. The basic security requirements of the system are represented as global assertions and verified at this stage. For PSOS, the alteration and detection principles are specified in terms of capabilities providing read and write access.

S3. **Mapping Functions.** A mapping function is defined to describe the state space at level $i$ in terms of the state space at level $i - 1$ $(0 < i \leq n)$. This is written as a set of expressions relating the *V*-functions at level $i$ to those at level $i - 1$. Consistency of the mapping function with the specifications and the hierarchical decomposition is verified.

S4. **Implementation.** Each module is implemented in hardware, microcode, or a high-level language, and the consistency of the implementation with the specifications and mapping function is verified. Implementation proceeds one level at a time, from the lowest level to the highest. Each function at level $i$ is implemented as an abstract program which runs on machine $M_{i-1}$.

Each abstract program is verified using Floyd's [Floy67] inductive-assertion method, which is extended to handle *V*- and *O*-function calls. Entry and exit assertions are constructed for each program from the specifications. A program is proved correct by showing that if the entry assertions hold when the program begins execution, then the exit assertions will hold when the program terminates, regardless of the execution path through the program. The proof is constructed by inserting intermediate assertions into the program; these define entry and exit conditions for simple paths of the program. A simple path is a sequence of statements with a single entry and a single exit. For each simple path $S$ with entry assertion $P$ and exit assertion $Q$, a verification condition ($VC$) in the form of an implication $P' \supset Q'$ is derived by transforming $P$ and $Q$ to reflect the effects of executing $S$. The program is proved (or disproved) correct by proving (or disproving) the *VC*s are theorems. The proof techniques are described in [Robi77]. Program proving is studied in Section 5.5.

Researchers at SRI have developed several tools [Silv79] to support HDM and its specification language SPECIAL. These include tools to check the consistency of specifications and mappings, the Boyer-Moore theorem-prover [Boye79], and verification condition generators for several programming languages including FORTRAN [Boye80] and Pascal [Levi81].

At UCLA, Popek and others have adopted a two-part strategy to develop and verify the UCLA security kernel [Pope78,Walk80]. The first part involves developing successively more detailed specifications of the kernel until an implementation is reached:

1. **Top-Level Specifications**, which give a high-level, intuitive description of the security requirements of the system.
2. **Abstract-Level Specifications**, which give a more refined description.
3. **Low-Level Specifications**, which give a detailed description.
4. Pascal **Code** satisfying the specifications.

The specifications at each level are formulated in terms of an abstract machine with states and state transitions satisfying the following general requirements:

1. Protected objects may be modified only by explicit request.
2. Protected objects may be read only by explicit request.
3. Specific access to protected objects is permitted only when the recorded protection data allows it; i.e., all accesses must be authorized.

These requirements are essentially the same as the Detection and Alteration principles of PSOS.

The second part involves verifying that the Pascal implementation satisfies the low-level specifications, and verifying that the specifications at each level are consistent with each other. Note that whereas for PSOS the implementation is constructed from the hierarchy of specifications and mapping functions, for UCLA Secure UNIX it represents a refinement of the bottom-level specifications. Verification of UCLA Secure UNIX was assisted by the AFFIRM verification system and its predecessor XIVUS, developed at ISI.

For further reading on specification and verification systems, see Cheheyl, Gasser, Huff, and Millen [Cheh81]. This paper surveys four systems, including HDM and AFFIRM. The other two systems are Gypsy, developed at the University of Texas at Austin, and the Formal Development Methodology (FDM) and its specification language Ina Jo, developed at SDC.

Rushby [Rush81] has proposed a new approach to the design and verification of secure systems. His approach is based on a distributed system architecture that provides security through physical separation of subjects where possible, trusted modules that control the communication channels among subjects, and a security kernel that enforces the logical separation of subjects sharing the same physical resources. Verification involves showing that the communication channels are used in accordance with the security policies of the system, and that the subjects become completely isolated when these channels are cut. The latter part, called "proof of separability," involves showing that a subject cannot distinguish its actual environment on a shared machine from its abstract environment on a private virtual machine. The approach is particularly suited for the development of secure networks that use encryption to protect data transmitted over the channels.

## 4.7 THEORY OF SAFE SYSTEMS

Harrison, Ruzzo, and Ullman [Harr76] studied the feasibility of proving properties about a high-level abstract model of a protection system. They used as their model the access-matrix model described in Section 4.1; thus, the state of a system is described by an access matrix $A$, and state transitions by commands that create and destroy subjects and objects, and enter and delete rights in $A$. They defined an unauthorized state $Q$ to be one in which a generic right $r$ could be **leaked** into $A$; the right would be leaked by a command $c$ that, when run in state $Q$, would execute a primitive operation entering $r$ into some cell of $A$ not previously containing $r$. An initial state $Q_0$ of the system is defined to be **safe** for $r$ if it cannot derive a state $Q$ in which $r$ could be leaked.

Leaks are not necessarily bad, as any system that allows sharing will have many leaks. Indeed, many subjects will intentionally transfer (leak) their rights to other "trustworthy" subjects. The interesting question is whether transfer of a right $r$ violates the security policies of the system. To answer this question, safety for $r$ is considered by deleting all trustworthy subjects from the access matrix.

Proving a system is safe does not mean it is secure—safety applies only to the abstract model. To prove security, it is also necessary to show the system correctly implements the model; that is, security requires both safety and correctness. Thus, safety relates to only part of the verification effort described in the preceding section.

Harrison, Ruzzo, and Ullman showed safety is undecidable in a given arbitrary protection system. Safety is decidable, however, if no new subjects or objects can be created. They also showed safety is decidable in a highly constrained class of systems permitting only "mono-operational" commands, which perform at most one elementary operation. We first review their results for mono-operational systems, and then review their results for general systems.

We then consider the prospects for developing a comprehensive theory of protection (or even a finite number of such theories) sufficiently general to enable proofs or disproofs of safety. Not surprisingly, we can show there is no decidable theory adequate for proving all propositions about safety.

These results must be interpreted carefully. They are about the fundamental limits of our abilities to prove properties about an abstract model of a protection system. They do not rule out constructing individual protection systems and proving they are secure, or finding practical restrictions on the model that make safety questions tractable. Yet, these results do suggest that systems without severe restrictions on their operation will have security questions too expensive to answer. Thus we are forced to shift our concern from proving arbitrary systems secure to designing systems that are provably secure. PSOS, KSOS, UCLA Unix, and the other systems described in the last section are provably secure only because they were designed to satisfy specified security requirements. Their security was continually verified at each stage in the development of the system.

We conclude this chapter with a discussion of safety in systems constrained by the "take-grant" graph rules. For such systems, safety is not only decidable, but decidable in time linear in the size of the protection graph.

### 4.7.1 Mono-Operational Systems

A protection system is **mono-operational** if each command performs a single primitive operation. Harrison, Ruzzo, and Ullman prove the following theorem:

***Theorem 4.1:***
There is an algorithm that decides whether a given mono-operational system and initial state $Q_0$ is safe for a given generic right $r$.

***Proof:***
We will show that only a finite number of command sequences need be checked for the presence of a leak. Observe first that we can ignore command sequences containing **delete** and **destroy** operators, since commands only check for the presence of rights, not their absence. Thus, if a leak occurs in a sequence containing these commands, then the leak would occur in one without them.

Observe next that we can ignore command sequences containing more than one **create** operator. The reason is that all commands that enter or check for rights in new positions of the access matrix can be replaced with commands which enter or check for rights in existing positions of the matrix; this is done simply by changing the actual parameters from the new subjects and objects to existing subjects and objects. It is necessary, however, to retain one **create subject** command in case the initial state has no subjects (to ensure that the matrix has at least one position in which to enter rights).

This means the only command sequences we need consider are those consisting of **enter** operations and at most one **create subject** operation.

Now, the number of distinct **enter** operations is bounded by $gn_sn_o$, where $g$ is the number of generic rights, $n_s = |S_0| + 1$ is the number of subjects, and $n_o = |O_0| + 1$ is the number of objects. Because the order of **enter** operations in a command sequence is not important, the number of command sequences that must be inspected is, therefore, bounded by:

$$2^{gn_sn_o+1} \; . \quad \blacksquare$$

Although it is possible to construct a general decision procedure to determine the decidability of arbitrary mono-operational systems, Harrison, Ruzzo, and Ullman show the problem to be **NP**-complete and thus intractable. They note, however, that by using the technique of "dynamic programming" (e.g., see [Aho74]), an algorithm polynomial in the size of the initial matrix can be devised for any given system.

Most systems are not mono-operational, as illustrated by the examples of Section 4.1.2. Nevertheless, the results are enlightening in that they show the safety question for general systems will, at best, be extremely difficult.

### 4.7.2 General Systems

Harrison, Ruzzo, and Ullman show that the general safety problem is undecidable. To prove this result, they show how the behavior of an arbitrary Turing machine (e.g., see [Aho74,Mins67]) can be encoded in a protection system such that leakage of a right corresponds to the Turing machine entering a final state. Therefore, if safety is decidable, then so is the halting problem. Because the halting problem is undecidable, the safety problem must also be undecidable.

A **Turing machine** $T$ consists of a finite set of **states** $K$ and a finite set of **tape symbols** $\Gamma$, which includes a **blank** b̸. The **tape** consists of an infinite number of cells numbered 1, 2, . . . , where each cell is initially blank. A **tape head** is always positioned at some cell of the tape.

The **moves** of $T$ are specified by a function $\delta$: $K \times \Gamma \rightarrow K \times \Gamma \times \{L, R\}$. If $\delta(q, X) = (p, Y, R)$ and $T$ is in state $q$ scanning symbol $X$ with its tape head at cell $i$, then $T$ enters state $p$, overwrites $X$ with $Y$, and moves its tape head right one cell (i.e., to cell $i + 1$). If $\delta(q, X) = (p, Y, L)$ the same thing happens, but the tape head is moved left one cell (unless $i = 1$).

$T$ begins in an initial state $q_0$, with its head at cell 1. There is also a final state $q_f$ such that if $T$ enters $q_f$ it halts. The "halting problem" is to determine whether an arbitrary Turing machine ever enters its final state. It is well-known that this problem is undecidable. We shall now show the safety problem is also undecidable.

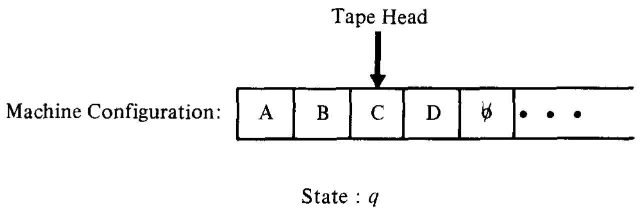
***Theorem 4.2:***
It is undecidable whether a given state of a given protection system is safe for a given generic right.

***Proof:***
Let $T$ be an arbitrary Turing machine. We shall first show how to encode the state of $T$ as an access matrix in a protection system, and then show how to encode the moves of $T$ as commands of the system. The tape symbols will be represented as generic access rights, and the tape cells as subjects of the matrix.

Suppose that $T$ is in state $q$. At that time, $T$ will have scanned a

Previous Chapter Previous Page Home Next Page Next Chapter

FIGURE 4.32 Encoding of Turing machine.



|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | A | *Own* |  |  |
| 2 |  | B | *Own* |  |
| 3 |  |  | C $q$ | *Own* |
| 4 |  |  |  | D *End* |

finite number $k$ of cells, and cells $k + 1$, $k + 2$, . . . will be blank. State $q$ is represented as an access matrix $A$ with $k$ subjects and $k$ objects (all subjects) such that:

1. If cell $s$ of the tape contains the symbol $X$, then $A[s, s]$ contains the right $X$.
2. $A[s, s + 1]$ contains the right *Own* ($s = 1, \ldots, k - 1$) (this induces an ordering on the subjects according to the sequence of symbols on the tape).
3. $A[k, k]$ contains the right *End* (this signals the current end of the tape).
4. If the tape head is positioned at cell $s$, then $A[s, s]$ contains the right $q$.

Figure 4.32 shows the access matrix corresponding to a Turing machine in state $q$ whose first four cells hold ABCD, with the tape head at cell 3.

Note that the machine's tape is encoded along the diagonal of the matrix. It cannot be encoded along a row (or column) of the matrix because for a given subject $s$, there is no concept of a predecessor or successor of $s$. The only way of ordering the subjects is by giving each subject $s$ a right (such as *Own*) for another subject (denoted $s + 1$).

The initial state $q_0$ is represented as an access matrix with one subject $s_0$, corresponding to the first tape cell. The rights in $A[s_0, s_0]$ include $q_0$, ƀ (since all cells are blank), and *End*.

The moves of $T$ are represented as commands. A move $\delta(q, X) = (p, Y, L)$ is represented by a command that revokes the right $q$ from subject $s'$ and grants the right $p$ to subject $s$, where $s'$ represents the current position of the tape head, and $s$ represents the cell to the left of $s'$. The command, shown next, also substitutes the right $Y$ for the right $X$ in the cell represented by $s'$.

```
command $C_{qX}(s, s')$
  if
    $Own$ in $A[s, s']$ and
    $q$ in $A[s', s']$ and
    $X$ in $A[s', s']$
  then
    delete $q$ from $A[s', s']$
    delete $X$ from $A[s', s']$
    enter $Y$ into $A[s', s']$
    enter $p$ into $A[s, s]$
end .
```

A move $\delta(q, X) = (p, Y, R)$ is represented by two commands to handle the possibility that the tape head could move past the current end of the tape (in which case a new subject must be created). The specification of these commands is left as an exercise for the reader.

Now, if the Turing machine reaches its final state $q_f$, then the right $q_f$ will be entered into some position of the corresponding access matrix. Equivalently, if the Turing machine halts, then the right $q_f$ is leaked by the protection system. Because the halting problem is undecidable, the safety problem must, therefore, also be undecidable. ■

Theorem 4.2 means that the set of safe protection systems is not **recursive**; that is, it is not possible to construct an algorithm that decides safety for all systems. Any procedure alleged to decide safety must either make mistakes or get hung in a loop trying to decide the safety of some systems.

We can, however, generate a list of all unsafe systems; this could be done by systematically enumerating all protection systems and all sequences of commands in each system, and outputting the description of any system for which there is a sequence of commands causing a leak. This is stated formally by Theorem 4.3:

***Theorem 4.3:***
The set of unsafe systems is **recursively enumerable**. ■

We cannot, however, enumerate all safe systems, for a set is recursive if and only if both it and its complement are recursively enumerable.

Whereas the safety problem is in general undecidable, it is decidable for finite systems (i.e., systems that have a finite number of subjects and objects).

Harrison, Ruzzo, and Ullman prove, however, the following theorem, which implies that any decision procedure is intractable:

> ***Theorem 4.4:***
> The question of safety for protection systems without **create** commands is **PSPACE**-complete (complete in polynomial space). ■

This means that safety for these systems can be solved in polynomial time (time proportional to a polynomial function of the length of the description of the system) if and only if **PSPACE** = **P**; that is, if and only if any problem solvable in polynomial space is also solvable in polynomial time. Although the relationship between time and space is not well understood, many believe **PSPACE** $\neq$ **P** and exponential time is required for such problems (see Section 1.5.2). The proof of Theorem 4.4 involves showing any polynomial-space bounded Turing machine can be reduced in polynomial time to an initial access matrix whose size is polynomial in the length of the Turing machine input.

Harrison and Ruzzo [Harr78] considered **monotonic** systems, which are restricted to the elementary operations **create** and **enter** (i.e., there are no **destroy** or **delete** operators). Even for this highly restricted class of systems, the safety problem is undecidable.

## 4.7.3 Theories for General Systems

Denning, Denning, Garland, Harrison, and Ruzzo [Denn78] studied the implications of the decidability results for developing a theory of protection powerful enough to resolve safety questions. Before presenting these results, we shall first review the basic concepts of theorem-proving systems. Readers can skip this section without loss of continuity.

A **formal language** $L$ is a recursive subset of the set of all possible strings over a given finite alphabet; the members of $L$ are called **sentences**.

A **deductive theory** $T$ over a formal language $L$ consists of a set $A$ of **axioms**, where $A \subseteq L$, and a finite set of **rules of inference**, which are recursive relations over $L$. The set of **theorems** of $T$ is defined inductively as follows:

1. If $t$ is an axiom (i.e., $t \in A$), then $t$ is a theorem of $T$; and
2. If $t_1, \ldots, t_k$ are theorems of $T$ and $<t_1, \ldots, t_k, t> \in R$ for some rule of inference $R$, then $t$ is a theorem of $T$.

Thus, every theorem $t$ of $T$ has a **proof**, which is a finite sequence $<t_1, \ldots, t_n>$ of sentences such that $t = t_n$ and each $t_i$ is either an axiom or follows from some subset of $t_1, \ldots, t_{i-1}$ by a rule of inference. We write $T \vdash t$ to denote $t$ is a theorem of $T$ (is provable in $T$).

Two theories $T$ and $T'$ are said to be **equivalent** if they have the same set of theorems. Equivalent theories need not have the same axioms or rules of inference.

A theory $T$ is **recursively axiomatizable** if it has (or is equivalent to a theory with) a recursive set of axioms. The set of theorems of any recursively axiomatizable theory is recursively enumerable: we can effectively generate all finite sequences of sentences, check each to see if it is a proof, and enter in the enumeration the final sentence of any sequence that is a proof.

A theory is **decidable** if its theorems form a recursive set.

Because the set of safe protection systems is not recursively enumerable, it cannot be the set of theorems of a recursively axiomatizable theory. This means the set of all safe protection systems cannot be effectively generated by rules of inference from a finite (or even recursive) set of safe systems. (Note this does not rule out the possibility of effectively generating smaller, but still interesting classes of safe systems.) This observation can be refined, as we shall do, to establish further limitations on any recursively axiomatizable theory of protection.

A **representation of safety** over a formal language $L$ is an effective map $p \longrightarrow t_p$ from protection systems to sentences of $L$. We interpret $t_p$ as a statement of the safety of protection system $p$.

A theory $T$ is **adequate for proving safety** if and only if there is a representation $p \longrightarrow t_p$ of safety such that

$T \vdash t_p$ if and only if $p$ is safe.

Analogs of the classical Church and Gödel theorems for the undecidability and incompleteness of formal theories of arithmetic follow for formal theories of protection systems.

***Theorem 4.5:***
Any theory $T$ adequate for proving safety must be undecidable. ■

This theorem follows from Theorem 4.2 by noting that, were there an adequate decidable $T$, we could decide whether a protection system $p$ were safe by checking whether $T \vdash t_p$.

***Theorem 4.6:***
There is no recursively axiomatizable theory $T$ adequate for proving safety. ■

This theorem follows from Theorems 4.2 and 4.3. If $T$ were adequate and recursively axiomatizable, we could decide the safety of $p$ by enumerating simultaneously the theorems of $T$ and the set of unsafe systems; eventually, either $t_p$ will appear in the list of theorems or $p$ will appear in the list of unsafe systems, enabling us to decide the safety of $p$.

Theorem 4.6 shows that, given any recursively axiomatizable theory $T$ and any representation $p \longrightarrow t_p$ of safety, there is some system whose safety either is established incorrectly by $T$ or is not established when it should be. This result in itself is of limited interest for two reasons: it is not constructive (i.e., it does not show us how to find such a $p$); and, in practice, we may be willing to settle for inadequate theories as long as they are sound, that is, as long as they do not err by falsely establishing the safety of unsafe systems. Formally, a theory $T$ is **sound** if and only if $p$ is safe whenever $T \vdash t_p$. The next theorem overcomes the first limitation, showing how to construct a protection system $p$ that is unsafe if and only if $T \vdash t_p$; the idea is to design the commands of $p$ so that they can simulate a Turing machine that "hunts" for a proof of the safety of $p$; if and when a sequence of commands finds such a proof, it generates a leak. If the theory $T$ is sound, then such a protection system $p$ must be safe but its safety cannot be provable in $T$.

***Theorem 4.7:***
Given any recursively axiomatizable theory $T$ and any representation of safety in $T$, one can construct a protection system $p$ for which $T \vdash t_p$ if and only if $p$ is unsafe. Furthermore, if $T$ is sound, then $p$ must be safe, but its safety is not provable in $T$.

***Proof:***
Given an indexing $\{M_i\}$ of Turing machines and an indexing $\{p_i\}$ of protection systems, the proof of Theorem 4.2 shows how to define a recursive function $f$ such that

(a) $M_i$ halts iff $p_{f(i)}$ is unsafe.

Since $T$ is recursively axiomatizable and the map $p \longrightarrow t_p$ is computable, there is a recursive function $g$ such that

(b) $\mathrm{T} \vdash t_{p_i}$ iff $M_{g(i)}$ halts;

the Turing machine $M_{g(i)}$ simply enumerates all theorems of $T$, halting if $t_{p_i}$ is found. By the Recursion Theorem [Roge67], one can find effectively an index $j$ such that

(c) $M_j$ halts iff $M_{g(f(j))}$ halts.

Combining (a), (b), and (c), and letting $p = p_{f(j)}$, we get

(d) $T \vdash t_p$ iff $M_{g(f(j))}$ halts
iff $M_j$ halts
iff $p_{f(j)} = p$ is unsafe ,

as was to be shown.

Now, suppose $T$ is sound. Then $t_p$ cannot be a theorem of $T$ lest $p$ be simultaneously safe by soundness and unsafe by (d). Hence, $\mathrm{T} \nvdash t_p$, and $p$ is safe by (d) . ■

The unprovability of the safety of a protection system $p$ in a given sound theory $T$ does not imply $p$'s safety is unprovable in every theory. We can, for example, augment $T$ by adding $t_p$ to its axioms. But no matter how much we augment $T$, there will always exist another safe $p'$ whose safety is unprovable in the new theory $T'$. In other words, this abstract view shows that systems for proving safety are necessarily **incomplete**: no single effective deduction system can be used to settle all questions of safety.

We also considered theories for protection systems of bounded size. Although the safety question becomes decidable (Theorem 4.4), any decision procedure is likely to require enormous amounts of time. This rules out practical mechanical safety tests, but not the possibility that ingenious or lucky people might always be able to find proofs faster than any mechanical method. Unfortunately, we found even this hope ill-founded. If we consider reasonable proof systems in which we can decide whether a given string of symbols constitutes a proof in time polynomial in the string's length, then we have the following:

***Theorem 4.8:***
For the class of protection systems in which the number of objects and domains of access is bounded, safety (or unsafety) is polynomial verifiable by some reasonable logical system if and only if **PSPACE** = **NP**; that is, if and only if any problem solvable in polynomial space is solvable in nor.ieterministic polynomial time. ■

Many believe **PSPACE** ≠ **NP** (see Section 1.5.2).

### 4.7.4 Take-Grant Systems

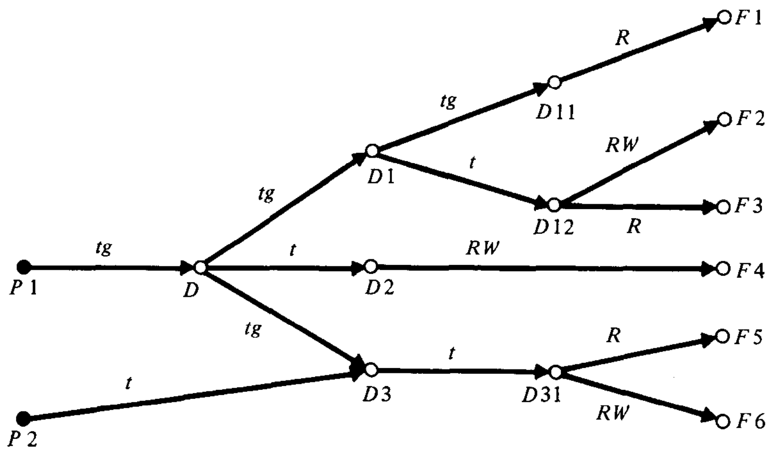
Jones, Lipton, and Snyder [Jone76b] introduced the **Take-Grant** graph model to describe a restricted class of protection systems. They showed that for such systems, safety is decidable even if the number of subjects and objects that can be created is unbounded. Furthermore, it is decidable in time linear in the size of the initial state. We shall describe these results, following their treatment in a survey paper by Snyder [Snyd81a].

As in the access-matrix model, a protection system is described in terms of states and state transitions. A protection state is described by a directed graph $G$, where the nodes of the graph represent the subjects and objects of the system (subjects are not objects in the take-grant model), and the edges represent rights. We shall let $(x, y)$ denote the set of access rights on an edge from $x$ to $y$, where $r \in (x, y)$ means that $x$ has right $r$ for $y$. If $x$ is a subject, then $(x, y)$ in $G$ is like $A[x, y]$ in the access matrix.

There are two special rights: **take** (abbreviated $t$), and **grant** (abbreviated $g$). If a subject $s$ has the right $t$ for an object $x$, then it can take any of $x$'s rights; if it has the right $g$ for $x$, then it can share any of its rights with $x$. These rights describe certain aspects of capability systems. If a subject has a capability to read from an object $x$ containing other capabilities, then it can take capabili'ies (rights) from $x$; similarly, if it has a capability to w.ite into $x$, it can grant capabilities to $x$.

***Example:***
Figure 4.33 shows the graph representation of the directory structure shown in Figure 4.24. We have used "●" to represent subject nodes, and "o" to represent object nodes. Note that $R$ and $W$ capabilities for a directory be-

FIGURE 4.33 Take-grant representation of directory structure.

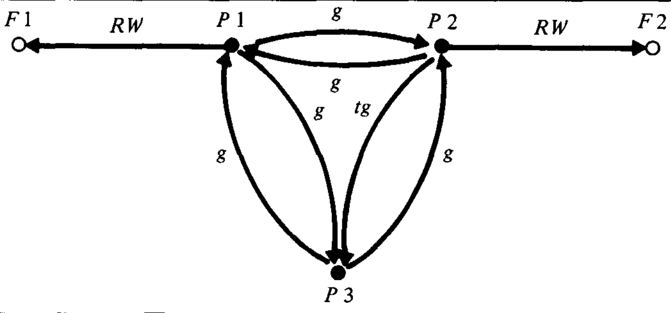


come take and grant rights, respectiıely, in the take-grant model, whereas *R* and *W* capabilities for files remain as *R* and *W* rights. ■

The take-grant model describes the **transfer of authority** (rights) in systems. It does not describe the protection state with respect to rights that cannot be transferred. Thus, it abstracts from the complete state only information needed to answer questions related to safety.

***Example:***
Figure 4.34 shows only part of the protection state of Figure 4.2. A process can grant rights for any of its owned files to any other process, so there is an edge labeled *g* connecting each pair of processes. But only process *P*2 is allowed to take rights from another process, namely its subordinate *P*3, so there is only one edge labeled *t*. Because rights for memory segments cannot

FIGURE 4.34 Take-grant graph for system shown in Figure 4.2.



Previous Chapter Previous Page Home Next Page Next Chapter

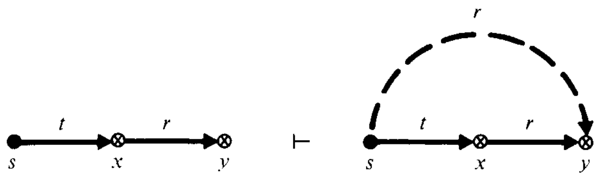
be granted along the $g$-edges (memory is not owned and the copy flag is not set), these rights are not shown. Consequently, the graph does not show $P2$ can take $P3$'s rights for memory segment $M3$ (as it did). ■

State transitions are modeled as graph rewriting rules for commands. There are four rules:

1. **Take:** Let $s$ be a subject such that $t \in (s, x)$, and $r \in (x, y)$ for some right $r$ and nodes $x$ and $y$. The command

   $s$ **take** $r$ for $y$ from $x$

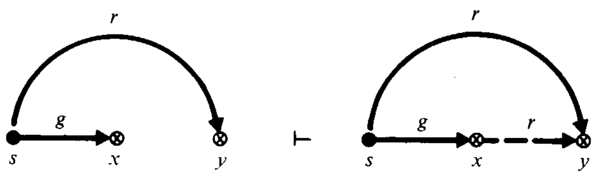
   adds $r$ to $(s, y)$. Graphically,



   where the symbol "⊗" denotes vertices that may be either subjects or objects.
2. **Grant:** Let $s$ be a subject such that $g \in (s, x)$ and $r \in (s, y)$ for some right $r$ and nodes $x$ and $y$. The command

   $s$ **grant** $r$ for $y$ to $x$

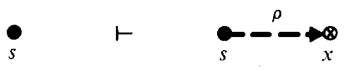
   adds $r$ to $(x, y)$. Graphically,



3. **Create:** Let $s$ be a subject and $\rho$ a set of rights. The command

   $s$ **create** $\rho$ for **new** $\left\{ \begin{array}{c} \textbf{subject} \\ \textbf{object} \end{array} \right\} x$

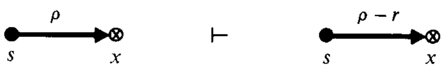
   adds a new node $x$ and sets $(s, x) = \rho$. Graphically,



4. **Remove:** Let $s$ be a subject and $x$ a node. The command

   s **remove** $r$ for $x$

   deletes $r$ from $(s, x)$. Graphically,



We have stated the commands **take**, **grant**, and **remove** as operations on a single right $r$. These commands can also be applied to subsets of rights, as is done in [Jone76b,Snyd81a].

***Example:***
The following commands show how process $P1$ can create a new file $F7$ and add it to the directory $D11$ shown in Figure 4.33.

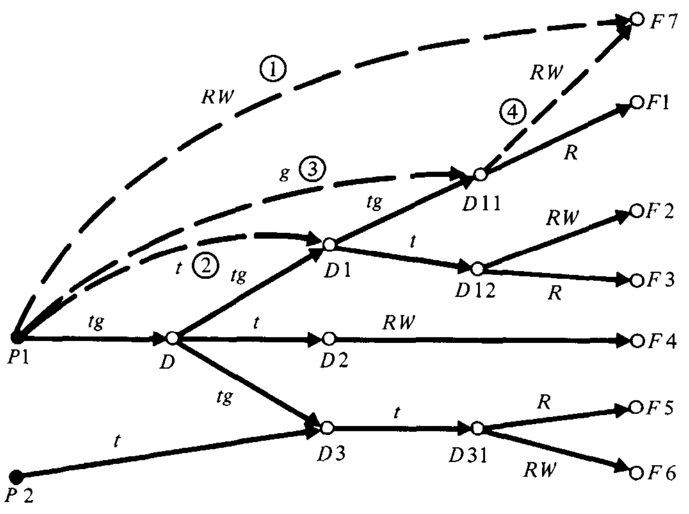
1. $P1$ **create** $RW$ for **new object** $F7$
2. $P1$ **take** $t$ for $D1$ from $D$
3. $P1$ **take** $g$ for $D11$ from $D1$
4. $P1$ **grant** $RW$ for $F7$ to $D11$

The effect of these commands is shown in Figure 4.35. ■

Let $G$ be a protection graph. We shall write $G \vdash_c G'$ if command $c$ transforms $G$ into graph $G'$, $G \vdash G'$ if there exists some command $c$ such that $G \vdash_c G'$, and $G \vdash^* G'$ if there exists a (possibility null) sequence of commands that transforms $G$ into $G'$.

We are now ready to consider safety in the context of protection graphs. Suppose there exists a node $s$ with right $r$ for node $x$; thus $r \in (s, x)$. Recall that the safety question is to determine whether another subject can acquire the right $r$ (not necessarily for $x$). Rather than considering whether an arbitrary subject can acquire the right $r$ for an arbitrary node, we shall consider whether a particular node $p$ (subject or object) can acquire the particular right $r$ for $x$. We are interested in knowing whether this question is decidable, and, if so, the computational

FIGURE 4.35 Adding a new file $F7$ to directory $D11$.



complexity of the decision procedure. Note that if this question is decidable with a decision procedure having linear time complexity $T = O(n)$ for an initial graph with $n$ nodes, then the more general question is decidable within time $T = O(n^3)$ [only nodes ($p$ and $x$) existing in the initial graph need be considered].

The safety question is formalized as follows. Given an initial graph $G_0$ with nodes $s$, $x$, and $p$ such that $r \in (s, x)$ and $r \notin (p, x)$, $G_0$ is **safe** for the right $r$ for $x$ if and only if $r \notin (p, x)$ in every graph $G$ derivable from $G_0$ (i.e., $G_0 \vdash^* G$). We shall consider the question in two contexts: first, where $s$ can "share" its right with other nodes (but not necessarily $p$), and second, where the right must be "stolen" from $s$.

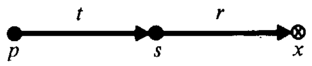
Given an initial graph $G_0$ with nodes $p$ and $x$ such that $r \notin (p, x)$ in $G_0$, the predicate *can.share*$(r, x, p, G_0)$ is true if and only if there exists a node $s$ in $G_0$ such that $r \in (s, x)$ and $G_0$ is unsafe for the right $r$ for $x$; that is, $p$ can acquire the right $r$ for $x$. To determine the conditions under which the predicate *can.share* is true, we first observe that rights can only be transferred along edges labeled with either $t$ or $g$. Two nodes $x$ and $y$ are ***tg*-connected** if there is a path between them such that each edge on the path is labeled with either $t$ or $g$ (the direction of the edge is not important); they are **directly *tg*-connected** if the path is the single edge $(x, y)$ or $(y, x)$. Jones, Lipton, and Snyder prove the following sufficient condition for *can.share:*

***Theorem 4.9:***
*can.share*$(r, x, p, G_0)$ is true if $p$ is a subject and

1. There exists a subject $s$ in $G_0$ such that $r \in (s, x)$ in $G_0$, and
2. $s$ and $p$ are directly $tg$-connected.

***Proof:***
There are four cases to consider:

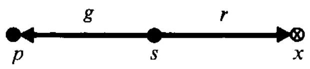
*Case 1.*



The first case is simple, as $p$ can simply take the right $r$ from $s$ with the command:

$p$ **take** $r$ for $x$ from $s$

*Case 2.*

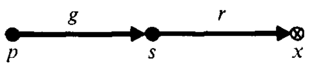


This case is also simple, as $s$ can grant (share) its right to $p$ with the command:

$s$ **grant** $r$ for $x$ to $p$

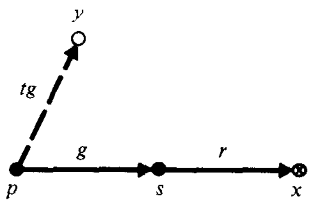
Previous Chapter Previous Page Home Next Page Next Chapter

*Case 3.*

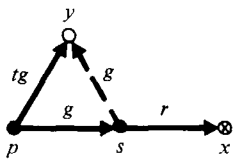


This case is less obvious, as $p$ cannot acquire the right with a single command. Nevertheless, with the cooperation of $s$, $p$ can acquire the right with four commands:

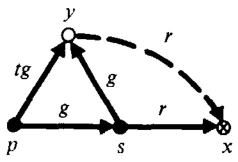
$p$ **create** $tg$ for **new object** $y$



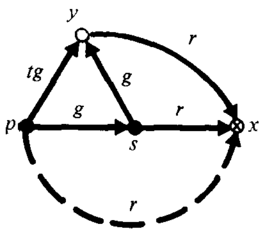
$p$ **grant** $g$ for $y$ to $s$



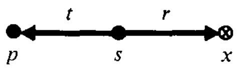
$s$ **grant** $r$ for $x$ to $y$



$p$ **take** $r$ for $x$ from $y$



*Case 4.*



This case also requires four commands; we leave it as an exercise for the reader. ■

Previous Chapter Previous Page Home Next Page Next Chapter

This result is easily extended to handle the case where subjects $s$ and $p$ are *tg*-connected by a path of length $\geq 1$ consisting of subjects only. Letting $p = p_0, p_1, \ldots, p_n = s$ denote the path between $p$ and $s$, each $p_i$ (for $i = n - 1, n - 2, \ldots, 0$) can acquire the right from $p_{i+1}$ as described in Theorem 4.9. It turns out that *tg*-connectivity is also a necessary condition for *can.share* in graphs containing only subjects; this is summarized in the following theorem:

***Theorem 4.10:***
If $G_0$ is a subject-only graph, then *can.share*$(r, x, p, G_0)$ is true if and only if:

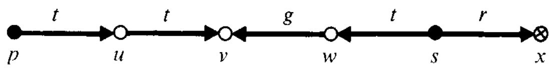
1. There exists a subject $s$ in $G_0$ such that $r \in (s, x)$, and
2. $s$ is *tg*-connected to $p$. ■

If the graph can contain both subjects and objects, the situation is more complicated. Before we can state results for this case, we must first introduce some new concepts.

An **island** is any maximal subject-only *tg*-connected subgraph. Clearly, once a right reaches an island, it can be shared with any of the subjects on the island. We must also describe how rights are transferred between two islands.

A ***tg*-path** is a path $s_1, o_2, \ldots, o_{n-1}, s_n$ of $n \geq 3$ *tg*-connected nodes, where $s_1$ and $s_n$ are subjects, and $o_2, \ldots, o_{n-1}$ are objects. A ***tg*-semipath** is a path $s_1, o_2, \ldots, o_n$ of $n \geq 2$ *tg*-connected nodes, where $s_1$ is a subject, and $o_2, \ldots, o_n$ are objects. Each *tg*-path or semipath may be described by a word over the alphabet $\{\overrightarrow{t}, \overrightarrow{g}, \overleftarrow{t}, \overleftarrow{g}\}$.

***Example:***
The *tg*-path connecting $p$ and $s$ in the following graph



is described by the word $\overrightarrow{t}\ \overrightarrow{t}\ \overleftarrow{g}\ \overleftarrow{t}$. ■

A **bridge** is a *tg*-path with an associated word in the regular expression:

$$(\overrightarrow{t})^* \cup (\overleftarrow{t})^* \cup (\overrightarrow{t})^*\ \overrightarrow{g}\ (\overleftarrow{t})^* \cup (\overrightarrow{t})^*\ \overleftarrow{g}\ (\overleftarrow{t})^* .$$

Bridges are used to transfer rights between two islands. The path $\overrightarrow{t}\ \overrightarrow{t}\ \overleftarrow{g}\ \overleftarrow{t}$ in the preceding example is a bridge; as an exercise, the reader should show how $s$ can share its right $r$ for $x$ with $p$.

An **initial span** is a *tg*-semipath with associated words in

$$(\overrightarrow{t})^*\ \overrightarrow{g}$$

and a **terminal span** is a *tg*-semipath with associated word in

$$(\overrightarrow{t})^* .$$

The arrows emanate from the subject $s_1$ in the semipaths. Note that a bridge is a composition of initial and terminal spans. The idea is that a subject on one island is responsible for transferring a right over the initial span of a bridge, and a subject on the other island is responsible for transferring the right over the terminal span; the middle of the bridge represents a node across which neither subject alone can transfer rights.

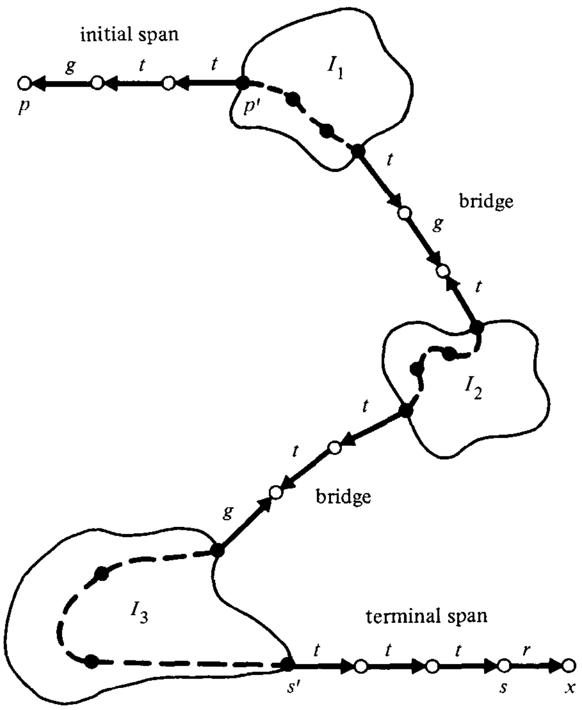
We now have the following theorem:

***Theorem 4.11:***
The predicate *can.share*$(r, x, p, G_0)$ is true if and only if:

1. There exists a node $s$ such that $r \in (s, x)$ in $G_0$; and
2. There exist subjects $p'$ and $s'$ such that
   a. $p' = p$ (if $p$ is a subject) or $p'$ is *tg*-connected to $p$ by an initial span (if $p$ is an object), and
   b. $s' = s$ (if $s$ is a subject) or $s'$ is *tg*-connected to $s$ by a terminal span (if $s$ is an object); and
   c. There exist islands $I_1, \ldots, I_u$ $(u \geq 1)$ such that $p' \in I_1$, $s' \in I_u$, and there is a bridge from $I_j$ to $I_{j+1}$ $(1 \leq j < u)$. ■

An initial span is used only when $p$ is an object; it allows the transfer of a right

FIGURE 4.36 Path over which $r$ for $x$ may be transferred from $s$ to $p$.



from an island to $p$ ($p$ is like the middle node of a bridge). A terminal span is similarly used only when $s$ is an object; it allows the transfer of the right $r$ from $s$ to an island ($s$ is like the middle node of a bridge). Figure 4.36 illustrates the path along which the right $r$ for $x$ is transferred from $s$ to $p$.

Jones, Lipton, and Snyder proved the following theorem:

***Theorem 4.12:***
There is an algorithm for testing *can.share* that operates in linear time in the size of the initial graph. ■

The algorithm performs a depth first search of the protection graph (e.g., see [Aho74]).

We now turn to the question of stealing. Intuitively, a node $p$ steals a right $r$ for $x$ from an owner $s$ if it acquires $r$ for $x$ without the explicit cooperation of $s$. Formally, the predicate *can.steal*$(r, x, p, G_0)$ is true if and only if $p$ does not have $r$ for $x$ in $G_0$ and there exist graphs $G_1, \dots, G_n$ such that:

1. $G_0 \vdash_{c_1} G_1 \vdash_{c_2} \dots \vdash_{c_n} G_n$;
2. $r \in (p, x)$ in $G_n$; and
3. For any subject $s$ such that $r \in (s, x)$ in $G_0$, no command $c_i$ is of the form

   $s$ **grant** $r$ for $x$ to $y$

   for any node $y$ in $G_{i-1}$.

Note, however, that condition (3) does not rule out an owner $s$ from transferring other rights. Snyder [Snyd81b] proved the following theorem, which states that a right must be stolen (taken) directly from its owner:

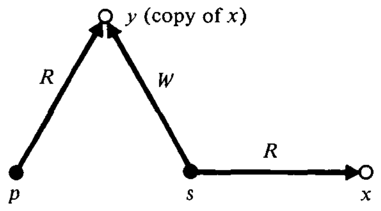
***Theorem 4.13:***
*can.steal*$(r, x, p, G_0)$ is true if and only if:

1. There is a subject $p'$ such that $p' = p$ (if $p$ is a subject) or $p'$ initially spans to $p$ (if $p$ is an object), and
2. There is a node $s$ such that $r \in (s, x)$ in $G_0$ and *can.share*$(t, s, p', G_0)$ is true; i.e., $p'$ can acquire the right to take from $s$. ■

This means if a subject cannot acquire the right to take from $s$, then it cannot steal a right from $s$ by some other means. Subjects that participate in the stealing are called **conspirators**.

If a subject $p$ cannot steal a right for an object $x$, this does not necessarily mean the information in $x$ is protected. For example, another subject $s$ may copy the information in $x$ into another object $y$ that $p$ can read (see Figure 4.37). To handle this problem, Bishop and Snyder [Bish79] distinguish between **de jure** acquisition, where $p$ obtains the read right $R$ for $x$, and **de facto** acquisition, where $p$ obtains the information in $x$, but not necessarily the right $R$ for $x$. They introduce four commands to describe information transfer using $R$, $W$ rights, and show that de facto acquisition can be described by graphs with certain kinds of $RW$-

FIGURE 4.37 De facto acquisition.



paths (analogous to the *tg*-paths). They also show de facto acquisition can be decided in time linear in the size of the initial graph. The problem of securing information flow is discussed in the next chapter.

The take-grant model is not intended to model any particular system or classes of systems, although it does describe many aspects of existing systems, especially capability systems. Nevertheless, the results are significant because they show that in properly constrained systems, safety decisions are not only possible but relatively simple. Safety is undecidable in the Harrison, Ruzzo, Ullman model because the commands of a system were unconstrained; a command could, if desired, grant some right *r* for *x* to every subject in the system. The take-grant model, on the other hand, constrains commands to pass rights only along *tg*-paths.

Snyder [Snyd77] investigated the problem of designing systems based on the take-grant model. He showed it is possible to design systems powerful enough to solve certain protection problems. One of his designs is outlined in the exercises at the end of this chapter.

Jones suggested an extension to the take-grant model for handling procedure calls and parameter passing [Jone78]. Her extension associates "property sets" with subjects and with passive procedure objects that serve as templates for subject creation. Execution of a procedure call causes a new subject to be created with rights defined by the templates.

## EXERCISES

4.1 Consider the revocation scheme used by System R (see Section 4.4.2), and suppose *Sysauth* contains the following tuples for a relation *Z* created by user *A*:

| User | Table | Grantor | Read | Insert | ... | Copy |
|---|---|---|---|---|---|---|
| *D* | *Z* | *A* | 5 | 5 | | yes |
| *B* | *Z* | *A* | 10 | 0 | | yes |
| *C* | *Z* | *B* | 15 | 0 | | yes |
| *C* | *Z* | *D* | 20 | 20 | | yes |
| *B* | *Z* | *C* | 30 | 30 | | yes |
| *E* | *Z* | *B* | 40 | 40 | | no |

Previous Chapter Previous Page Home Next Page Next Chapter

Draw a graph showing the transfer of rights. Suppose that at time $t = 50$, $A$ revokes all rights granted to $D$. Show the resulting state of *Sysauth*.

4.2 Write an algorithm for implementing the revocation procedure of System R.

4.3 Specify a policy for confinement (see Section 4.2.2), and design a capability-based mechanism for enforcing the policy.

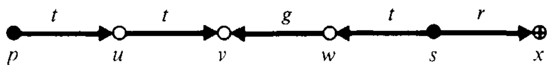
4.4 Complete the specifications of the module shown in Figure 4.31 by writing an *OV*-function for *pop*.

4.5 Consider the representation of a Turing machine as a protection system as described in Section 4.7.2. Complete the proof of Theorem 4.2 by showing how the move $\delta(q, X) = (p, Y, R)$ can be represented with two commands. Given the access matrix shown in Figure 4.32, show the matrix that results after the following two moves:

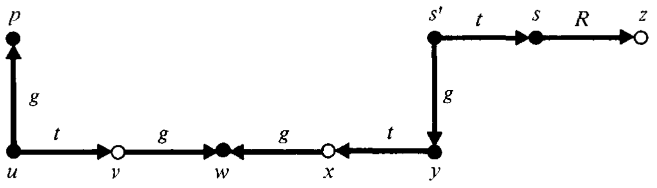
$$\delta(q, C) = (p, D, R)$$
$$\delta(p, D) = (s, E, R)\ .$$

4.6 Complete the proof of Theorem 4.9 by giving the command sequence for Case 4.

4.7 Give a sequence of commands showing how the right $r$ for $x$ can be transferred over the bridge $\overrightarrow{t}\ \overrightarrow{t}\ \overleftarrow{g}\ \overleftarrow{t}$ connecting $p$ and $s$ in the following graph:



4.8 Let $G_0$ be the protection graph:



a. Give a sequence of rule applications showing *can.share*$(R, z, p, G_0)$ is true.

b. Is *can.share*$(t, s', p, G_0)$ true? Why or why not?

c. Show *can.steal*$(R, z, p, G_0)$ is true, and list the conspirators.

4.9 Consider a system in which processes $p$ and $q$ communicate information stored in their private files through a shared message buffer $b$ provided by a trusted supervisor process $s$. Show that this system can be modeled as a take-grant system with subjects $s$, $p$, and $q$. Show an initial state in which process $p$ owns a file $x$, process $q$ owns a file $y$, and the supervisor $s$ has whatever rights it needs to establish the buffer (do not give the supervisor any more rights than it needs to do this). Construct a command sequence whereby the buffer is established, and show the graph produced by the command sequence.

## REFERENCES

Aho74. Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

Ande72. Anderson, J. P., "Computer Security Technology Planning Study," ESD-TR-73-51, Vols. I and II, USAF Electronic Systems Div., Bedford, Mass. (Oct. 1972).

Bara64. Baran, P., "On Distributed Communications: IX. Security, Secrecy, and Tamper-Free Considerations," RM-3765-PR, The Rand Corp., Santa Monica, Calif. (1964).

Bens72. Bensoussan, A., Clingen, C. T., and Daley, R. C., "The MULTICS Virtual Memory: Concepts and Design," *Comm. ACM* Vol. 15(5) pp. 308–318 (May 1972).

Bers79. Berson, T. A. and Barksdale, G. L., "KSOS—Development Methodology for a Secure Operating System," pp. 365–371 in *Proc. NCC*, Vol. 48, AFIPS Press, Montvale, N.J. (1979).

Bish79. Bishop, M. and Snyder, L., "The Transfer of Information and Authority in a Protection System," *Proc. 7th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.*, pp. 45–54 (Dec. 1979).

Boye79. Boyer, R. and Moore, J. S., *A Computational Logic*, Academic Press, New York (1979).

Boye80. Boyer, R. and Moore, J. S., "A Verification Condition Generator for Fortran," Computer Science Lab. Report CSL-103, SRI International, Menlo Park, Calif. (June 1980).

Broa76. Broadbridge, R. and Mekota, J., "Secure Communications Processor Specification," ESD-TR-76-351, AD-A055164, Honeywell Information Systems, McLean, Va. (June 1976).

Buck80. Buckingham, B. R. S., "CL/SWARD Command Language," SRI-CSL-79-013c, IBM Systems Research Institute, New York (Sept. 1980).

Carl75. Carlstedt, J., Bisbey, R. II, and Popek, G., "Pattern-Directed Protection Evaluation," NTIS AD-A012-474, Information Sciences Inst., Univ. of Southern Calif., Marina del Rey, Calif. (June 1975).

Cash76. Cash, J., Haseman, W. D., and Whinston, A. B., "Security for the GPLAN System," *Info. Systems* Vol. 2 pp. 41–48 (1976).

Cheh81. Cheheyl, M. H., Gasser, M., Huff, G. A., and Millen, J. K. "Verifying Security," *ACM Computing Surveys* Vol. 13(3) pp. 279–339 (Sept. 1981).

Codd70. Codd, E. F., "A Relational Model for Large Shared Data Banks," *Comm. ACM* Vol. 13(6) pp. 377–387 (1970).

Codd79. Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," *ACM Trans. on Database Syst.* Vol. 4(4) pp. 397–434 (Dec. 1979).

Cohe75. Cohen, E. and Jefferson, D., "Protection in the HYDRA Operating System," *Proc. 5th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 9(5) pp. 141–160 (Nov. 1975).

Conw72. Conway, R. W., Maxwell, W. L., and Morgan, H. L., "On the Implementation of Security Measures in Information Systems," *Comm. ACM* Vol. 15(4) pp. 211–220 (Apr. 1972).

Dahl72. Dahl, O. J. and Hoare, C. A. R., "Hierarchical Program Structures," in *Structured Programming*, ed. Dahl, Dijkstra, Hoare, Academic Press, New York (1972).

Dale65. Daley, R. C. and Neumann, P. G., "A General-Purpose File System for Secondary Storage," pp. 213–229 in *Proc. Fall Jt. Computer Conf.*, Vol. 27, AFIPS Press, Montvale, N.J. (1965).

Denn78. Denning, D. E., Denning, P. J., Garland, S. J., Harrison, M. A., and Ruzzo, W. L., "Proving Protection Systems Safe," Computer Sciences Dept., Purdue Univ., W. Lafayette, Ind. (Feb. 1978).

DenP71a. Denning, P. J., "An Undergraduate Course on Operating Systems Principles," Report of the Cosine Comm. of the Commission on Education, National Academy of Engineering, Washington, D.C. (June 1971).

DenP71b. Denning, P. J., "Third Generation Computer Systems," *Computing Surveys* Vol. 3(4) pp. 175–216 (Dec. 1971).

DeVH66. Dennis, J. B. and VanHorn, E. C., "Programming Semantics for Multiprogrammed Computations," *Comm. ACM* Vol. 9(3) pp. 143–155 (Mar. 1966).

Dens80. Dennis, T. D., "A Capability Architecture," Ph.D. Thesis, Computer Sciences Dept., Purdue Univ., W. Lafayette, Ind. (1980).

Dijk68. Dijkstra, E. W., "The Structure of the 'THE'—Multiprogramming System," *Comm. ACM* Vol. 11(5) pp. 341–346 (May 1968).

Down77. Downs, D. and Popek, G. J., "A Kernel Design for a Secure Data Base Management System," pp. 507–514 in *Proc. 3rd Conf. Very Large Data Bases,* IEEE and ACM, New York (1977).

Engl74. England, D. M., "Capability Concept Mechanism and Structure in System 250," pp. 63–82 in *Proc. Int. Workshop on Protection in Operating Systems,* Inst. Recherche d'Informatique, Rocquencourt, Le Chesnay, France (Aug. 1974).

Fabr71a. Fabry, R. S., "Preliminary Description of a Supervisor for a Machine Oriented Around Capabilities," ICR Quarterly Report 18, Univ. of Chicago, Chicago, Ill. (Mar. 1971).

Fabr71b. Fabry, R. S., "List Structured Addressing," Ph.D. Thesis, Univ. of Chicago, Chicago, Ill. (Mar. 1971).

Fabr74. Fabry, R. S. "Capability-Based Addressing," *Comm. ACM* Vol. 17(7) pp. 403–412 (July 1974).

Fagi78. Fagin, R., "On an Authorization Mechanism," *ACM Trans. on Database Syst.* Vol. 3(3) pp. 310–319 (Sept. 1978).

Feie79. Feiertag, R. J. and Neumann, P. G., "The Foundations of a Provably Secure Operating System (PSOS)," pp. 329–334 in *Proc. NCC,* Vol. 48, AFIPS Press, Montvale, N.J. (1979).

Floy67. Floyd, R. W., "Assigning Meaning to Programs," pp. 19–32 in *Math. Aspects of Computer Science,* ed. J. T. Schwartz, Amer. Math. Soc. (1967).

Gehr79. Gehringer, E., "Variable-Length Capabilities as a Solution to the Small-Object Problem," *Proc. 7th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.,* pp. 131–142 (Dec. 1979).

Giff82. Gifford, D. K., "Cryptographic Sealing for Information Security and Authentication," *Comm. ACM* (Apr. 1982).

Gold79. Gold, B. D., Linde, R. R., Peeler, R. J., Schaefer, M., Scheid, J. F., and Ward, P. D., "A Security Retrofit of VM/370," pp. 335–344 in *Proc. NCC,* Vol. 48, AFIPS Press, Montvale, N.J. (1979).

GrDe72. Graham, G. S. and Denning, P. J., "Protection—Principles and Practice," pp. 417–429 in *Proc. Spring Jt. Computer Conf.,* Vol. 40, AFIPS Press, Montvale, N. J. (1972).

Grah68. Graham, R. M., "Protection in an Information Processing Utility," *Comm. ACM* Vol. 11(5) pp. 365–369 (May 1968).

Grif76. Griffiths, P. P. and Wade, B. W., "An Authorization Mechanism for a Relational Database System," *ACM Trans. on Database Syst.* Vol. 1(3) pp. 242–255 (Sept. 1976).

Harr76. Harrison, M. A., Ruzzo, W. L., and Ullman, J. D., "Protection in Operating Systems," *Comm. ACM* Vol. 19(8) pp. 461–471 (Aug. 1976).

Harr78. Harrison, M. A. and Ruzzo, W. L., "Monotonic Protection Systems," pp. 337–365 in *Foundations of Secure Computation*, ed. R. A. DeMillo et al., Academic Press, New York (1978).

Hart76. Hartson, H. R. and Hsiao, D. K., "Full Protection Specifications in the Semantic Model for Database Protection Languages," *Proc. 1976 ACM Annual Conf.*, pp. 90–95 (Oct. 1976).

Hebb80. Hebbard, B. et al., "A Penetration Analysis of the Michigan Terminal System," *ACM Oper. Syst. Rev.* Vol. 14(1) pp. 7–20 (Jan. 1980).

Hoff71. Hoffman, L. J., "The Formulary Model for Flexible Privacy and Access Control," pp. 587–601 in *Proc. Fall Jt. Computer Conf.*, Vol. 39, AFIPS Press, Montvale, N.J. (1971).

IBM68. IBM, "IBM System/360 Principles of Operation," IBM Report No. GA22-6821 (Sept. 1968).

Ilif62. Iliffe, J. K. and Jodeit, J. G., "A Dynamic Storage Allocation System," *Computer J.* Vol. 5 pp. 200–209 (1962).

Ilif72. Iliffe, J. K., *Basic Machine Principles*, Elsevier/MacDonald, New York (1st ed. 1968, 2nd ed. 1972).

Jone76a. Jones, A. K. and Liskov, B. H., "A Language Extension Mechanism for Controlling Access to Shared Data," *Proc. 2nd Int. Conf. Software Eng.*, pp. 62–68 (1976).

Jone76b. Jones, A. K., Lipton, R. J., and Snyder, L., "A Linear Time Algorithm for Deciding Security," *Proc. 17th Annual Symp. on Found. of Comp. Sci.* (1976).

Jone78. Jones, A. K., "Protection Mechanism Models: Their Usefulness," pp. 237–254 in *Foundations of Secure Computation*, ed. R. A. DeMillo et al., Academic Press, New York (1978).

Jone79. Jones, A. K., Chansler, R. J., Durham, I., Schwans, K., and Vegdahl, S. R., "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proc. 7th Symp. on Oper. Syst. Princ., ACM Oper. Sys. Rev.*, pp. 117–121 (Dec. 1979).

Kahn81. Kahn, K. C., Corwin, W. M., Dennis, T. D., D'Hooge, H., Hubka, D. E., Hutchins, L. A., Montague, J. T., Pollack, F. J., Gifkins, M. R., "iMAX: A Multiprocessor Operating System for an Object-Based Computer," *Proc. 8th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.*, Vol. 15(5), pp. 127–136 (Dec. 1981).

Krei80. Kreissig, G., "A Model to Describe Protection Problems," pp. 9–17 in *Proc. 1980 Symp. on Security and Privacy*, IEEE Computer Society (Apr. 1980).

Lamp69. Lampson, B. W., "Dynamic Protection Structures," pp. 27–38 in *Proc. Fall Jt. Computer Conf.*, Vol. 35, AFIPS Press, Montvale, N.J. (1969).

Lamp71. Lampson, B. W., "Protection," *Proc. 5th Princeton Symp. of Info. Sci. and Syst.*, pp. 437–443 Princeton Univ., (Mar. 1971). Reprinted in *ACM Oper. Syst. Rev.*, Vol. 8(1) pp. 18–24 (Jan. 1974).

Lamp73. Lampson, B. W., "A Note on the Confinement Problem," *Comm. ACM* Vol. 16(10) pp. 613–615 (Oct. 1973).

Lamp76a. Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. J., "Report on the Programming Language Euclid" (Aug. 1976).

Lamp76b. Lampson, B. W. and Sturgis, H. E., "Reflections on an Operating System Design," *Comm. ACM* Vol. 19(5) pp. 251–265 (May 1976).

Levi81. Levitt, K. N. and Neumann, P. G., "Recent SRI Work in Verification," *ACM SIGSOFT Software Engineering Notes* Vol. 6(3) pp. 33–47. (July 1981).

Lind75. Linde, R. R., "Operating System Penetration," pp. 361–368 in *Proc. NCC*, Vol. 44, AFIPS Press, Montvale, N.J. (1975).

Linn76. Linden, T. A., "Operating System Structures to Support Security and Reliable Software," *Computing Surveys* Vol. 8(4) pp. 409–445 (Dec. 1976).

Lisk77. Liskov, B. H., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU," *Comm. ACM* Vol. 20(8) pp. 564–576 (Aug. 1977).

McCa79. McCauley, E. J. and Drongowski, P. J., "KSOS—The Design of a Secure Operating System," pp. 345–353 in *Proc. NCC,* Vol. 48, AFIPS Press, Montvale, N.J. (1979).

Mill76. Millen, J. K., "Security Kernel Validation in Practice," *Comm. ACM* Vol. 19(5) pp. 243–250 (May 1976).

Mins67. Minsky, M., *Computation: Finite and Infinite Machines,* Prentice-Hall, Englewood Cliffs, N.J. (1967).

MinN78. Minsky, N., "The Principle of Attenuation of Privileges and its Ramifications," pp. 255–276 in *Foundations of Secure Computation,* ed. R. A. DeMillo et al., Academic Press, New York (1978).

Morr78. Morris, J. B., "Programming by Successive Refinement," Dept. of Computer Sciences, Purdue Univ., W. Lafayette, Ind. (1978).

Mors73. Morris, J. H., "Protection in Programming Languages," *Comm. ACM* Vol. 16(1) pp. 15–21 (Jan. 1973).

Myer78. Myers, G., *Advances in Computer Architecture,* John Wiley & Sons, New York (1978).

Myer80. Myers, G. and Buckingham, B. R. S., "A Hardware Implementation of Capability-Based Addressing," *ACM Oper. Syst. Rev.* Vol. 14(4) pp. 13–25 (Oct. 1980).

Need77. Needham, R. M. and Walker, R. D. H., "The Cambridge CAP Computer and Its Protection System," *Proc. 6th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 11(5) pp. 1–10 (Nov. 1977).

Neum78. Neumann, P. G., "Computer Security Evaluation," pp. 1087–1095 in *Proc. NCC,* Vol. 47, AFIPS Press, Montvale, N.J. (1978).

Neum80. Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L., "A Provably Secure Operating System: The System, Its Applications, and Proofs," Computer Science Lab. Report CSL-116, SRI International, Menlo Park, Calif. (May 1980).

Orga72. Organick, E. I., *The Multics System: An Examination of Its Structure,* MIT Press, Cambridge, Mass. (1972).

Parn72. Parnas, D. L., "A Technique for Module Specification with Examples," *Comm. ACM* Vol. 15(5) pp. 330–336 (May 1972).

Pope74. Popek, G. J. and Kline, C. S., "Verifiable Secure Operating System Software," pp. 145–151 in *Proc. NCC,* Vol. 43, AFIPS Press, Montvale, N.J. (1974).

Pope78. Popek, G. J. and Farber, D. A., "A Model for Verification of Data Security in Operating Systems," *Comm. ACM* Vol. 21(9) pp. 737–749 (Sept. 1978).

Pope79. Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M., and Walton, E., "UCLA Secure Unix," pp. 355–364 in *Proc. NCC,* Vol. 48, AFIPS Press, Montvale, N.J. (1979).

Pric73. Price, W. R., "Implications of a Vertical Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. (1973).

Rede74. Redell, D. R. and Fabry, R. S., "Selective Revocation and Capabilities," pp. 197–209 in *Proc. Int. Workshop on Protection in Operating Systems,* Inst. de Recherche d'Informatique, Rocquencourt, Le Chesnay, France (Aug. 1974).

Ritc74. Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Comm. ACM* Vol. 17(7) pp. 365–375 (July 1974).

Robi77. Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," *Comm. ACM* Vol. 20(4) pp. 271–283 (Apr. 1977).

Robi79. Robinson, L.,"The HDM Handbook, Volume I: The Foundations of HDM," SRI Project 4828, SRI International, Menlo Park, Calif. (June 1979).

Roge67. Rogers, H., *Theory of Recursive Functions and Effective Computability,* McGraw-Hill, New York (1967). Section 11.2

Rush81. Rushby, J. M., "Design and Verification of Secure Systems," *Proc. 8th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.,* Vol. 15(5), pp. 12–21 (Dec. 1981).

Salt75. Saltzer, J. H. and Schroeder, M. D., "The Protection of Information in Computer Systems," *Proc. IEEE* Vol. 63(9) pp. 1278–1308 (Sept. 1975).

Schi75. Schiller, W. L., "The Design and Specification of a Security Kernel for the PDP 11/45," ESD-TR-75-69, The MITRE Corp., Bedford, Mass. (Mar. 1975).

Schr72. Schroeder, M. D. and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings," *Comm. ACM* Vol. 15(3) pp. 157–170 (Mar. 1972).

Schr77. Schroeder, M. D., Clark, D. D., and Saltzer, J. H., "The MULTICS Kernel Design Project," *Proc. 6th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 11(5) pp. 43–56 (Nov. 1977).

Sevc74. Sevcik, K. C. and Tsichritzis, D. C., "Authorization and Access Control Within Overall System Design," pp. 211–224 in *Proc. Int. Workshop on Protection in Operating Systems,* IRIA, Rocquencourt, Le Chesnay, France (1974).

Silv79. Silverberg, B., Robinson, L., and Levitt, K., "The HDM Handbook, Volume II: The Languages and Tools of HDM," SRI Project 4828, SRI International, Menlo Park, Calif. (June 1979).

Snyd77. Snyder, L., "On the Synthesis and Analysis of Protection Systems," *Proc. 6th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Rev.* Vol. 11(5) pp. 141–150 (Nov. 1977).

Snyd81a. Snyder, L., "Formal Models of Capability-Based Protection Systems," *IEEE Trans. on Computers* Vol. C-30(3) pp. 172–181 (Mar. 1981).

Snyd81b. Snyder, L., "Theft and Conspiracy in the Take-Grant Model," *JCSS* Vol. 23(3), pp. 333–347 (Dec. 1981).

Ston74. Stonebraker, M. and Wong, E., "Access Control in a Relational Data Base Management System by Query Modification," *Proc. 1974 ACM Annual Conf.,* pp. 180–186 (Nov. 1974).

Walk80. Walker, B. J., Kemmerer, R. A., and Popek, G. J., "Specification and Verification of the UCLA Unix Security Kernel," *Comm. ACM* Vol. 23(2) pp. 118–131 (Feb. 1980).

Walt75. Walter, K. G. et al., "Structured Specification of a Security Kernel," *Proc. Int. Conf. Reliable Software, ACM SIGPLAN Notices* Vol. 10(6) pp. 285–293 (June 1975).

Wulf74. Wulf, W. A., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., "HYDRA: The Kernel of a Multiprocessor System," *Comm. ACM* Vol. 17(6) pp. 337–345 (June 1974).

Wulf76. Wulf, W. A., London, R. L., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Trans. on Software Eng.* Vol. SE-2(4) pp. 253–265 (Dec. 1976).