Chapter 1
An Overview of Computer Security

ANTONIO: Whereof what's past is prologue, what to come
In yours and my discharge.
—*The Tempest*, II, i, 257–258.

This chapter presents the basic concepts of computer security. The remainder of the book will elaborate on these concepts in order to reveal the logic underlying the principles of these concepts.

We begin with basic security-related services that protect against threats to the security of the system. The next section discusses security policies that identify the threats and define the requirements for ensuring a secure system. Security mechanisms detect and prevent attacks and recover from those that succeed. Analyzing the security of a system requires an understanding of the mechanisms that enforce the security policy. It also requires a knowledge of the related assumptions and trust, which lead to the threats and the degree to which they may be realized. Such knowledge allows one to design better mechanisms and policies to neutralize these threats. This process leads to risk analysis. Human beings are the weakest link in the security mechanisms of any system. Therefore, policies and procedures must take people into account. This chapter discusses each of these topics.

1.1 The Basic Components

Computer security rests on confidentiality, integrity, and availability. The interpretations of these three aspects vary, as do the contexts in which they arise. The interpretation of an aspect in a given environment is dictated by the needs of the individuals, customs, and laws of the particular organization.

1.1.1 Confidentiality

Confidentiality is the concealment of information or resources. The need for keeping information secret arises from the use of computers in sensitive fields such as government and industry. For example, military and civilian institutions in the government often restrict access to information to those who need that information. The first formal work in computer security was motivated by the military's attempt to implement controls to enforce a "need to know" principle. This principle also applies to industrial firms, which keep their proprietary designs secure lest their competitors try to steal the designs. As a further example, all types of institutions keep personnel records secret.

Access control mechanisms support confidentiality. One access control mechanism for preserving confidentiality is cryptography, which scrambles data to make it incomprehensible. A *cryptographic key* controls access to the unscrambled data, but then the cryptographic key itself becomes another datum to be protected.

EXAMPLE: Enciphering an income tax return will prevent anyone from reading it. If the owner needs to see the return, it must be deciphered. Only the possessor of the cryptographic key can enter it into a deciphering program. However, if someone else can read the key when it is entered into the program, the confidentiality of the tax return has been compromised.

Other system-dependent mechanisms can prevent processes from illicitly accessing information. Unlike enciphered data, however, data protected only by these controls can be read when the controls fail or are bypassed. Then their advantage is offset by a corresponding disadvantage. They can protect the secrecy of data more completely than cryptography, but if they fail or are evaded, the data becomes visible.

Confidentiality also applies to the existence of data, which is sometimes more revealing than the data itself. The precise number of people who distrust a politician may be less important than knowing that such a poll was taken by the politician's staff. How a particular government agency harassed citizens in its country may be less important than knowing that such harassment occurred. Access control mechanisms sometimes conceal the mere existence of data, lest the existence itself reveal information that should be protected.

Resource hiding is another important aspect of confidentiality. Sites often wish to conceal their configuration as well as what systems they are using; organizations may not wish others to know about specific equipment (because it could be used without authorization or in inappropriate ways), and a company renting time from a service provider may not want others to know what resources it is using. Access control mechanisms provide these capabilities as well.

All the mechanisms that enforce confidentiality require supporting services from the system. The assumption is that the security services can rely on the kernel, and other agents, to supply correct data. Thus, assumptions and trust underlie confidentiality mechanisms.

1.1.2 Integrity

Integrity refers to the trustworthiness of data or resources, and it is usually phrased in terms of preventing improper or unauthorized change. Integrity includes data integrity (the content of the information) and origin integrity (the source of the data, often called *authentication*). The source of the information may bear on its accuracy and credibility and on the trust that people place in the information.This dichotomy illustrates the principle that the aspect of integrity known as credibility is central to the proper functioning of a system. We will return to this issue when discussing malicious logic.

EXAMPLE: A newspaper may print information obtained from a leak at the White House but attribute it to the wrong source. The information is printed as received (preserving data integrity), but its source is incorrect (corrupting origin integrity).

Integrity mechanisms fall into two classes: *prevention* mechanisms and *detection* mechanisms.

Prevention mechanisms seek to maintain the integrity of the data by blocking any unauthorized attempts to change the data or any attempts to change the data in unauthorized ways. The distinction between these two types of attempts is important. The former occurs when a user tries to change data which she has no authority to change. The latter occurs when a user authorized to make certain changes in the data tries to change the data in other ways. For example, suppose an accounting system is on a computer. Someone breaks into the system and tries to modify the accounting data. Then an unauthorized user has tried to violate the integrity of the accounting database. But if an accountant hired by the firm to maintain its books tries to embezzle money by sending it overseas and hiding the transactions, a user (the accountant) has tried to change data (the accounting data) in unauthorized ways (by moving it to a Swiss bank account). Adequate authentication and access controls will generally stop the break-in from the outside, but preventing the second type of attempt requires very different controls.

Detection mechanisms do not try to prevent violations of integrity; they simply report that the data's integrity is no longer trustworthy. Detection mechanisms may analyze system events (user or system actions) to detect problems or (more commonly) may analyze the data itself to see if required or expected constraints still hold. The mechanisms may report the actual cause of the integrity violation (a specific part of a file was altered), or they may simply report that the file is now corrupt.

Working with integrity is very different from working with confidentiality. With confidentiality, the data is either compromised or it is not, but integrity includes both the correctness and the trustworthiness of the data. The origin of the data (how and from whom it was obtained), how well the data was protected before it arrived at the current machine, and how well the data is protected on the current machine all affect the integrity of the data. Thus, evaluating integrity is often very difficult, because it relies on assumptions about the source of the data and about trust in that source—two underpinnings of security that are often overlooked.

1.1.3 Availability

Availability refers to the ability to use the information or resource desired. Availability is an important aspect of reliability as well as of system design because an unavailable system is at least as bad as no system at all. The aspect of availability that is relevant to security is that someone may deliberately arrange to deny access to data or to a service by making it unavailable. System designs usually assume a statistical model to analyze expected patterns of use, and mechanisms ensure availability when that statistical model holds. Someone may be able to manipulate use (or parameters that control use, such as network traffic) so that the assumptions of the statistical model are no longer valid. This means that the mechanisms for keeping the resource or data available are working in an environment for which they were not designed. As a result, they will often fail.

EXAMPLE: Suppose Anne has compromised a bank's secondary system server, which supplies bank account balances. When anyone else asks that server for information, Anne can supply any information she desires. Merchants validate checks by contacting the bank's primary balance server. If a merchant gets no response, the secondary server will be asked to supply the data. Anne's colleague prevents merchants from contacting the primary balance server, so all merchant queries go to the secondary server. Anne will never have a check turned down, regardless of her actual account balance. Notice that if the bank had only one server (the primary one), this scheme would not work. The merchant would be unable to validate the check.

Attempts to block availability, called *denial of service attacks*, can be the most difficult to detect, because the analyst must determine if the unusual access patterns are attributable to deliberate manipulation of resources or of environment. Complicating this determination is the nature of statistical models. Even if the model accurately describes the environment, atypical events simply contribute to the nature of the statistics. A deliberate attempt to make a resource unavailable may simply look like, or be, an atypical event. In some environments, it may not even appear atypical.

1.2 Threats

A *threat* is a potential violation of security. The violation need not actually occur for there to be a threat. The fact that the violation *might* occur means that those actions that could cause it to occur must be guarded against (or prepared for). Those actions are called *attacks*. Those who execute such actions, or cause them to be executed, are called *attackers*.

The three security services—confidentiality, integrity, and availability—counter threats to the security of a system. Shirey [823] divides threats into four broad classes: *disclosure*, or unauthorized access to information; *deception*, or

acceptance of false data; *disruption*, or interruption or prevention of correct operation; and *usurpation*, or unauthorized control of some part of a system. These four broad classes encompass many common threats. Because the threats are ubiquitous, an introductory discussion of each one will present issues that recur throughout the study of computer security.

Snooping, the unauthorized interception of information, is a form of disclosure. It is passive, suggesting simply that some entity is listening to (or reading) communications or browsing through files or system information. *Wiretapping*, or *passive wiretapping*, is a form of snooping in which a network is monitored. (It is called "wiretapping" because of the "wires" that compose the network, although the term is used even if no physical wiring is involved.) Confidentiality services counter this threat.

Modification or *alteration*, an unauthorized change of information, covers three classes of threats. The goal may be deception, in which some entity relies on the modified data to determine which action to take, or in which incorrect information is accepted as correct and is released. If the modified data controls the operation of the system, the threats of disruption and usurpation arise. Unlike snooping, modification is active; it results from an entity changing information. *Active wiretapping* is a form of modification in which data moving across a network is altered; the term "active" distinguishes it from snooping ("passive" wiretapping). An example is the *man-in-the-middle* attack, in which an intruder reads messages from the sender and sends (possibly modified) versions to the recipient, in hopes that the recipient and sender will not realize the presence of the intermediary. Integrity services counter this threat.

Masquerading or *spoofing*, an impersonation of one entity by another, is a form of both deception and usurpation. It lures a victim into believing that the entity with which it is communicating is a different entity. For example, if a user tries to log into a computer across the Internet but instead reaches another computer that claims to be the desired one, the user has been spoofed. Similarly, if a user tries to read a file, but an attacker has arranged for the user to be given a different file, another spoof has taken place. This may be a passive attack (in which the user does not attempt to authenticate the recipient, but merely accesses it), but it is usually an active attack (in which the masquerader issues responses to mislead the user about its identity). Although primarily deception, it is often used to usurp control of a system by an attacker impersonating an authorized manager or controller. Integrity services (called "authentication services" in this context) counter this threat.

Some forms of masquerading may be allowed. *Delegation* occurs when one entity authorizes a second entity to perform functions on its behalf. The distinctions between delegation and masquerading are important. If Susan delegates to Thomas the authority to act on her behalf, she is giving permission for him to perform specific actions as though she were performing them herself. All parties are aware of the delegation. Thomas will not pretend to be Susan; rather, he will say, "I am Thomas and I have authority to do this on Susan's behalf." If asked, Susan will verify this. On the other hand, in a masquerade, Thomas will pretend to be Susan. No other parties (including Susan) will be aware of the masquerade, and Thomas will say, "I am Susan." Should anyone discover that he or she is dealing with Thomas and ask Susan

about it, she will deny that she authorized Thomas to act on her behalf. In terms of security, masquerading is a violation of security, whereas delegation is not.

Repudiation of origin, a false denial that an entity sent (or created) something, is a form of deception. For example, suppose a customer sends a letter to a vendor agreeing to pay a large amount of money for a product. The vendor ships the product and then demands payment. The customer denies having ordered the product and by law is therefore entitled to keep the unsolicited shipment without payment. The customer has repudiated the origin of the letter. If the vendor cannot prove that the letter came from the customer, the attack succeeds. A variant of this is denial by a user that he created specific information or entities such as files. Integrity mechanisms cope with this threat.

Denial of receipt, a false denial that an entity received some information or message, is a form of deception. Suppose a customer orders an expensive product, but the vendor demands payment before shipment. The customer pays, and the vendor ships the product. The customer then asks the vendor when he will receive the product. If the customer has already received the product, the question constitutes a denial of receipt attack. The vendor can defend against this attack only by proving that the customer did, despite his denials, receive the product. Integrity and availability mechanisms guard against these attacks.

Delay, a temporary inhibition of a service, is a form of usurpation, although it can play a supporting role in deception. Typically, delivery of a message or service requires some time t; if an attacker can force the delivery to take more than time t, the attacker has successfully delayed delivery. This requires manipulation of system control structures, such as network components or server components, and hence is a form of usurpation. If an entity is waiting for an authorization message that is delayed, it may query a secondary server for the authorization. Even though the attacker may be unable to masquerade as the primary server, she might be able to masquerade as that secondary server and supply incorrect information. Availability mechanisms can thwart this threat.

Denial of service, a long-term inhibition of service, is a form of usurpation, although it is often used with other mechanisms to deceive. The attacker prevents a server from providing a service. The denial may occur at the source (by preventing the server from obtaining the resources needed to perform its function), at the destination (by blocking the communications from the server), or along the intermediate path (by discarding messages from either the client or the server, or both). Denial of service poses the same threat as an infinite delay. Availability mechanisms counter this threat.

Denial of service or delay may result from direct attacks or from nonsecurity-related problems. From our point of view, the cause and result are important; the intention underlying them is not. If delay or denial of service compromises system security, or is part of a sequence of events leading to the compromise of a system, then we view it as an attempt to breach system security. But the attempt may not be deliberate; indeed, it may be the product of environmental characteristics rather than specific actions of an attacker.

1.3 Policy and Mechanism

Critical to our study of security is the distinction between policy and mechanism.

Definition 1–1. A *security policy* is a statement of what is, and what is not, allowed.

Definition 1–2. A *security mechanism* is a method, tool, or procedure for enforcing a security policy.

Mechanisms can be nontechnical, such as requiring proof of identity before changing a password; in fact, policies often require some procedural mechanisms that technology cannot enforce.

As an example, suppose a university's computer science laboratory has a policy that prohibits any student from copying another student's homework files. The computer system provides mechanisms for preventing others from reading a user's files. Anna fails to use these mechanisms to protect her homework files, and Bill copies them. A breach of security has occurred, because Bill has violated the security policy. Anna's failure to protect her files does not authorize Bill to copy them.

In this example, Anna could easily have protected her files. In other environments, such protection may not be easy. For example, the Internet provides only the most rudimentary security mechanisms, which are not adequate to protect information sent over that network. Nevertheless, acts such as the recording of passwords and other sensitive information violate an implicit security policy of most sites (specifically, that passwords are a user's confidential property and cannot be recorded by anyone).

Policies may be presented mathematically, as a list of allowed (secure) and disallowed (nonsecure) states. For our purposes, we will assume that any given policy provides an axiomatic description of secure states and nonsecure states. In practice, policies are rarely so precise; they normally describe in English what users and staff are allowed to do. The ambiguity inherent in such a description leads to states that are not classified as "allowed" or "disallowed." For example, consider the homework policy discussed above. If someone looks through another user's directory without copying homework files, is that a violation of security? The answer depends on site custom, rules, regulations, and laws, all of which are outside our focus and may change over time.

When two different sites communicate or cooperate, the entity they compose has a security policy based on the security policies of the two entities. If those policies are inconsistent, either or both sites must decide what the security policy for the combined site should be. The inconsistency often manifests itself as a security breach. For example, if proprietary documents were given to a university, the policy of confidentiality in the corporation would conflict with the more open policies of most universities. The university and the company must develop a mutual security policy that meets both their needs in order to produce a consistent policy. When the

two sites communicate through an independent third party, such as an Internet service provider, the complexity of the situation grows rapidly.

1.3.1 Goals of Security

Given a security policy's specification of "secure" and "nonsecure" actions, these security mechanisms can prevent the attack, detect the attack, or recover from the attack. The strategies may be used together or separately.

Prevention means that an attack will fail. For example, if one attempts to break into a host over the Internet and that host is not connected to the Internet, the attack has been prevented. Typically, prevention involves implementation of mechanisms that users cannot override and that are trusted to be implemented in a correct, unalterable way, so that the attacker cannot defeat the mechanism by changing it. Preventative mechanisms often are very cumbersome and interfere with system use to the point that they hinder normal use of the system. But some simple preventative mechanisms, such as passwords (which aim to prevent unauthorized users from accessing the system), have become widely accepted. Prevention mechanisms can prevent compromise of parts of the system; once in place, the resource protected by the mechanism need not be monitored for security problems, at least in theory.

Detection is most useful when an attack cannot be prevented, but it can also indicate the effectiveness of preventative measures. Detection mechanisms accept that an attack will occur; the goal is to determine that an attack is under way, or has occurred, and report it. The attack may be monitored, however, to provide data about its nature, severity, and results. Typical detection mechanisms monitor various aspects of the system, looking for actions or information indicating an attack. A good example of such a mechanism is one that gives a warning when a user enters an incorrect password three times. The login may continue, but an error message in a system log reports the unusually high number of mistyped passwords. Detection mechanisms do not prevent compromise of parts of the system, which is a serious drawback. The resource protected by the detection mechanism is continuously or periodically monitored for security problems.

Recovery has two forms. The first is to stop an attack and to assess and repair any damage caused by that attack. As an example, if the attacker deletes a file, one recovery mechanism would be to restore the file from backup tapes. In practice, recovery is far more complex, because the nature of each attack is unique. Thus, the type and extent of any damage can be difficult to characterize completely. Moreover, the attacker may return, so recovery involves identification and fixing of the vulnerabilities used by the attacker to enter the system. In some cases, retaliation (by attacking the attacker's system or taking legal steps to hold the attacker accountable) is part of recovery. In all these cases, the system's functioning is inhibited by the attack. By definition, recovery requires resumption of correct operation.

In a second form of recovery, the system continues to function correctly while an attack is under way. This type of recovery is quite difficult to implement because of the complexity of computer systems. It draws on techniques of fault tolerance as

well as techniques of security and is typically used in safety-critical systems. It differs from the first form of recovery, because at no point does the system function incorrectly. However, the system may disable nonessential functionality. Of course, this type of recovery is often implemented in a weaker form whereby the system detects incorrect functioning automatically and then corrects (or attempts to correct) the error.

1.4 Assumptions and Trust

How do we determine if the policy correctly describes the required level and type of security for the site? This question lies at the heart of all security, computer and otherwise. Security rests on assumptions specific to the type of security required and the environment in which it is to be employed.

EXAMPLE: Opening a door lock requires a key. The assumption is that the lock is secure against lock picking. This assumption is treated as an axiom and is made because most people would require a key to open a door lock. A good lock picker, however, can open a lock without a key. Hence, in an environment with a skilled, untrustworthy lock picker, the assumption is wrong and the consequence invalid.

If the lock picker is trustworthy, the assumption is valid. The term "trustworthy" implies that the lock picker will not pick a lock unless the owner of the lock authorizes the lock picking. This is another example of the role of trust. A well-defined exception to the rules provides a "back door" through which the security mechanism (the locks) can be bypassed. The trust resides in the belief that this back door will not be used except as specified by the policy. If it is used, the trust has been misplaced and the security mechanism (the lock) provides no security.

Like the lock example, a policy consists of a set of axioms that the policy makers believe can be enforced. Designers of policies always make two assumptions. First, the policy correctly and unambiguously partitions the set of system states into "secure" and "nonsecure" states. Second, the security mechanisms prevent the system from entering a "nonsecure" state. If either assumption is erroneous, the system will be nonsecure.

These two assumptions are fundamentally different. The first assumption asserts that the policy is a correct description of what constitutes a "secure" system. For example, a bank's policy may state that officers of the bank are authorized to shift money among accounts. If a bank officer puts $100,000 in his account, has the bank's security been violated? Given the aforementioned policy statement, no, because the officer was authorized to move the money. In the "real world," that action would constitute embezzlement, something any bank would consider a security violation.

The second assumption says that the security policy can be enforced by security mechanisms. These mechanisms are either *secure*, *precise*, or *broad.* Let P be the

set of all possible states. Let Q be the set of secure states (as specified by the security policy). Let the security mechanisms restrict the system to some set of states R (thus, $R \subseteq P$). Then we have the following definition.

> **Definition 1–3.** A security mechanism is *secure* if $R \subseteq Q$; it is *precise* if $R = Q$; and it is *broad* if there are states r such that $r \in R$ and $r \notin Q$.

Ideally, the union of all security mechanisms active on a system would produce a single precise mechanism (that is, $R = Q$). In practice, security mechanisms are broad; they allow the system to enter nonsecure states. We will revisit this topic when we explore policy formulation in more detail.

Trusting that mechanisms work requires several assumptions.

1. Each mechanism is designed to implement one or more parts of the security policy.
2. The union of the mechanisms implements all aspects of the security policy.
3. The mechanisms are implemented correctly.
4. The mechanisms are installed and administered correctly.

Because of the importance and complexity of trust and of assumptions, we will revisit this topic repeatedly and in various guises throughout this book.

1.5 Assurance

Trust cannot be quantified precisely. System specification, design, and implementation can provide a basis for determining "how much" to trust a system. This aspect of trust is called *assurance*. It is an attempt to provide a basis for bolstering (or substantiating or specifying) how much one can trust a system.

EXAMPLE: In the United States, aspirin from a nationally known and reputable manufacturer, delivered to the drugstore in a safety-sealed container, and sold with the seal still in place, is considered trustworthy by most people. The bases for that trust are as follows.

- The testing and certification of the drug (aspirin) by the Food and Drug Administration. The FDA has jurisdiction over many types of medicines and allows medicines to be marketed only if they meet certain clinical standards of usefulness.

- The manufacturing standards of the company and the precautions it takes to ensure that the drug is not contaminated. National and state regulatory commissions and groups ensure that the manufacture of the drug meets specific acceptable standards.
- The safety seal on the bottle. To insert dangerous chemicals into a safety-sealed bottle without damaging the seal is very difficult.

The three technologies (certification, manufacturing standards, and preventative sealing) provide some degree of assurance that the aspirin is not contaminated. The degree of trust the purchaser has in the purity of the aspirin is a result of these three processes.

In the 1980s, drug manufacturers met two of the criteria above, but none used safety seals.[1] A series of "drug scares" arose when a well-known manufacturer's medicines were contaminated after manufacture but before purchase. The manufacturer promptly introduced safety seals to assure its customers that the medicine in the container was the same as when it was shipped from the manufacturing plants.

Assurance in the computer world is similar. It requires specific steps to ensure that the computer will function properly. The sequence of steps includes detailed specifications of the desired (or undesirable) behavior; an analysis of the design of the hardware, software, and other components to show that the system will not violate the specifications; and arguments or proofs that the implementation, operating procedures, and maintenance procedures will produce the desired behavior.

Definition 1–4. A system is said to *satisfy* a specification if the specification correctly states how the system will function.

This definition also applies to design and implementation satisfying a specification.

1.5.1 Specification

A *specification* is a (formal or informal) statement of the desired functioning of the system. It can be highly mathematical, using any of several languages defined for that purpose. It can also be informal, using, for example, English to describe what the system should do under certain conditions. The specification can be low-level, combining program code with logical and temporal relationships to specify ordering of events. The defining quality is a statement of what the system is allowed to do or what it is not allowed to do.

[1] Many used childproof caps, but they prevented only young children (and some adults) from opening the bottles. They were not designed to protect the medicine from malicious adults.

EXAMPLE: A company is purchasing a new computer for internal use. They need to trust the system to be invulnerable to attack over the Internet. One of their (English) specifications would read "The system cannot be attacked over the Internet."

Specifications are used not merely in security but also in systems designed for safety, such as medical technology. They constrain such systems from performing acts that could cause harm. A system that regulates traffic lights must ensure that pairs of lights facing the same way turn red, green, and yellow at the same time and that at most one set of lights facing cross streets at an intersection is green.

A major part of the derivation of specifications is determination of the set of requirements relevant to the system's planned use. Section 1.6 discusses the relationship of requirements to security.

1.5.2 Design

The *design* of a system translates the specifications into components that will implement them. The design is said to *satisfy* the specifications if, under all relevant circumstances, the design will not permit the system to violate those specifications.

EXAMPLE: A design of the computer system for the company mentioned above had no network interface cards, no modem cards, and no network drivers in the kernel. This design satisfied the specification because the system would not connect to the Internet. Hence it could not be attacked over the Internet.

An analyst can determine whether a design satisfies a set of specifications in several ways. If the specifications and designs are expressed in terms of mathematics, the analyst must show that the design formulations are consistent with the specifications. Although much of the work can be done mechanically, a human must still perform some analyses and modify components of the design that violate specifications (or, in some cases, components that cannot be shown to satisfy the specifications). If the specifications and design do not use mathematics, then a convincing and compelling argument should be made. Most often, the specifications are nebulous and the arguments are half-hearted and unconvincing or provide only partial coverage. The design depends on assumptions about what the specifications mean. This leads to vulnerabilities, as we will see.

1.5.3 Implementation

Given a design, the *implementation* creates a system that satisfies that design. If the design also satisfies the specifications, then by transitivity the implementation will also satisfy the specifications.

The difficulty at this step is the complexity of proving that a program correctly implements the design and, in turn, the specifications.

Definition 1–5. A program is *correct* if its implementation performs as specified.

Proofs of correctness require each line of source code to be checked for mathematical correctness. Each line is seen as a function, transforming the input (constrained by preconditions) into some output (constrained by postconditions derived from the function and the preconditions). Each routine is represented by the composition of the functions derived from the lines of code making up the routine. Like those functions, the function corresponding to the routine has inputs and outputs, constrained by preconditions and postconditions, respectively. From the combination of routines, programs can be built and formally verified. One can apply the same techniques to sets of programs and thus verify the correctness of a system.

There are three difficulties in this process. First, the complexity of programs makes their mathematical verification difficult. Aside from the intrinsic difficulties, the program itself has preconditions derived from the environment of the system. These preconditions are often subtle and difficult to specify, but unless the mathematical formalism captures them, the program verification may not be valid because critical assumptions may be wrong. Second, program verification assumes that the programs are compiled correctly, linked and loaded correctly, and executed correctly. Hardware failure, buggy code, and failures in other tools may invalidate the preconditions. A compiler that incorrectly compiles

```
x := x + 1
```

to

```
move x to regA
subtract 1 from contents of regA
move contents of regA to x
```

would invalidate the proof statement that the value of x after the line of code is 1 more than the value of x before the line of code. This would invalidate the proof of correctness. Third, if the verification relies on conditions on the input, the program must reject any inputs that do not meet those conditions. Otherwise, the program is only partially verified.

Because formal proofs of correctness are so time-consuming, *a posteriori* verification techniques known as *testing* have become widespread. During testing, the tester executes the program (or portions of it) on data to determine if the output is what it should be and to understand how likely the program is to contain an error. Testing techniques range from supplying input to ensure that all execution paths are exercised to introducing errors into the program and determining how they affect the output to stating specifications and testing the program to see if it satisfies the specifications. Although these techniques are considerably simpler than the more formal methods, they do not provide the same degree of assurance that formal methods do.

Furthermore, testing relies on test procedures and documentation, errors in either of which could invalidate the testing results.

Although assurance techniques do not guarantee correctness or security, they provide a firm basis for assessing what one must trust in order to believe that a system is secure. Their value is in eliminating possible, and common, sources of error and forcing designers to define precisely what the system is to do.

1.6 Operational Issues

Any useful policy and mechanism must balance the benefits of the protection against the cost of designing, implementing, and using the mechanism. This balance can be determined by analyzing the risks of a security breach and the likelihood of it occurring. Such an analysis is, to a degree, subjective, because in very few situations can risks be rigorously quantified. Complicating the analysis are the constraints that laws, customs, and society in general place on the acceptability of security procedures and mechanisms; indeed, as these factors change, so do security mechanisms and, possibly, security policies.

1.6.1 Cost-Benefit Analysis

Like any factor in a complex system, the benefits of computer security are weighed against their total cost (including the additional costs incurred if the system is compromised). If the data or resources cost less, or are of less value, than their protection, adding security mechanisms and procedures is not cost-effective because the data or resources can be reconstructed more cheaply than the protections themselves. Unfortunately, this is rarely the case.

EXAMPLE: A database provides salary information to a second system that prints checks. If the data in the database is altered, the company could suffer grievous financial loss; hence, even a cursory cost-benefit analysis would show that the strongest possible integrity mechanisms should protect the data in the database.

Now suppose the company has several branch offices, and every day the database downloads a copy of the data to each branch office. The branch offices use the data to recommend salaries for new employees. However, the main office makes the final decision using the original database (not one of the copies). In this case, guarding the integrity of the copies is not particularly important, because branch offices cannot make any financial decisions based on the data in their copies. Hence, the company cannot suffer any financial loss.

Both of these situations are extreme situations in which the analysis is clear-cut. As an example of a situation in which the analysis is less clear, consider the need

for confidentiality of the salaries in the database. The officers of the company must decide the financial cost to the company should the salaries be disclosed, including potential loss from lawsuits (if any); changes in policies, procedures, and personnel; and the effect on future business. These are all business-related judgments, and determining their value is part of what company officers are paid to do.

Overlapping benefits are also a consideration. Suppose the integrity protection mechanism can be augmented very quickly and cheaply to provide confidentiality. Then the cost of providing confidentiality is much lower. This shows that evaluating the cost of a particular security service depends on the mechanism chosen to implement it and on the mechanisms chosen to implement other security services. The cost-benefit analysis should take into account as many mechanisms as possible. Adding security mechanisms to an existing system is often more expensive (and, incidentally, less effective) than designing them into the system in the first place.

1.6.2 Risk Analysis

To determine whether an asset should be protected, and to what level, requires analysis of the potential threats against that asset and the likelihood that they will materialize. The level of protection is a function of the probability of an attack occurring and the effects of the attack should it succeed. If an attack is unlikely, protecting against it has a lower priority than protecting against a likely one. If the unlikely attack would cause long delays in the company's production of widgets but the likely attack would be only a nuisance, then more effort should be put into preventing the unlikely attack. The situations between these extreme cases are far more subjective.

Let's revisit our company with the salary database that transmits salary information over a network to a second computer that prints employees' checks. The data is stored on the database system and then moved over the network to the second system. Hence, the risk of unauthorized changes in the data occurs in three places: on the database system, on the network, and on the printing system. If the network is a local (company-wide) one and no wide area networks are accessible, the threat of attackers entering the systems is confined to untrustworthy internal personnel. If, however, the network is connected to the Internet, the risk of geographically distant attackers attempting to intrude is substantial enough to warrant consideration.

This example illustrates some finer points of risk analysis. First, risk is a function of environment. Attackers from a foreign country are not a threat to the company when the computer is not connected to the Internet. If foreign attackers wanted to break into the system, they would need physically to enter the company (and would cease to be "foreign" because they would then be "local"). But if the computer is connected to the Internet, foreign attackers become a threat because they can attack over the Internet. An additional, less tangible issue is the faith in the company. If the company is not able to meet its payroll because it does not know *whom* it is to pay, the company will lose the faith of its employees. It may be unable to hire anyone, because the people hired would not be sure they would get paid. Investors would not

fund the company because of the likelihood of lawsuits by unpaid employees. The risk arises from the environments in which the company functions.

Second, the risks change with time. If a company's network is not connected to the Internet, there seems to be no risk of attacks from other hosts on the Internet. However, despite any policies to the contrary, someone could connect a modem to one of the company computers and connect to the Internet through the modem. Should this happen, any risk analysis predicated on isolation from the Internet would no longer be accurate. Although policies can forbid the connection of such a modem and procedures can be put in place to make such connection difficult, unless the responsible parties can guarantee that no such modem will ever be installed, the risks can change.

Third, many risks are quite remote but still exist. In the modem example, the company has sought to minimize the risk of an Internet connection. Hence, this risk is "acceptable" but not nonexistent. As a practical matter, one does not worry about acceptable risks; instead, one worries that the risk will become unacceptable.

Finally, the problem of "analysis paralysis" refers to making risk analyses with no effort to act on those analyses. To change the example slightly, suppose the company performs a risk analysis. The executives decide that they are not sure if all risks have been found, so they order a second study to verify the first. They reconcile the studies then wait for some time to act on these analyses. At that point, the security officers raise the objection that the conditions in the workplace are no longer those that held when the original risk analyses were done. The analysis is repeated. But the company cannot decide how to ameliorate the risks, so it waits until a plan of action can be developed, and the process continues. The point is that the company is paralyzed and cannot act on the risks it faces.

1.6.3 Laws and Customs

Laws restrict the availability and use of technology and affect procedural controls. Hence, any policy and any selection of mechanisms must take into account legal considerations.

EXAMPLE: Until the year 2000, the United States controlled the export of cryptographic hardware and software (considered munitions under United States law). If a U.S. software company worked with a computer manufacturer in London, the U.S. company could not send cryptographic software to the manufacturer. The U.S. company first would have to obtain a license to export the software from the United States. Any security policy that depended on the London manufacturer using that cryptographic software would need to take this into account.

EXAMPLE: Suppose the law makes it illegal to read a user's file without the user's permission. An attacker breaks into the system and begins to download users' files. If the system administrators notice this and observe what the attacker is reading, they will be reading the victim's files without his permission and therefore will be violat-

ing the law themselves. For this reason, most sites require users to give (implicit or explicit) permission for system administrators to read their files. In some jurisdictions, an explicit exception allows system administrators to access information on their systems without permission in order to protect the quality of service provided or to prevent damage to their systems.

Complicating this issue are situations involving the laws of multiple jurisdictions—especially foreign ones.

EXAMPLE: In the 1990s, the laws involving the use of cryptography in France were very different from those in the United States. The laws of France required companies sending enciphered data out of the country to register their cryptographic keys with the government. Security procedures involving the transmission of enciphered data from a company in the United States to a branch office in France had to take these differences into account.

EXAMPLE: If a policy called for prosecution of attackers and intruders came from Russia to a system in the United States, prosecution would involve asking the United States authorities to extradite the alleged attackers from Russia. This undoubtedly would involve court testimony from company personnel involved in handling the intrusion, possibly trips to Russia, and more court time once the extradition was completed. The cost of prosecuting the attackers might be considerably higher than the company would be willing (or able) to pay.

Laws are not the only constraints on policies and selection of mechanisms. Society distinguishes between *legal* and *acceptable* practices. It may be legal for a company to require all its employees to provide DNA samples for authentication purposes, but it is not socially acceptable. Requiring the use of Social Security numbers as passwords is legal (unless the computer is one owned by the U.S. government) but also unacceptable. These practices provide security but at an unacceptable cost, and they encourage users to evade or otherwise overcome the security mechanisms.

The issue that laws and customs raise is the issue of psychological acceptability. A security mechanism that would put users and administrators at legal risk would place a burden on these people that few would be willing to bear; thus, such a mechanism would not be used. An unused mechanism is worse than a nonexistent one, because it gives a false impression that a security service is available. Hence, users may rely on that service to protect their data, when in reality their data is unprotected.

1.7 Human Issues

Implementing computer security controls is complex, and in a large organization procedural controls often become vague or cumbersome. Regardless of the strength

of the technical controls, if nontechnical considerations affect their implementation and use, the effect on security can be severe. Moreover, if configured or used incorrectly, even the best security control is useless at best and dangerous at worst. Thus, the designers, implementers, and maintainers of security controls are essential to the correct operation of those controls.

1.7.1 Organizational Problems

Security provides no direct financial rewards to the user. It limits losses, but it also requires the expenditure of resources that could be used elsewhere. Unless losses occur, organizations often believe they are wasting effort related to security. After a loss, the value of these controls suddenly becomes appreciated. Furthermore, security controls often add complexity to otherwise simple operations. For example, if concluding a stock trade takes two minutes without security controls and three minutes with security controls, adding those controls results in a 50% loss of productivity.

Losses occur when security protections are in place, but such losses are expected to be less than they would have been without the security mechanisms. The key question is whether such a loss, combined with the resulting loss in productivity, would be greater than a financial loss or loss of confidence should one of the nonsecured transactions suffer a breach of security.

Compounding this problem is the question of who is responsible for the security of the company's computers. The power to implement appropriate controls must reside with those who are responsible; the consequence of not doing so is that the people who can most clearly see the need for security measures, and who are responsible for implementing them, will be unable to do so. This is simply sound business practice; responsibility without power causes problems in any organization, just as does power without responsibility.

Once clear chains of responsibility and power have been established, the need for security can compete on an equal footing with other needs of the organization. The most common problem a security manager faces is the lack of people trained in the area of computer security. Another common problem is that knowledgeable people are overloaded with work. At many organizations, the "security administrator" is also involved in system administration, development, or some other secondary function. In fact, the security aspect of the job is often secondary. The problem is that indications of security problems often are not obvious and require time and skill to spot. Preparation for an attack makes dealing with it less chaotic, but such preparation takes enough time and requires enough attention so that treating it as a secondary aspect of a job means that it will not be performed well, with the expected consequences.

Lack of resources is another common problem. Securing a system requires resources as well as people. It requires time to design a configuration that will provide an adequate level of security, to implement the configuration, and to administer the system. It requires money to purchase products that are needed to build an adequate security system or to pay someone else to design and implement security mea-

sures. It requires computer resources to implement and execute the security mechanisms and procedures. It requires training to ensure that employees understand how to use the security tools, how to interpret the results, and how to implement the nontechnical aspects of the security policy.

1.7.2 People Problems

The heart of any security system is people. This is particularly true in computer security, which deals mainly with technological controls that can usually be bypassed by human intervention. For example, a computer system authenticates a user by asking that user for a secret code; if the correct secret code is supplied, the computer assumes that the user is authorized to use the system. If an authorized user tells another person his secret code, the unauthorized user can masquerade as the authorized user with significantly less likelihood of detection.

People who have some motive to attack an organization and are not authorized to use that organization's systems are called *outsiders* and can pose a serious threat. Experts agree, however, that a far more dangerous threat comes from disgruntled employees and other *insiders* who are authorized to use the computers. Insiders typically know the organization of the company's systems and what procedures the operators and users follow and often know enough passwords to bypass many security controls that would detect an attack launched by an outsider. Insider *misuse* of authorized privileges is a very difficult problem to solve.

Untrained personnel also pose a threat to system security. As an example, one operator did not realize that the contents of backup tapes needed to be verified before the tapes were stored. When attackers deleted several critical system files, she discovered that none of the backup tapes could be read.

System administrators who misread the output of security mechanisms, or do not analyze that output, contribute to the probability of successful attacks against their systems. Similarly, administrators who misconfigure security-related features of a system can weaken the site security. Users can also weaken site security by misusing security mechanisms (such as selecting passwords that are easy to guess).

Lack of training need not be in the technical arena. Many successful break-ins have arisen from the art of *social engineering*. If operators will change passwords based on telephone requests, all an attacker needs to do is to determine the name of someone who uses the computer. A common tactic is to pick someone fairly far above the operator (such as a vice president of the company) and to feign an emergency (such as calling at night and saying that a report to the president of the company is due the next morning) so that the operator will be reluctant to refuse the request. Once the password has been changed to one that the attacker knows, he can simply log in as a normal user. Social engineering attacks are remarkably successful and often devastating.

The problem of misconfiguration is aggravated by the complexity of many security-related configuration files. For instance, a typographical error can disable key protection features. Even worse, software does not always work as advertised.

One widely used system had a vulnerability that arose when an administrator made too long a list that named systems with access to certain files. Because the list was too long, the system simply assumed that the administrator meant to allow those files to be accessed without restriction on who could access them—exactly the opposite of what was intended.

1.8 Tying It All Together

The considerations discussed above appear to flow linearly from one to the next (see Figure 1–1). Human issues pervade each stage of the cycle. In addition, each stage of the cycle feeds back to the preceding stage, and through that stage to all earlier stages. The operation and maintenance stage is critical to the life cycle. Figure 1–1 breaks it out so as to emphasize the impact it has on all stages. The following example shows the importance of feedback.

EXAMPLE: A major corporation decided to improve its security. It hired consultants, determined the threats, and created a policy. From the policy, the consultants derived several specifications that the security mechanisms had to meet. They then developed a design that would meet the specifications.

During the implementation phase, the company discovered that employees could connect modems to the telephones without being detected. The design required

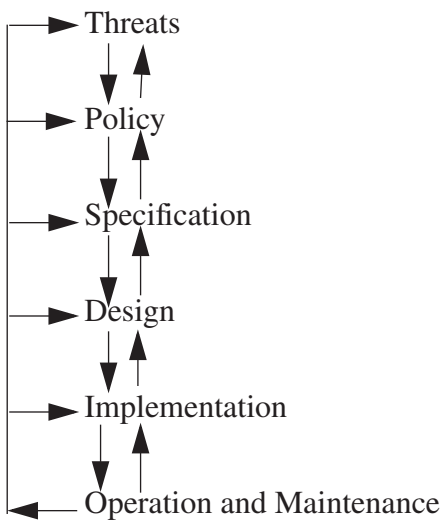

Figure 1–1 The security life cycle.

all incoming connections to go through a firewall. The design had to be modified to divide systems into two classes: systems connected to "the outside," which were put outside the firewall; and all other systems, which were put behind the firewall. The design needed other modifications as well.

When the system was deployed, the operation and maintenance phase revealed several unexpected threats. The most serious was that systems were repeatedly misconfigured to allow sensitive data to be sent across the Internet in the clear. The implementation made use of cryptographic software very difficult. Once this problem had been remedied, the company discovered that several "trusted" hosts (those allowed to log in without authentication) were physically outside the control of the company. This violated policy, but for commercial reasons the company needed to continue to use these hosts. The policy element that designated these systems as "trusted" was modified. Finally, the company detected proprietary material being sent to a competitor over electronic mail. This added a threat that the company had earlier discounted. The company did not realize that it needed to worry about insider attacks.

Feedback from operation is critical. Whether or not a program is tested or proved to be secure, operational environments always introduce unexpected problems or difficulties. If the assurance (specification, design, implementation, and testing/proof) phase is done properly, the extra problems and difficulties are minimal. The analysts can handle them, usually easily and quickly. If the assurance phase has been omitted or done poorly, the problems may require a complete reevaluation of the system. The tools used for the feedback include auditing, in which the operation of the system is recorded and analyzed so that the analyst can determine what the problems are.

1.9 Summary

Computer security depends on many aspects of a computer system. The threats that a site faces, and the level and quality of the countermeasures, depend on the quality of the security services and supporting procedures. The specific mix of these attributes is governed by the site security policy, which is created after careful analysis of the value of the resources on the system or controlled by the system and of the risks involved.

Underlying all this are key assumptions describing what the site and the system accept as true or trustworthy; understanding these assumptions is the key to analyzing the strength of the system's security. This notion of "trust" is the central notion for computer security. If trust is well placed, any system can be made acceptably secure. If it is misplaced, the system cannot be secure in any sense of the word.

Once this is understood, the reason that people consider security to be a relative attribute is plain. Given enough resources, an attacker can often evade the security procedures and mechanisms that are in place. Such a desire is tempered by the cost of the attack, which in some cases can be very expensive. If it is less expensive to regenerate the data than to launch the attack, most attackers will simply regenerate the data.

This chapter has laid the foundation for what follows. All aspects of computer security begin with the nature of threats and countering security services. In future chapters, we will build on these basic concepts.

1.10 Further Reading

Risk analysis arises in a variety of contexts. Molak [646] presents essays on risk management and analysis in a variety of fields. Laudan [552] provides an enjoyable introduction to the subject. Neumann [688] discusses the risks of technology and recent problems. Software safety (Leveson [557]) requires an understanding of the risks posed in the environment. Peterson [717] discusses many programming errors in a readable way. All provide insights into the problems that arise in a variety of environments.

Many authors recount stories of security incidents. The earliest, Parker's wonderful book [713], discusses motives and personalities as well as technical details. Stoll recounts the technical details of uncovering an espionage ring that began as the result of a 75¢ accounting error [878, 880]. Hafner and Markoff describe the same episode in a study of "cyberpunks" [386]. The Internet worm [292, 386, 757, 858] brought the problem of computer security into popular view. Numerous other incidents [339, 386, 577, 821, 838, 873] have heightened public awareness of the problem.

Several books [55, 57, 737, 799] discuss computer security for the layperson. These works tend to focus on attacks that are visible or affect the end user (such as pornography, theft of credit card information, and deception). They are worth reading for those who wish to understand the results of failures in computer security.

1.11 Exercises

1. Classify each of the following as a violation of confidentiality, of integrity, of availability, or of some combination thereof.
   a. John copies Mary's homework.
   b. Paul crashes Linda's system.
   c. Carol changes the amount of Angelo's check from $100 to $1,000.
   d. Gina forges Roger's signature on a deed.
   e. Rhonda registers the domain name "AddisonWesley.com" and refuses to let the publishing house buy or use that domain name.
   f. Jonah obtains Peter's credit card number and has the credit card company cancel the card and replace it with another card bearing a different account number.

g. Henry spoofs Julie's IP address to gain access to her computer.

2. Identify mechanisms for implementing the following. State what policy or policies they might be enforcing.
   a. A password-changing program will reject passwords that are less than five characters long or that are found in the dictionary.
   b. Only students in a computer science class will be given accounts on the department's computer system.
   c. The login program will disallow logins of any students who enter their passwords incorrectly three times.
   d. The permissions of the file containing Carol's homework will prevent Robert from cheating and copying it.
   e. When World Wide Web traffic climbs to more than 80% of the network's capacity, systems will disallow any further communications to or from Web servers.
   f. Annie, a systems analyst, will be able to detect a student using a program to scan her system for vulnerabilities.
   g. A program used to submit homework will turn itself off just after the due date.
3. The aphorism "security through obscurity" suggests that hiding information provides some level of security. Give an example of a situation in which hiding information does not add appreciably to the security of a system. Then give an example of a situation in which it does.
4. Give an example of a situation in which a compromise of confidentiality leads to a compromise in integrity.
5. Show that the three security services—confidentiality, integrity, and availability—are sufficient to deal with the threats of disclosure, disruption, deception, and usurpation.
6. In addition to mathematical and informal statements of policy, policies can be implicit (not stated). Why might this be done? Might it occur with informally stated policies? What problems can this cause?
7. For each of the following statements, give an example of a situation in which the statement is true.
   a. Prevention is more important than detection and recovery.
   b. Detection is more important than prevention and recovery.
   c. Recovery is more important than prevention and detection.
8. Is it possible to design and implement a system in which *no* assumptions about trust are made? Why or why not?
9. Policy restricts the use of electronic mail on a particular system to faculty and staff. Students cannot send or receive electronic mail on that host. Classify the following mechanisms as secure, precise, or broad.

a. The electronic mail sending and receiving programs are disabled.
b. As each letter is sent or received, the system looks up the sender (or recipient) in a database. If that party is listed as faculty or staff, the mail is processed. Otherwise, it is rejected. (Assume that the database entries are correct.)
c. The electronic mail sending programs ask the user if he or she is a student. If so, the mail is refused. The electronic mail receiving programs are disabled.

10. Consider a very high-assurance system developed for the military. The system has a set of specifications, and both the design and implementation have been proven to satisfy the specifications. What questions should school administrators ask when deciding whether to purchase such a system for their school's use?
11. How do laws protecting privacy impact the ability of system administrators to monitor user activity?
12. Computer viruses are programs that, among other actions, can delete files without a user's permission. A U.S. legislator wrote a law banning the deletion of any files from computer disks. What was the problem with this law from a computer security point of view? Specifically, state which security service would have been affected if the law had been passed.
13. Users often bring in programs or download programs from the Internet. Give an example of a site for which the benefits of allowing users to do this outweigh the dangers. Then give an example of a site for which the dangers of allowing users to do this outweigh the benefits.
14. A respected computer scientist has said that no computer can ever be made perfectly secure. Why might she have said this?
15. An organization makes each lead system administrator responsible for the security of the system he or she runs. However, the management determines what programs are to be on the system and how they are to be configured.
    a. Describe the security problem(s) that this division of power would create.
    b. How would you fix them?
16. The president of a large software development company has become concerned about competitors learning proprietary information. He is determined to stop them. Part of his security mechanism is to require all employees to report any contact with employees of the company's competitors, even if it is purely social. Do you believe this will have the desired effect? Why or why not?

17. The police and the public defender share a computer. What security problems does this present? Do you feel it is a reasonable cost-saving measure to have all public agencies share the same (set of) computers?
18. Companies usually restrict the use of electronic mail to company business but do allow minimal use for personal reasons.
    a. How might a company detect excessive personal use of electronic mail, other than by reading it? (*Hint*: Think about the personal use of a company telephone.)
    b. Intuitively, it seems reasonable to ban *all* personal use of electronic mail on company computers. Explain why most companies do not do this.
19. Argue for or against the following proposition. Ciphers that the government cannot cryptanalyze should be outlawed. How would your argument change if such ciphers could be used provided that the users registered the keys with the government?
20. For many years, industries and financial institutions hired people who broke into their systems once those people were released from prison. Now, such a conviction tends to prevent such people from being hired. Why you think attitudes on this issue changed? Do you think they changed for the better or for the worse?
21. A graduate student accidentally releases a program that spreads from computer system to computer system. It deletes no files but requires much time to implement the necessary defenses. The graduate student is convicted. Despite demands that he be sent to prison for the maximum time possible (to make an example of him), the judge sentences him to pay a fine and perform community service. What factors do you believe caused the judge to hand down the sentence he did? What would you have done were you the judge, and what extra information would you have needed to make your decision?

Chapter 15
Information Flow

BOTTOM: Masters, I am to discourse wonders: but
ask me not what; for if I tell you, I am no true
Athenian. I will tell you every thing, right as it
fell out.
—*A Midsummer Night's Dream*, IV, ii, 30–33.

Although access controls can constrain the rights of a user, they cannot constrain the flow of information about a system. In particular, when a system has a security policy regulating information flow, the system must ensure that the information flows do not violate the constraints of the policy. Both compile-time mechanisms and runtime mechanisms support the checking of information flows. Several systems implementing these mechanisms demonstrate their effectiveness.

15.1 Basics and Background

Information flow policies define the way information moves throughout a system. Typically, these policies are designed to preserve confidentiality of data or integrity of data. In the former, the policy's goal is to prevent information from flowing to a user not authorized to receive it. In the latter, information may flow only to processes that are no more trustworthy than the data.

Any confidentiality and integrity policy embodies an information flow policy.

EXAMPLE: The Bell-LaPadula Model describes a lattice-based information flow policy. Given two compartments A and B, information can flow from an object in A to a subject in B if and only if B dominates A.

Let x be a variable in a program. The notation $\underline{x}$ refers to the information flow class of x.

EXAMPLE: Consider a system that uses the Bell-LaPadula Model. The variable *x,* which holds data in the compartment (TS, { NUC, EUR }), is set to 3. Then $x = 3$ and $\underline{x}$ = (TS, { NUC, EUR }).

Intuitively, information flows from an object *x* to an object *y* if the application of a sequence of commands *c* causes the information initially in *x* to affect the information in *y*.

> **Definition 15–1.** The command sequence *c* causes a *flow of information from x to y* if, after execution of *c*, some information about the value of *x* before *c* was executed can be deduced from the value of *y* after *c* was executed.

This definition views information flow in terms of the information that the value of *y* allows one to deduce about the value in *x*. For example, the statement

```
y := x;
```

reveals the value of *x* in the initial state, so information about the value of *x* in the initial state can be deduced from the value of *y* after the statement is executed. The statement

```
y := x / z;
```

reveals some information about *x*, but not as much as the first statement.

The final result of the sequence *c* must reveal information about the initial value of *x* for information to flow. The sequence

```
tmp := x;
y := tmp;
```

has information flowing from *x* to *y* because the (unknown) value of *x* at the beginning of the sequence is revealed when the value of *y* is determined at the end of the sequence. However, no information flow occurs from *tmp* to *x*, because the initial value of *tmp* cannot be determined at the end of the sequence.

EXAMPLE: Consider the statement

```
x := y + z;
```

Let *y* take any of the integer values from 0 to 7, inclusive, with equal probability, and let *z* take the value 1 with probability 0.5 and the values 2 and 3 with probability 0.25 each. Once the resulting value of *x* is known,the initial value of *y* can assume at most three values. Thus, information flows from *y* to *x*. Similar results hold for *z*.

EXAMPLE: Consider a program in which x and y are integers that may be either 0 or 1. The statement

```
if x = 1 then y := 0;
else y := 1;
```

does not explicitly assign the value of x to y.

Assume that x is equally likely to be 0 or 1. Then $\mathrm{H}(x_s) = 1$. But $H(x_s \mid y_t) = 0$, because if y is 0, x is 1, and vice versa. Hence, $H(x_s \mid y_t) = 0 < H(x_s \mid y_s) = H(x_s) = 1$. Thus, information flows from x to y.

Definition 15–2. An *implicit flow of information* occurs when information flows from x to y without an explicit assignment of the form $y := f(x)$, where $f(x)$ is an arithmetic expression with the variable x.

The flow of information occurs, not because of an assignment of the value of x, but because of a flow of control based on the value of x. This demonstrates that analyzing programs for assignments to detect information flows is not enough. To detect all flows of information, implicit flows must be examined.

15.1.1 Information Flow Models and Mechanisms

An information flow policy is a security policy that describes the authorized paths along which that information can flow. Each model associates a label, representing a security class, with information and with entities containing that information. Each model has rules about the conditions under which information can move throughout the system.

In this chapter, we use the notation $\underline{x} \leq \underline{y}$ to mean that information can flow from an element of class x to an element of class y. Equivalently, this says that information with a label placing it in class $\underline{x}$ can flow into class $\underline{y}$.

Earlier chapters usually assumed that the models of information flow policies were lattices. We first consider nonlattice information flow policies and how their structures affect the analysis of information flow. We then turn to compiler-based information flow mechanisms and runtime mechanisms. We conclude with a look at flow controls in practice.

15.2 Compiler-Based Mechanisms

Compiler-based mechanisms check that information flows throughout a program are authorized. The mechanisms determine if the information flows in a program *could* violate a given information flow policy. This determination is not precise, in that

secure paths of information flow may be marked as violating the policy; but it is secure, in that no unauthorized path along which information may flow will be undetected.

> **Definition 15–3.** A set of statements is *certified* with respect to an information flow policy if the information flow within that set of statements does not violate the policy.

EXAMPLE: Consider the program statement

```
if x = 1 then y := a;
else y := b;
```

By the rules discussed earlier, information flows from x and a to y or from x and b to y, so if the policy says that $\underline{a} \le \underline{y}$, $\underline{b} \le \underline{y}$, and $\underline{x} \le \underline{y}$, then the information flow is secure. But if $\underline{a} \le \underline{y}$ only when some other variable $z = 1$, the compiler-based mechanism must determine whether $z = 1$ before certifying the statement. Typically, this is infeasible. Hence, the compiler-based mechanism would not certify the statement. The mechanisms described here follow those developed by Denning and Denning [247] and Denning [242].

15.2.1 Declarations

For our discussion, we assume that the allowed flows are supplied to the checking mechanisms through some external means, such as from a file. The specifications of allowed flows involve security classes of language constructs. The program involves variables, so some language construct must relate variables to security classes. One way is to assign each variable to exactly one security class. We opt for a more liberal approach, in which the language constructs specify the set of classes from which information may flow into the variable. For example,

```
x: integer class { A, B }
```

states that x is an integer variable and that data from security classes A and B may flow into x. Note that the classes are statically, not dynamically, assigned. Viewing the security classes as a lattice, this means that x's class must be at least the least upper bound of classes A and B—that is, $lub\{A, B\} \le \underline{x}$.

Two distinguished classes, *Low* and *High*, represent the greatest lower bound and least upper bound, respectively, of the lattice. All constants are of class *Low*.

Information can be passed into or out of a procedure through parameters. We classify parameters as *input parameters* (through which data is passed into the procedure), *output parameters* (through which data is passed out of the procedure), and *input/output parameters* (through which data is passed into and out of the procedure).

```
(* input parameters are named i_s; output parameters, o_s; *)
(* and input/output parameters, io_s, with s a subscript *)
proc something(i_1, ..., i_k; var o_1, ..., o_m, io_1, ..., io_n);
var l_1, ..., l_j;                (* local variables *)
begin
        S;                        (* body of procedure *)
end;
```

The class of an input parameter is simply the class of the actual argument:

i_s: *type* **class** { i_s }

Let r_1, ..., r_p be the set of input and input/output variables from which information flows to the output variable o_s. The declaration for the type must capture this:

o_s: *type* **class** { r_1, ..., r_p }

(We implicitly assume that any output-only parameter is initialized in the procedure.) The input/output parameters are like output parameters, except that the initial value (as input) affects the allowed security classes. Again, let r_1, ..., r_p be defined as above. Then:

io_s: *type* **class** {r_1, ..., r_p, io_1, ..., io_k }

EXAMPLE: Consider the following procedure for adding two numbers.

```
proc sum(x: int class { x };
             var out: int class { x, out });
begin
        out := out + x;
end;
```

Here, we require that $x \leq out$ and $out \leq out$ (the latter holding because $\leq$ is reflexive).

The declarations presented so far deal only with basic types, such as integers, characters, floating point numbers, and so forth. Nonscalar types, such as arrays, records (structures), and variant records (unions) also contain information. The rules for information flow classes for these data types are built on the scalar types.

Consider the array

```
a: array 1 .. 100 of int;
```

First, look at information flows out of an element $a[i]$ of the array. In this case, information flows from $a[i]$ and from i, the latter by virtue of the index indicating

which element of the array to use. Information flows into $a[i]$ affect only the value in $a[i]$, and so do not affect the information in i. Thus, for information flows from $a[i]$, the class involved is $lub\{ \underline{a[i]}, \underline{i} \}$; for information flows into $a[i]$, the class involved is $\underline{a[i]}$.

15.2.2 Program Statements

A program consists of several types of statements. Typically, they are

1. Assignment statements
2. Compound statements
3. Conditional statements
4. Iterative statements
5. Goto statements
6. Procedure calls
7. Function calls
8. Input/output statements.

We consider each of these types of statements separately, with two exceptions. Function calls can be modeled as procedure calls by treating the return value of the function as an output parameter of the procedure. Input/output statements can be modeled as assignment statements in which the value is assigned to (or assigned from) a file. Hence, we do not consider function calls and input/output statements separately.

15.2.2.1 Assignment Statements

An assignment statement has the form

```
y := f(x1, ..., xn)
```

where y and x_1, ..., x_n are variables and f is some function of those variables. Information flows from each of the x_i's to y. Hence, the requirement for the information flow to be secure is

- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq \underline{y}$

EXAMPLE: Consider the statement

```
x := y + z;
```

Then the requirement for the information flow to be secure is $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$.

15.2.2.2 Compound Statements

A compound statement has the form

```
begin
     S1;
     ...
     Sn;
end;
```

where each of the S_i's is a statement. If the information flow in each of the statements is secure, then the information flow in the compound statement is secure. Hence, the requirements for the information flow to be secure are

- S_1 secure
- ...
- S_n secure

EXAMPLE: Consider the statements

```
begin
     x := y + z;
     a := b * c - x;
end;
```

Then the requirements for the information flow to be secure are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$ for S_2. So, the requirements for secure information flow are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$ and $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$.

15.2.2.3 Conditional Statements

A conditional statement has the form

```
if f(x1, ..., xn) then
     S1;
else
     S2;
end;
```

where $x_1, \ldots, x_n$ are variables and f is some (boolean) function of those variables. Either S_1 or S_2 may be executed, depending on the value of f, so both must be secure. As discussed earlier, the selection of either S_1 or S_2 imparts information about the values of the variables x_1, ..., x_n, so information must be able to flow from those variables to any targets of assignments in S_1 and S_2. This is possible if and only if the

lowest class of the targets dominates the highest class of the variables x_1, ..., x_n. Thus, the requirements for the information flow to be secure are

- S_1 secure
- S_2 secure
- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq glb\{\ \underline{y} \mid y$ is the target of an assignment in S_1 and $S_2\ \}$

As a degenerate case, if statement S_2 is empty, it is trivially secure and has no assignments.

EXAMPLE: Consider the statements

```
if x + y < z then
     a := b;
else
     d := b * c - x;
end;
```

Then the requirements for the information flow to be secure are $\underline{b} \leq \underline{a}$ for S_1 and $lub\{\ \underline{b}, \underline{c}, \underline{x}\ \} \leq \underline{d}$ for S_2. But the statement that is executed depends on the values of x, y, and z. Hence, information also flows from x, y, and z to d and a. So, the requirements are $lub\{\ \underline{y}, \underline{z}\ \} \leq \underline{x}$, $\underline{b} \leq a$, and $lub\{\ \underline{x}, \underline{y}, \underline{z}\ \} \leq glb\{\ \underline{a}, \underline{d}\ \}$.

15.2.2.4 Iterative Statements

An iterative statement has the form

```
while f(x1, ..., xn) do
     S;
```

where x_1, ..., x_n are variables and f is some (boolean) function of those variables. Aside from the repetition, this is a conditional statement, so the requirements for information flow to be secure for a conditional statement apply here.

To handle the repetition, first note that the number of repetitions causes information to flow only through assignments to variables in S. The number of repetitions is controlled by the values in the variables x_1, ..., x_n, so information flows from those variables to the targets of assignments in S—but this is detected by the requirements for information flow of conditional statements.

However, if the program never leaves the iterative statement, statements after the loop will never be executed. In this case, information has flowed from the variables x_1, ..., x_n by the *absence* of execution. Hence, secure information flow also requires that the loop terminate.

Thus, the requirements for the information flow to be secure are

- Iterative statement terminates
- S secure
- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq glb\{\ \underline{y} \mid y$ is the target of an assignment in $S\ \}$

EXAMPLE: Consider the statements

```
while i < n do
begin
     a[i] := b[i];
     i := i + 1;
end;
```

This loop terminates. If $n \leq i$ initially, the loop is never entered. If $i < n$, i is incremented by a positive integer, 1, and so increases, at each iteration. Hence, after $n - i$ iterations, $n = i$, and the loop terminates.

Now consider the compound statement that makes up the body of the loop. The first statement is secure if $\underline{i} \leq \underline{a[i]}$ and $\underline{b[i]} \leq \underline{a[i]}$; the second statement is secure because $\underline{i} \leq \underline{i}$. Hence, the compound statement is secure if $lub\{\ \underline{i}, \underline{b[i]}\ \} \leq \underline{a[i]}$.

Finally, $a[i]$ and i are targets of assignments in the body of the loop. Hence, information flows into them from the variables in the expression in the *while* statement. So, $lub\{\ \underline{i}, \underline{n}\ \} \leq glb\{\ \underline{a[i]}, \underline{i}\ \}$. Putting these together, the requirement for the information flow to be secure is $lub\{\ \underline{b[i]}, \underline{i}, \underline{n}\ \} \leq glb\{\ \underline{a[i]}, \underline{i}\ \}$ (see Exercise 2).

15.2.2.5 Goto Statements

A goto statement contains no assignments, so no explicit flows of information occur. Implicit flows may occur; analysis detects these flows.

Definition 15–4. A *basic block* is a sequence of statements in a program that has one entry point and one exit point.

EXAMPLE: Consider the following code fragment.

```
proc transmatrix(x: array [1..10][1..10] of int class { x };
                 var y: array [1..10][1..10] of int class { y } );
     var i, j: int class { tmp };
     begin
         i := 1;                        (* b1 *)
     l2: if i > 10 goto l7;             (* b2 *)
         j := 1;                        (* b3 *)
     l4: if j > 10 then goto l6;        (* b4 *)
```

```
        y[j][i] := x[i][j];        (* b5 *)
        j := j + 1;
        goto l4;
-------------------------------------------
    l6: i := i + 1;                (* b6 *)
        goto l2;
-------------------------------------------
    l7:                            (* b7 *)
end;
```

There are seven basic blocks, labeled b_1 through b_7 and separated by lines. The second and fourth blocks have two ways to arrive at the entry—either from a jump to the label or from the previous line. They also have two ways to exit—either by the branch or by falling through to the next line. The fifth block has three lines and always ends with a branch. The sixth block has two lines and can be entered either from a jump to the label or from the previous line. The last block is always entered by a jump.

Control within a basic block flows from the first line to the last. Analyzing the flow of control within a program is therefore equivalent to analyzing the flow of control among the program's basic blocks. Figure 15–1 shows the flow of control among the basic blocks of the body of the procedure *transmatrix*.

When a basic block has two exit paths, the block reveals information implicitly by the path along which control flows. When these paths converge later in the program, the (implicit) information flow derived from the exit path from the basic block becomes either explicit (through an assignment) or irrelevant. Hence, the class

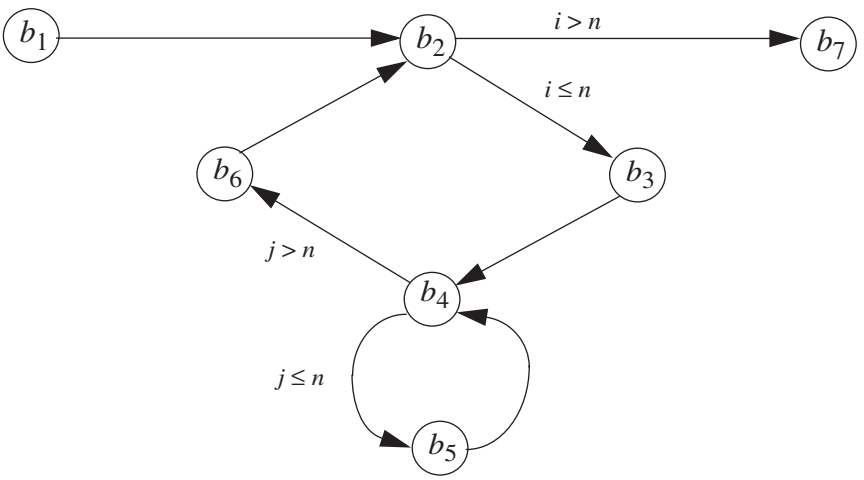

Figure 15–1 The control flow graph of the procedure *transmatrix*. The basic blocks are labeled b_1 through b_7. The conditions under which branches are taken are shown over the edges corresponding to the branches.

of the expression that causes a particular execution path to be selected affects the required classes of the blocks along the path up to the block at which the divergent paths converge.

> **Definition 15–5.** An *immediate forward dominator* of a basic block b (written $IFD(b)$) is the first block that lies on all paths of execution that pass through b.

EXAMPLE: In the procedure *transmatrix*, the immediate forward dominators of each block are $IFD(b_1) = b_2$, $IFD(b_2) = b_7$, $IFD(b_3) = b_4$, $IFD(b_4) = b_6$, $IFD(b_5) = b_4$, and $IFD(b_6) = b_2$.

Computing the information flow requirement for the set of blocks along the path is now simply applying the logic for the conditional statement. Each block along the path is taken because of the value of an expression. Information flows from the variables of the expression into the set of variables assigned in the blocks. Let B_i be the set of blocks along an execution path from b_i to $IFD(b_i)$, but excluding these endpoints. (See Exercise 3.) Let x_{i1}, ..., x_{in} be the set of variables in the expression that selects the execution path containing the blocks in B_i. The requirements for the program's information flows to be secure are

- All statements in each basic block secure
- $lub\{\underline{x}_{i1}, ..., \underline{x}_{in}\} \le glb\{\ \underline{y} \mid y$ is the target of an assignment in $B_i\ \}$

EXAMPLE: Consider the body of the procedure *transmatrix*. We first state requirements for information flow within each basic block:

b_1: $Low \le \underline{i} \Rightarrow$ secure
b_3: $Low \le \underline{j} \Rightarrow$ secure
b_5: $lub\{\ \underline{x[i][j]}, \underline{i}, \underline{j}\ \} \le \underline{y[j][i]}; \underline{j} \le \underline{j} \Rightarrow lub\{\ \underline{x[i][j]}, \underline{i}, \underline{j}\ \} \le \underline{y[j][i]}$
b_6: $lub\{\ Low, \underline{i}\ \} \le \underline{i} \Rightarrow$ secure

The requirement for the statements in each basic block to be secure is, for $i = 1, ..., n$ and $j = 1, ..., n$, $lub\{\ \underline{x[i][j]}, \underline{i}, \underline{j}\ \} \le \underline{y[j][i]}$. By the declarations, this is true when $lub\{\underline{x}, \underline{i}\} \le \underline{y}$.

In this procedure, $B_2 = \{\ b_3, b_4, b_5, b_6\ \}$ and $B_4 = \{\ b_5\ \}$. Thus, in B_2, statements assign values to i, j, and $y[j][i]$. In B_4, statements assign values to j and $y[j][i]$. The expression controlling which basic blocks in B_2 are executed is $i \le 10$; the expression controlling which basic blocks in B_4 are executed is $j \le 10$. Secure information flow requires that $\underline{i} \le glb\{\ \underline{i}, \underline{j}, \underline{y[j][i]}\}$ and $\underline{j} \le glb\{\ \underline{j}, \underline{y[j][i]}\ \}$. In other words, $\underline{i} \le glb\{\ \underline{i}, \underline{y}\ \}$ and $\underline{i} \le glb\{\ \underline{i}, \underline{y}\ \}$, or $\underline{i} \le \underline{y}$.

Combining these requirements, the requirement for the body of the procedure to be secure with respect to information flow is $lub\{\underline{x}, \underline{i}\} \le \underline{y}$.

15.2.2.6 Procedure Calls

A procedure call has the form

```
proc procname(i1, ..., im : int; var o1, ..., on : int);
begin
     S;
end;
```

where each of the i_j's is an input parameter and each of the o_j's is an input/output parameter. The information flow in the body S must be secure. As discussed earlier, information flow relationships may also exist between the input parameters and the output parameters. If so, these relationships are necessary for S to be secure. The actual parameters (those variables supplied in the call to the procedure) must also satisfy these relationships for the call to be secure. Let x_1, ..., x_m and y_1, ..., y_n be the actual input and input/output parameters, respectively. The requirements for the information flow to be secure are

- S secure
- For $j = 1, ..., m$ and $k = 1, ..., n$, if $\underline{i_j} \leq \underline{o_k}$ then $\underline{x_j} \leq \underline{y_k}$
- For $j = 1, ..., n$ and $k = 1, ..., n$, if $\underline{o_j} \leq \underline{o_k}$ then $\underline{y_j} \leq \underline{y_k}$

EXAMPLE: Consider the procedure *transmatrix* from the preceding section. As we showed there, the body of the procedure is secure with respect to information flow when $lub\{\underline{x}, \underline{tmp}\} \leq \underline{y}$. This indicates that the formal parameters x and y have the information flow relationship $\underline{x} \leq \underline{y}$. Now, suppose a program contains the call

```
transmatrix(a, b)
```

The second condition asserts that this call is secure with respect to information flow if and only if $\underline{a} \leq \underline{b}$.

15.2.3 Exceptions and Infinite Loops

Exceptions can cause information to flow.

EXAMPLE: Consider the following procedure, which copies the (approximate) value of x to y.[1]

```
proc copy(x: int class { x }; var y: int class Low);
var  sum: int class { x };
     z: int class Low;
```

[1] From Denning [242], p. 306.

```
begin
     z := 0;
     sum := 0;
     y := 0;
     while z = 0 do begin
          sum := sum + x;
          y := y + 1;
     end
end
```

When *sum* overflows, a trap occurs. If the trap is not handled, the procedure exits. The value of x is *MAXINT* / y, where *MAXINT* is the largest integer representable as an *int* on the system. At no point, however, is the flow relationship $\underline{x} \leq \underline{y}$ checked.

If exceptions are handled explicitly, the compiler can detect problems such as this. Denning again supplies such a solution.

EXAMPLE: Suppose the system ignores all exceptions unless the programmer specifically handles them. Ignoring the exception in the preceding example would cause the program to loop indefinitely. So, the programmer would want the loop to terminate when the exception occurred. The following line does this.

```
on overflowexception sum do z := 1;
```

This line causes information to flow from *sum* to z, meaning that $\underline{sum} \leq \underline{z}$. Because $\underline{z}$ is *Low* and $\underline{sum}$ is $\{ x \}$, this is incorrect and the procedure is not secure with respect to information flow.

Denning also notes that infinite loops can cause information to flow in unexpected ways.

EXAMPLE: The following procedure copies data from x to y. It assumes that x and y are either 0 or 1.

```
proc copy(x: int 0..1 class { x };
          var y: int 0..1 class Low);
begin
     y := 0;
     while x = 0 do
          (* nothing *);
     y := 1;
end.
```

If x is 0 initially, the procedure does not terminate. If x is 1, it does terminate, with y being 1. At no time is there an explicit flow from x to y. This is an example of a *covert channel*, which we will discuss in detail in the next chapter.

15.2.4 Concurrency

Of the many concurrency control mechanisms that are available, we choose to study information flow using semaphores [270]. Their operation is simple, and they can be used to express many higher-level constructs [135, 718]. The specific semaphore constructs are

```
wait(x): if x = 0 then block until x > 0; x := x - 1;
signal(x): x := x + 1;
```

where *x* is a semaphore. As usual, the *wait* and the *signal* are indivisible; once either one has started, no other instruction will execute until the *wait* or *signal* finishes.

Reitman and his colleagues [33, 748] point out that concurrent mechanisms add information flows when values common to multiple processes cause specific actions. For example, in the block

```
begin
     wait(sem);
     x := x + 1;
end;
```

the program blocks at the *wait* if *sem* is 0, and executes the next statement when *sem* is nonzero. The earlier certification requirement for compound statements is not sufficient because of the implied flow between *sem* and *x*. The certification requirements must take flows among local and shared variables (semaphores) into account.

Let the block be

```
begin
     S1;
     ...
     Sn;
end;
```

Assume that each of the statements S_1, ..., S_n is certified. Semaphores in the *signal* do not affect information flow in the program in which the *signal* occurs, because the *signal* statement does not block. But following a *wait* statement, which may block, information implicitly flows from the semaphore in the *wait* to the targets of successive assignments.

Let statement S_i be a *wait* statement, and let $shared(S_i)$ be the set of shared variables that are read (so information flows from them). Let $\underline{g(S_i)}$ be the greatest lower bound of the targets of assignments following S_i. A requirement that the block be secure is that $\underline{shared(S_i)} \leq \underline{g(S_i)}$. Thus, the requirements for certification of a compound statement with concurrent constructs are

- S_1 secure
- ...
- S_n secure
- For $i = 1, ..., n$ [$\underline{shared(S_i)} \leq \underline{g(S_i)}$]

EXAMPLE: Consider the statements

```
begin
     x := y + z;
     wait(sem);
     a := b * c - x;
end;
```

The requirements that the information flow be secure are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$ for S_2. Information flows implicitly from *sem* to a, so $\underline{sem} \leq \underline{a}$. The requirements for certification are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$, $lub\{ \underline{b}, \underline{c}, \underline{x} \} \leq \underline{a}$, and $\underline{sem} \leq \underline{a}$.

Loops are handled similarly. The only difference is in the last requirement, because after completion of one iteration of the loop, control may return to the beginning of the loop. Hence, a semaphore may affect assignments that precede the *wait* statement in which the semaphore is used. This simplifies the last condition in the compound statement requirement considerably. Information must be able to flow from all shared variables named in the loop to the targets of all assignments. Let $shared(S_i)$ be the set of shared variables read, and let $t_1, ..., t_m$ be the targets of assignments in the loop. Then the certification conditions for the iterative statement

```
while f(x1, ..., xn) do
     S;
```

are

- Iterative statement terminates
- S secure
- $lub\{\underline{x}_1, ..., \underline{x}_n\} \leq glb\{ t_1, ..., \underline{t}_m \}$
- $lub\{\underline{shared(S_1)}, ,,,, \underline{shared(S_n)} \} \leq glb\{ \underline{t}_1, ..., \underline{t}_m \}$

EXAMPLE: Consider the statements

```
while i < n do
begin
     a[i] := item;
     wait(sem);
     i := i + 1;
end;
```

This loop terminates. If $n \leq i$ initially, the loop is never entered. If $i < n$, i is incremented by a positive integer, 1, and so increases, at each iteration. Hence, after $n - i$ iterations, $n = i$, and the loop terminates.

Now consider the compound statement that makes up the body of the loop. The first statement is secure if $\underline{i} \leq \underline{a[i]}$ and $\underline{item} \leq \underline{a[i]}$.The third statement is secure because $\underline{i} \leq \underline{i}$. The second statement induces an implicit flow, so $\underline{sem} \leq \underline{a[i]}$ and $\underline{sem} \leq \underline{i}$. The requirements are thus $\underline{i} \leq \underline{a[i]}$, $\underline{item} \leq \underline{a[i]}$, $\underline{sem} \leq \underline{a[i]}$, and $\underline{sem} \leq \underline{i}$.

Finally, concurrent statements have no information flow among them per se. Any such flows occur because of semaphores and involve compound statements (discussed above). The certification conditions for the concurrent statement

```
cobegin
     S1;
     ...
     Sn;
coend;
```

are

- S_1 secure
- ...
- S_n secure

EXAMPLE: Consider the statements

```
cobegin
     x := y + z;
     a := b * c - y;
coend;
```

The requirements that the information flow be secure are $lub\{ \underline{y}, \underline{z} \} \leq \underline{x}$ for S_1 and $lub\{ \underline{b}, \underline{c}, \underline{y} \} \leq \underline{a}$ for S_2. The requirement for certification is simply that both of these requirements hold.

15.2.5 Soundness

Denning and Denning [247], Andrews and Reitman [33], and others build their argument for security on the intuition that combining secure information flows produces a secure information flow, for some security policy. However, they never formally prove this intuition. Volpano, Irvine, and Smith [920] express the semantics of the

above-mentioned information on flow analysis as a set of types, and equate certification that a certain flow can occur to the correct use of types. In this context, checking for valid information flows is equivalent to checking that variable and expression types conform to the semantics imposed by the security policy.

Let x and y be two variables in the program. Let x's label dominate y's label. A set of information flow rules is sound if the value in x cannot affect the value in y during the execution of the program. Volpano, Irvine, and Smith use language-based techniques to prove that, given a type system equivalent to the certification rules discussed above, all programs without type errors have the noninterference property described above. Hence, the information flow certification rules of Denning and of Andrews and Reitman are sound.

15.3 Execution-Based Mechanisms

The goal of an execution-based mechanism is to prevent an information flow that violates policy. Checking the flow requirements of explicit flows achieves this result for statements involving explicit flows. Before the assignment

```
y = f(x1, ..., xn)
```

is executed, the execution-based mechanism verifies that

$$lub(\underline{x}_1, ..., \underline{x}_n) \leq \underline{y}$$

If the condition is true, the assignment proceeds. If not, it fails. A naïve approach, then, is to check information flow conditions whenever an explicit flow occurs.

Implicit flows complicate checking.

EXAMPLE: Let x and y be variables. The requirement for certification for a particular statement y *op* x is that $\underline{x} \leq \underline{y}$. The conditional statement

```
if x = 1 then y := a;
```

causes a flow from x to y. Now, suppose that when $x \neq 1$, $\underline{x} = High$ and $\underline{y} = Low$. If flows were verified only when explicit, and $x \neq 1$, the implicit flow would not be checked. The statement may be incorrectly certified as complying with the information flow policy.

Fenton explored this problem using a special abstract machine.

15.3.1 Fenton's Data Mark Machine

Fenton [313] created an abstract machine called the *Data Mark Machine* to study handling of implicit flows at execution time. Each variable in this machine had an associated security class, or tag. Fenton also included a tag for the program counter (PC).

The inclusion of the PC allowed Fenton to treat implicit flows as explicit flows, because branches are merely assignments to the PC. He defined the semantics of the Data Mark Machine. In the following discussion, *skip* means that the instruction is not executed, *push*(x, $\underline{x}$) means to push the variable x and its security class $\underline{x}$ onto the program stack, and *pop*(x, $\underline{x}$) means to pop the top value and security class off the program stack and assign them to x and $\underline{x}$, respectively.

Fenton defined five instructions. The relationships between execution of the instructions and the classes of the variables are as follows.

1. The increment instruction

   ```
   x := x + 1
   ```

   is equivalent to

   ```
   if PC ≤ x then x := x + 1; else skip
   ```

2. The conditional instruction

   ```
   if x = 0 then goto n else x := x - 1
   ```

   is equivalent to

   ```
   if x = 0 then { push(PC, PC); PC = lub(PC, x); PC := n; }
   else          { if PC ≤ x then { x := x - 1; } else skip }
   ```

   This branches, and pushes the PC and its security class onto the program stack. (As is customary, the PC is incremented so that when it is popped, the instruction following the *if* statement is executed.) This captures the PC containing information from x (specifically, that x is 0) while following the **goto**.

3. The return

   ```
   return
   ```

   is equivalent to

   ```
   pop(PC, PC);
   ```

This returns control to the statement following the last *if* statement. Because the flow of control would have arrived at this statement, the PC no longer contains information about x, and the old class can be restored.

4. The branch instruction

```
if' x = 0 then goto n else x := x - 1
```

is equivalent to

```
if x = 0 then { if x ≤ PC then { PC := n; } else skip }
else          { if PC ≤ x then { x := x - 1; } else skip }
```

This branches without saving the PC on the stack. If the branch occurs, the PC is in a higher security class than the conditional variable x, so adding information from x to the PC does not change the PC's security class.

5. The halt instruction

```
halt
```

is equivalent to

```
if program stack empty then halt execution
```

The program stack being empty ensures that the user cannot obtain information by looking at the program stack after the program has halted (for example, to determine which *if* statement was last taken).

EXAMPLE: Consider the following program, in which x initially contains 0 or 1.[2]

```
1. if x = 0 then goto 4 else x := x - 1
2. if z = 0 then goto 6 else z := z - 1
3. halt
4. z := z + 1
5. return
6. y := y + 1
7. return
```

This program copies the value of x to y. Suppose that $x = 1$ initially. The following table shows the contents of memory, the security class of the PC at each step, and the corresponding certification check.

[2] From Denning [242], Figure 5.7, p. 290.

x	y	z	PC	$\underline{\text{PC}}$	*stack*	*certification check*
1	0	0	1	*Low*	—	
0	0	0	2	*Low*	—	$Low \le \underline{x}$
0	0	0	6	$\underline{x}$	(3, *Low*)	
0	1	0	7	$\underline{x}$	(3, *Low*)	$\underline{\text{PC}} \le \underline{y}$
0	1	0	3	*Low*	—	

Fenton's machine handles errors by ignoring them. Suppose that, in the program above, $\underline{y} \le \underline{x}$. Then at the fifth step, the certification check fails (because $\underline{\text{PC}} = \underline{x}$). So, the assignment is skipped, and at the end $y = 0$ regardless of the value of x. But if the machine reports errors, the error message informing the user of the failure of the certification check means that the program has attempted to execute step 6. It could do so only if it had taken the branch in step 2, meaning that $z = 0$. If $z = 0$, then the *else* branch of statement 1 could not have been taken, meaning that $x = 0$ initially.

To prevent this type of deduction, Fenton's machine continues executing in the face of errors, but ignores the statement that would cause the violation. This satisfies the requirements. Aborting the program, or creating an exception visible to the user, would also cause information to flow against policy.

The problem with reporting of errors is that a user with lower clearance than the information causing the error can deduce the information from knowing that there has been an error. If the error is logged in such a way that the entries in the log, and the action of logging, are visible only to those who have adequate clearance, then no violation of policy occurs. But if the clearance of the user is sufficiently high, then the user can see the error without a violation of policy. Thus, the error can be logged for the system administrator (or other appropriate user), even if it cannot be displayed to the user who is running the program. Similar comments apply to any exception action, such as abnormal termination.

15.3.2 Variable Classes

The classes of the variables in the examples above are fixed. Fenton's machine alters the class of the PC as the program runs. This suggests a notion of dynamic classes, wherein a variable can change its class. For explicit assignments, the change is straightforward. When the assignment

```
y := f(x1, …, xn)
```

occurs, y's class is changed to $lub(\underline{x}_1, \ldots, \underline{x}_n)$. Again, implicit flows complicate matters.

EXAMPLE: Consider the following program (which is the same as the program in the example for the Data Mark Machine).[3]

[3] From Denning [242], Figure 5.5, p. 285.

```
proc copy(x : integer class { x };
                var y : integer class { y });
var z : integer class variable { Low };
begin
      y := 0;
      z := 0;
      if x = 0 then z := 1;
      if z = 0 then y := 1;
end;
```

In this program, $\underline{z}$ is variable and initially *Low*. It changes when something is assigned to z. Flows are certified whenever anything is assigned to y. Suppose $\underline{y} < \underline{x}$.

If $x = 0$ initially, the first statement checks that $Low \leq \underline{y}$ (trivially true). The second statement sets z to 0 and $\underline{z}$ to *Low*. The third statement changes z to 1 and $\underline{z}$ to $lub(Low, \underline{x}) = \underline{x}$. The fourth statement is skipped (because $z = 1$). Hence, y is set to 0 on exit.

If $x = 1$ initially, the first statement checks that $Low \leq \underline{y}$ (trivially true). The second statement sets z to 0 and $\underline{z}$ to *Low*. The third statement is skipped (because $x = 1$). The fourth statement assigns 1 to y and checks that $lub(Low, \underline{z}) = Low \leq \underline{y}$ (again, trivially true). Hence, y is set to 1 on exit.

Information has therefore flowed from x to y even though $\underline{y} < \underline{x}$. The program violates the policy but is nevertheless certified.

Fenton's Data Mark Machine would detect the violation (see Exercise 4).

Denning [239] suggests an alternative approach. She raises the class of the targets of assignments in the conditionals and verifies the information flow requirements, even when the branch is not taken. Her method would raise $\underline{z}$ to $\underline{x}$ in the third statement (even when the conditional is false). The certification check at the fourth statement then would fail, because $lub(Low, \underline{z}) = \underline{x} \leq \underline{y}$ is false.

Denning ([242], p. 285) credits Lampson with another mechanism. Lampson suggested changing classes only when explicit flows occur. But all flows force certification checks. For example, when $x = 0$, the third statement sets $\underline{z}$ to *Low* and then verifies $\underline{x} \leq \underline{z}$ (which is true if and only if $\underline{x} = Low$).

15.4 Example Information Flow Controls

Like the program-based information flow mechanisms discussed above, both special-purpose and general-purpose computer systems have information flow controls at the system level. File access controls, integrity controls, and other types of access controls are mechanisms that attempt to inhibit the flow of information within a system, or between systems.

The first example is a special-purpose computer that checks I/O operations between a host and a secondary storage unit. It can be easily adapted to other purposes.

A mail guard for electronic mail moving between a classified network and an unclassified one follows. The goal of both mechanisms is to prevent the illicit flow of information from one system unit to another.

15.4.1 Security Pipeline Interface

Hoffman and Davis [428] propose adding a processor, called a *security pipeline interface* (SPI), between a host and a destination. Data that the host writes to the destination first goes through the SPI, which can analyze the data, alter it, or delete it. But the SPI does not have access to the host's internal memory; it can only operate on the data being output. Furthermore, the host has no control over the SPI. Hoffman and Davis note that SPIs could be linked into a series of SPIs, or be run in parallel.

They suggest that the SPI could check for corrupted programs. A host requests a file from the main disk. An SPI lies on the path between the disk and the host (see Figure 15–2.) Associated with each file is a cryptographic checksum that is stored on a second disk connected to the first SPI. When the file reaches the first SPI, it computes the cryptographic checksum of the file and compares it with the checksum stored on the second disk. If the two match, it assumes that the file is uncorrupted. If not, the SPI requests a clean copy from the second disk, records the corruption in a log, and notifies the user, who can update the main disk.

The information flow being restricted here is an integrity flow, rather than the confidentiality flow of the other examples. The inhibition is not to prevent the corrupt data from being seen, but to prevent the system from trusting it. This emphasizes that, although information flow is usually seen as a mechanism for maintaining confidentiality, its application in maintaining integrity is equally important.

15.4.2 Secure Network Server Mail Guard

Consider two networks, one of which has data classified SECRET[4] and the other of which is a public network. The authorities controlling the SECRET network need to

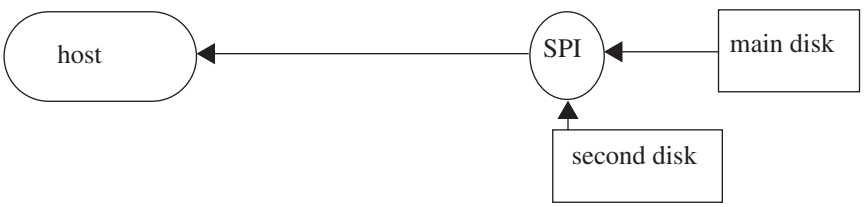

Figure 15–2 Use of an SPI to check for corrupted files.

[4] For this example, assume that the network has only one category, which we omit.

allow electronic mail to go to the unclassified network. They do not want SECRET information to transit the unclassified network, of course. The Secure Network Server Mail Guard (SNSMG) [844] is a computer that sits between the two networks. It analyzes messages and, when needed, sanitizes or blocks them.

The SNSMG accepts messages from either network to be forwarded to the other. It then applies several filters to the message; the specific filters may depend on the source address, destination address, sender, recipient, and/or contents of the message. Examples of the functions of such filters are as follows.

- Check that the sender of a message from the SECRET network is authorized to send messages to the unclassified network.
- Scan any attachments to messages coming from the unclassified network to locate, and eliminate, any computer viruses.
- Require all messages moving from the SECRET to the unclassified network to have a clearance label, and if the label is anything other than UNCLASS (unclassified), encipher the message before forwarding it to the unclassified network.

The SNSMG is a computer that runs two different message transfer agents (MTAs), one for the SECRET network and one for the unclassified network (see Figure 15–3). It uses an assured pipeline [700] to move messages from the MTA to the filter, and vice versa. In this pipeline, messages output from the SECRET network's MTA have type *a*, and messages output from the filters have a different type, type *b*. The unclassified network's MTA will accept as input only messages of type *b*. If a message somehow goes from the SECRET network's MTA to the unclassified network's MTA, the unclassified network's MTA will reject the message as being of the wrong type.

The SNSMG is an information flow enforcement mechanism. It ensures that information cannot flow from a higher security level to a lower one. It can perform other functions, such as restricting the flow of untrusted information from the unclassified network to the trusted, SECRET network. In this sense, the information flow is an integrity issue, not a confidentiality issue.

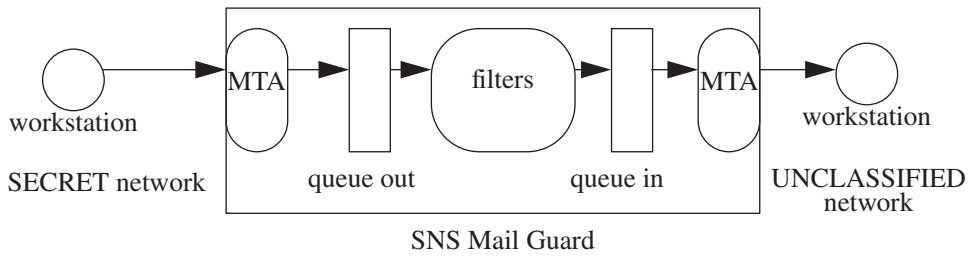

Figure 15–3 Secure Network Server Mail Guard. The SNSMG is processing a message from the SECRET network. The filters are part of a highly trusted system and perform checking and sanitizing of messages.

15.5 Summary

Two aspects of information flow are the amount of information flowing and the way in which it flows. Given the value of one variable, entropy measures the amount of information that one can deduce about a second variable. The flow can be explicit, as in the assignment of the value of one variable to another, or implicit, as in the antecedent of a conditional statement depending on the conditional expression.

Traditionally, models of information flow policies form lattices. Should the models not form lattices, they can be embedded in lattice structures. Hence, analysis of information flow assumes a lattice model.

A compiler-based mechanism assesses the flow of information in a program with respect to a given information flow policy. The mechanism either certifies that the program meets the policy or shows that it fails to meet the policy. It has been shown that if a set of statements meet the information flow policy, their combination (using higher-level language programming constructs) meets the information flow policy.

Execution-based mechanisms check flows at runtime. Unlike compiler-based mechanisms, execution-based mechanisms either allow the flow to occur (if the flow satisfies the information flow policy) or block it (if the flow violates the policy). Classifications of information may be static or dynamic.

Two example information flow control mechanisms, the Security Pipeline Interface and the Secure Network Server Mail Guard, provide information flow controls at the system level rather than at the program and program statement levels.

15.6 Further Reading

The Decentralized Label Model [660] allows one to specify information flow policies on a per-entity basis. Formal models sometimes lead to reports of flows not present in the system; Eckmann [290] discusses these reports, as well as approaches to eliminating them. Guttmann draws lessons from the failure of an information flow analysis technique [385].

Foley [327] presented a model of confinement flow suitable for nonlattice structures, and models nontransitive systems of infoormation flow. Denning [240] describes how to turn a partially ordered set into a lattice, and presents requirements for information flow policies.

The cascade problem is identified in the Trusted Network Interpretation [258]. Numerous studies of this problem describe analyses and approaches [320, 441, 631]; the problem of correcting it with minimum cost is *NP*-complete [440].

Gendler-Fishman and Gudes [351] examine a compile-time flow control mechanism for object-oriented databases. McHugh and Good describe a flow analysis tool [606] for the language Gypsy. Greenwald et al. [379], Kocher [522], Sands

[787], and Shore [826] discuss guards and other mechanisms for control of information flow.

A multithreaded environment adds to the complexity of constraints on information flow [842]. Some architectural characteristics can be used to enforce these constraints [462].

15.7 Exercises

1. Extend the semantics of the information flow security mechanism in Section 15.2.1 for records (structures).
2. Why can we omit the requirement $lub\{ \underline{i}, \underline{b[i]} \} \leq \underline{a[i]}$ from the requirements for secure information flow in the example for iterative statements (see Section 15.2.2.4)?
3. In the flow certification requirement for the *goto* statement in Section 15.2.2.5, the set of blocks along an execution path from b_i to IFD(b_i) excludes these endpoints. Why are they excluded?
4. Prove that Fenton's Data Mark Machine described in Section 15.3.1 would detect the violation of policy in the execution time certification of the *copy* procedure.
5. Discuss how the Security Pipeline Interface in Section 15.4.1 can prevent information flows that violate a confidentiality model. (*Hint:* Think of scanning messages for confidential data and sanitizing or blocking that data.)