Part III

Policy

Security analysts organize the needs of a site in order to define a security policy. From this policy, analysts develop and implement mechanisms for enforcing the policy. The mechanisms may be procedural, technical, or physical. Part III describes the notion of policy and how it can be expressed and formalized, and how different types of policies affect accesses.

Chapter 4, "Security Policies," presents the abstract notion of a security policy and some ways to represent policies. Policy languages abstract some of the common elements of policies and allow expression of policies both at abstract levels and in terms of the properties of the particular systems under consideration.

Chapter 5, "Confidentiality Policies," discusses policies designed primarily for confidentiality. Many government organizations, especially the military, must keep information secret, as described by these policies. Chapter 5 focuses on the Bell-LaPadula security policy.

Chapter 6, "Integrity Policies," discusses policies designed primarily for integrity. Banks, insurance companies, and other commercial and industrial firms worry more about data and programs being corrupted than about them being read, and use these policies.

Chapter 7, "Availability Policies," considers policies that govern the ability to access resources, and the quality of service that defines "access." With the growth of the Internet and the sharing of systems on a wide scale, especially as epitomized by the idea of "cloud computing," the ability to access resources as intended is critical for computing.

Chapter 8, "Hybrid Policies," presents policies that are hybrids of confidentiality and integrity security policies. One comes from the world of stock brokerage, and another from

medical systems. Other types of policy models discussed here are originator controlled models and role-based models.

Chapter 9, "Noninterference and Policy Composition," discusses the noninterference and nondeducibility models of security policies and the composition of security policies in general.

Chapter 4
Security Policies

PORTIA: Of a strange nature is the suit you follow;
Yet in such rule that the Venetian law
Cannot impugn you as you do proceed.
[*To Antonio*.] You stand within his danger, do you not?
— *The Merchant of Venice*, IV, i, 177–180.

A security policy defines "secure" for a system or a set of systems. Security policies can be informal or highly mathematical in nature. After defining a security policy precisely, we expand on the nature of "trust" and its relationship to security policies. We also discuss different types of policy models.

4.1 The Nature of Security Policies

Consider a computer system to be a finite-state automaton with a set of transition functions that change state. Then:

> **Definition 4–1.** A *security policy* is a statement that partitions the states of the system into a set of *authorized*, or *secure*, states and a set of *unauthorized*, or *nonsecure*, states.

A security policy sets the context in which we can define a secure system. What is secure under one policy may not be secure under a different policy. More precisely:

> **Definition 4–2.** A *secure system* is a system that starts in an authorized state and cannot enter an unauthorized state.

Consider the finite-state machine in Figure 4–1. It consists of four states and five transitions. The security policy partitions the states into a set of authorized states $A = \{s_1, s_2\}$ and a set of unauthorized states $UA = \{s_3, s_4\}$. This system

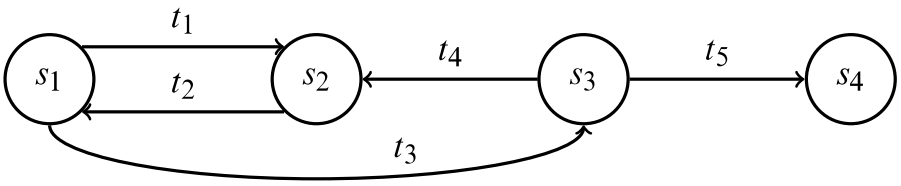

Figure 4–1 A simple finite-state machine.

is not secure, because regardless of which authorized state it starts in, it can enter an unauthorized state. However, if the edge from s_1 to s_3 were not present, the system would be secure, because it could not enter an unauthorized state from an authorized state.

> **Definition 4–3.** A *breach of security* occurs when a system enters an unauthorized state.

We informally discussed the three basic properties relevant to security in Section 1.1. We now define them precisely.

> **Definition 4–4.** Let X be a set of entities and let I be some information. Then I has the property of *confidentiality* with respect to X if no member of X can obtain information about I.

Confidentiality implies that information must not be disclosed to some set of entities. It may be disclosed to others. The membership of set X is often implicit—for example, when we speak of a document that is confidential. Some entity has access to the document. All entities not authorized to have such access make up the set X.

> **Definition 4–5.** Let X be a set of entities and let I be some information or a resource. Then I has the property of *integrity* with respect to X if all members of X trust I.

This definition is deceptively simple. In addition to trusting the information itself, the members of X also trust that the conveyance and storage of I do not change the information or its trustworthiness (this aspect is sometimes called *data integrity*). If I is information about the origin of something, or about an identity, the members of X trust that the information is correct and unchanged (this aspect is sometimes called *origin integrity* or, more commonly, *authentication*). Also, I may be a resource rather than information. In that case, integrity means that the resource functions correctly (meeting its specifications). This aspect is called *assurance* and will be discussed in Part VI, "Assurance." As with confidentiality, the membership of X is often implicit.

Definition 4–6. Let X be a set of entities and let I be a resource. Then I has the property of *availability* with respect to X if all members of X can access I.

The exact definition of "access" in Definition 4–6 varies depending on the needs of the members of X, the nature of the resource, and the use to which the resource is put. If a book-selling server takes up to 1 hour to service a request to purchase a book, that may meet the client's requirements for "availability." If a server of medical information takes up to 1 hour to service a request for information regarding an allergy to an anesthetic, that will not meet an emergency room's requirements for "availability."

A security policy considers all relevant aspects of confidentiality, integrity, and availability. With respect to confidentiality, it identifies those states in which information leaks to those not authorized to receive it. This includes the leakage of rights and the illicit transmission of information without leakage of rights, called *information flow*. Also, the policy must handle changes of authorization, so it includes a temporal element. For example, a contractor working for a company may be authorized to access proprietary information during the lifetime of a nondisclosure agreement, but when that nondisclosure agreement expires, the contractor can no longer access that information. This aspect of the security policy is often called a *confidentiality policy*.

With respect to integrity, a security policy identifies authorized ways in which information may be altered and entities authorized to alter it. Authorization may derive from a variety of relationships, and external influences may constrain it; for example, in many transactions, a principle called *separation of duties* forbids an entity from completing the transaction on its own. Those parts of the security policy that describe the conditions and manner in which data can be altered are called the *integrity policy*.

With respect to availability, a security policy describes what services must be provided. It may present parameters within which the services will be accessible—for example, that a browser may download web pages but not Java applets. It may require a level of service—for example, that a server will provide authentication data within 1 minute of the request being made. This relates directly to issues of quality of service.

The statement of a security policy may formally state the desired properties of the system. If the system is to be provably secure, the formal statement will allow the designers and implementers to prove that those desired properties hold. If a formal proof is unnecessary or infeasible, analysts can test that the desired properties hold for some set of inputs. Later chapters will discuss both these topics in detail.

In practice, a less formal type of security policy defines the set of authorized states. Typically, the security policy assumes that the reader understands the context in which the policy is issued—in particular, the laws, organizational policies, and other environmental factors. The security policy then describes conduct, actions, and authorizations defining "authorized users" and "authorized use."

EXAMPLE: A university disallows cheating, which is defined to include copying another student's homework assignment (with or without permission). A computer science class requires the students to do their homework on the department's computer. One student notices that a second student has not read-protected the file containing her homework and copies it. Has either student (or have both students) breached security?

The second student has not, despite her failure to protect her homework. The security policy requires no action to prevent files from being read. Although she may have been too trusting, the policy does not ban this; hence, the second student has not breached security.

The first student has breached security. The security policy disallows the copying of homework, and the student has done exactly that. Whether the security policy specifically states that "files containing homework shall not be copied" or simply says that "users are bound by the rules of the university" is irrelevant; in the latter case, one of those rules bans cheating. If the security policy is silent on such matters, the most reasonable interpretation is that the policy disallows actions that the university disallows, because the computer science department is part of the university.

The retort that the first user could copy the files, and therefore the action is allowed, confuses *mechanism* with *policy*. The distinction is sharp:

> **Definition 4–7.** A *security mechanism* is an entity or procedure that enforces some part of the security policy.

EXAMPLE: In the preceding example, the policy is the statement that no student may copy another student's homework. One mechanism is the file access controls; if the second student had set permissions to prevent the first student from reading the file containing her homework, the first student could not have copied that file.

EXAMPLE: Another site's security policy states that information relating to a particular product is proprietary and is not to leave the control of the company. The company stores its backup tapes in a vault in the town's bank (this is common practice in case the computer installation is completely destroyed). The company must ensure that only authorized employees have access to the backup tapes even when the tapes are stored off-site; hence, the bank's controls on access to the vault, and the procedures used to transport the tapes to and from the bank, are considered security mechanisms. Note that these mechanisms are not technical controls built into the computer. Procedural, or operational, controls also can be security mechanisms.

Security policies are often implicit rather than explicit. This causes confusion, especially when the policy is defined in terms of the mechanisms. This definition may be ambiguous—for example, if some mechanisms prevent a specific

action and others allow it. Such policies lead to confusion, and sites should avoid them.

EXAMPLE: The UNIX operating system, initially developed for a small research group, had mechanisms sufficient to prevent users from accidentally damaging one another's files; for example, the user *ken* could not delete the user *dmr*'s files (unless *dmr* had set the files and the containing directories to allow this). The implied security policy for this friendly environment was "do not delete or corrupt another's files, and any file not protected may be read."

When the UNIX operating system moved into academic institutions and commercial and government environments, the previous security policy became inadequate; for example, some files had to be protected from individual users (rather than from groups of users). Not surprisingly, the security mechanisms were inadequate for those environments.

The difference between a policy and an abstract description of that policy is crucial to the analysis that follows.

Definition 4–8. A *policy model* is a model that represents a particular policy or class of policies.

A model abstracts details relevant for analysis. Analyses rarely discuss particular policies; they usually focus on *specific characteristics* of policies, because many policies exhibit these characteristics, and the more policies with those characteristics, the more useful the analysis. By the HRU result (see Theorem 3.2), no single nontrivial analysis can cover all policies, but restricting the class of security policies sufficiently allows meaningful analysis of that class of policies.

4.2 Types of Security Policies

Each site has its own requirements for the levels of confidentiality, integrity, and availability, and the site policy states these needs for that particular site.

Definition 4–9. A *military security policy* (also called a *governmental security policy*) is a security policy developed primarily to provide confidentiality.

The name comes from the military's need to keep some information secret, such as the date that a troop ship will sail. Although integrity and availability are important, organizations using this class of policies can overcome the loss of either—for example, by using orders not sent through a computer network. But the compromise of confidentiality would be catastrophic, because an opponent would be able to plan countermeasures (and the organization may not know of the compromise).

Confidentiality is one of the factors of privacy, an issue recognized in the laws of many government entities (such as the Privacy Act of the United States [1702, 2205] and similar legislation of the European Union [2198, 2199]). Aside from constraining what information a government entity can legally obtain from individuals, such acts place constraints on the disclosure and use of that information. Unauthorized disclosure can result in penalties that include jail or fines; also, such disclosure undermines the authority and respect that individuals have for the government and inhibits them from disclosing that type of information to the agencies so compromised.

Definition 4–10. A *commercial security policy* is a security policy developed primarily to provide integrity.

The name comes from the need of commercial firms to prevent tampering with their data, because they could not survive such compromises. For example, if the confidentiality of a bank's computer is compromised, a customer's account balance may be revealed. This would certainly embarrass the bank and possibly cause the customer to take her business elsewhere. But the loss to the bank's "bottom line" would be minor. However, if the integrity of the computer holding the accounts were compromised, the balances in the customers' accounts could be altered, with financially ruinous effects.

Some integrity policies use the notion of a transaction. Like database specifications, they require that actions occur in such a way as to leave the database in a consistent state. These policies, called *transaction-oriented integrity security policies*, are critical to organizations that require consistency of databases.

EXAMPLE: When a customer moves money from one account to another, the bank uses a well-formed transaction. This transaction has two distinct parts: money is first debited to the original account and then credited to the second account. Unless both parts of the transaction are completed, the customer will lose the money. With a well-formed transaction, if the transaction is interrupted, the state of the database is still consistent—either as it was before the transaction began or as it would have been when the transaction ended. Hence, part of the bank's security policy is that all transactions must be well-formed.

The role of trust in these policies highlights their difference. Confidentiality policies place no trust in objects; so far as the policy is concerned, the object could be a factually correct report or a tale taken from *Aesop's Fables*. The policy statement dictates whether that object can be disclosed. It says nothing about whether the object should be believed.

Integrity policies, to the contrary, indicate how much the object can be trusted. Given that this level of trust is correct, the policy dictates what a subject can do with that object. But the crucial question is how the level of trust is assigned. For example, if a site obtains a new version of a program, should that program have high integrity (that is, the site trusts the new version of that program) or low integrity (that is, the site does not yet trust the new program), or

should the level of trust be somewhere in between (because the vendor supplied the program, but it has not been tested at the local site as thoroughly as the old version)? This makes integrity policies considerably more nebulous than confidentiality policies. The assignment of a level of confidentiality is based on what the classifier wants others to know, but the assignment of a level of integrity is based on what the classifier subjectively believes to be true about the trustworthiness of the information.

Two other terms describe policies related to security needs. Because they appear elsewhere, we define them now.

Definition 4–11. A *confidentiality policy* is a security policy dealing only with confidentiality.

Definition 4–12. An *integrity policy* is a security policy dealing only with integrity.

Both confidentiality policies and military policies deal with confidentiality. However, a confidentiality policy does not deal with integrity at all, whereas a military policy may. A similar distinction holds for integrity policies and commercial policies.

4.3 The Role of Trust

The role of trust is crucial to understanding the nature of computer security. This book presents theories and mechanisms for analyzing and enhancing computer security, but any theories or mechanisms rest on certain assumptions. When someone understands the assumptions her security policies, mechanisms, and procedures rest on, she will have a very good understanding of how effective those policies, mechanisms, and procedures are. Let us examine the consequences of this maxim.

A system administrator receives a security patch for her computer's operating system. She installs it. Has she improved the security of her system? She has indeed, given the correctness of certain assumptions:

- She is assuming that the patch came from the vendor and was not tampered with in transit, rather than from an attacker trying to trick her into installing a bogus patch that would actually open security holes. Winkler [2011] describes a penetration test in which this technique enabled attackers to gain direct access to the computer systems of the target.
- She is assuming that the vendor tested the patch thoroughly. Vendors are often under considerable pressure to issue patches quickly and sometimes test them only against a particular attack. The vulnerability may be deeper, however, and other attacks may succeed. When someone

released an exploit of one vendor's operating system code, the vendor released a correcting patch in 24 hours. Unfortunately, the patch opened a second hole, one that was far easier to exploit. The next patch (released 48 hours later) fixed both problems correctly.

- She is assuming that the vendor's test environment corresponds to her environment. Otherwise, the patch may not work as expected. As an example, a vendor's patch once enabled the host's personal firewall, causing it to block incoming connections by default. This prevented many programs from functioning. The host had to be reconfigured to allow the programs to continue to function [2253]. This assumption also covers possible conflicts between different patches, such as patches from different vendors of software that the system is using.
- She is assuming that the patch is installed correctly. Some patches are simple to install, because they are simply executable files. Others are complex, requiring the system administrator to reconfigure network-oriented properties, add a user, modify the contents of a registry, give rights to some set of users, and then reboot the system. An error in any of these steps could prevent the patch from correcting the problems, as could an inconsistency between the environments in which the patch was developed and in which the patch is applied. Furthermore, the patch may claim to require specific privileges, when in reality the privileges are unnecessary and in fact dangerous.

These assumptions are fairly high-level, but invalidating any of them makes the patch a potential security problem.

Assumptions arise also at a much lower level. Consider formal verification (see Chapter 21), an oft-touted panacea for security problems. The important aspect is that formal verification provides a formal mathematical proof that a given program P is correct—that is, given any set of inputs i, j, k, the program P will produce the output x that its specification requires. This level of assurance is greater than most existing programs provide, and hence makes P a desirable program. Suppose a security-related program S has been formally verified for the operating system O. What assumptions would be made when it was installed?

- The formal verification of S is correct—that is, the proof has no errors. Because formal verification relies on automated theorem provers or other formal methods as well as human analysis, the theorem provers must be programmed correctly.
- The assumptions made in the formal verification of S are correct; specifically, the preconditions hold in the environment in which the program is to be executed. These preconditions are typically fed to the theorem provers as well as the program S. An implicit aspect of this assumption is that the version of O in the environment in which the program is to be executed is the same as the version of O used to verify S.

- The program will be transformed into an executable whose actions correspond to those indicated by the source code; in other words, the compiler, linker, loader, and any libraries are correct. An experiment with one version of the UNIX operating system demonstrated how devastating a rigged compiler could be [1875]. Some attack tools replace libraries with others that perform additional functions, thereby increasing security risks [307, 353, 418].
- The hardware will execute the program as intended. A program that relies on floating-point calculations would yield incorrect results on some computer CPU chips, regardless of any formal verification of the program, owing to a flaw in these chips [431]. Similarly, a program that relies on inputs from hardware assumes that specific conditions cause those inputs.

The point is that *any* security policy, mechanism, or procedure is based on assumptions that, if incorrect, destroy the superstructure on which it is built. Analysts and designers (and users) must bear this in mind, because unless they understand what the security policy, mechanism, or procedure is based on, they jump from an unwarranted assumption to an erroneous conclusion.

4.4 Types of Access Control

A security policy may use two types of access controls, alone or in combination. In one, access control is left to the discretion of the owner. In the other, the operating system controls access, and the owner cannot override the controls.

The first type is based on user identity and is the most widely known.

Definition 4–13. If an individual user can set an access control mechanism to allow or deny access to an object, that mechanism is a *discretionary access control* (DAC), also called an *identity-based access control* (IBAC).

Discretionary access controls base access rights on the identity of the subject and the identity of the object involved. Identity is the key; the owner of the object constrains who can access it by allowing only particular subjects to have access. The owner states the constraint in terms of the identity of the subject, or the owner of the object.

EXAMPLE: Suppose a child keeps a diary. The child controls access to the diary, because she can allow someone to read it (grant read access) or not allow someone to read it (deny read access). The child allows her mother to read it, but no one else. This is a discretionary access control because access to the diary is based on the identity of the subject (mom) requesting read access to the object (the diary).

The second type of access control is based on fiat, and identity is irrelevant:

Definition 4–14. When a system mechanism controls access to an object and an individual user cannot alter that access, the control is a *mandatory access control* (MAC), occasionally called a *rule-based access control*.

The operating system enforces mandatory access controls. Neither the subject nor the owner of the object can determine whether access is granted. Typically, the system mechanism will check attributes associated with both the subject and the object to determine whether the subject should be allowed to access the object. Rules describe the conditions under which access is allowed.

EXAMPLE: The law allows a court to access driving records without an owner's permission. This is a mandatory control, because the owner of the record has no control over the court's access to the information.

Definition 4–15. An *originator controlled access control* (ORCON or ORGCON) bases access on the creator of an object (or the information it contains).

The goal of this control is to allow the originator of the file (or of the information it contains) to control the dissemination of the information. The owner of the file has no control over who may access the file. Section 8.3 discusses this type of control in detail.

EXAMPLE: Bit Twiddlers, Inc., a company famous for its embedded systems, contracts with Microhackers Ltd., a company equally famous for its microcoding abilities. The contract requires Microhackers to develop a new microcode language for a particular processor designed to be used in high-performance embedded systems. Bit Twiddlers gives Microhackers a copy of its specifications for the processor. The terms of the contract require Microhackers to obtain permission before it gives any information about the processor to its subcontractors. This is an originator controlled access mechanism because, even though Microhackers owns the file containing the specifications, it may not allow anyone to access that information unless the creator of that information, Bit Twiddlers, gives permission.

4.5 Policy Languages

A *policy language* is a language for representing a security policy. High-level policy languages express policy constraints on entities using abstractions. Low-level policy languages express constraints in terms of input or invocation options to programs existing on the systems.

Chapter 5
Confidentiality Policies

SHEPHERD: Sir, there lies such secrets in this fardel
and box which none must know but the king;
and which he shall know within this hour, if I
may come to the speech of him.
— *The Winter's Tale*, IV, iv, 785–788.

Confidentiality policies emphasize the protection of confidentiality. The importance of these policies lies in part in what they provide, and in part in their role in the development of the concept of security. This chapter explores one such policy—the Bell-LaPadula Model—and the controversy it engendered.

5.1 Goals of Confidentiality Policies

A confidentiality policy, also called an *information flow policy*, prevents the unauthorized disclosure of information. Unauthorized alteration of information is secondary. For example, the navy must keep confidential the date on which a troop ship will sail. If the date is changed, the redundancy in the systems and paperwork should catch that change. But if the enemy knows the date of sailing, the ship could be sunk. Because of extensive redundancy in military communications channels, availability is also less of a problem.

The term "governmental" covers several requirements that protect citizens' privacy. In the United States, the Privacy Act requires that certain personal data be kept confidential. Income tax returns are legally confidential and are available only to the Internal Revenue Service or to legal authorities with a court order. The principle of "executive privilege" and the system of nonmilitary classifications suggest that the people working in the government need to limit the distribution of certain documents and information. Governmental models represent the policies that satisfy these requirements.

5.2 The Bell-LaPadula Model

The Bell-LaPadula Model [149, 150] corresponds to military-style classifications. It has influenced the development of many other models and indeed much of the development of computer security technologies.

5.2.1 Informal Description

The simplest type of confidentiality classification is a set of *security clearances* arranged in a linear (total) ordering (see Figure 5–1). These clearances represent sensitivity levels. The higher the security clearance, the more sensitive the information (and the greater the need to keep it confidential). A subject has a *security clearance*. In the figure, Claire's security clearance is C (for CONFIDENTIAL), and Thomas's is TS (for TOP SECRET). An object has a *security classification*; the security classification of the electronic mail files is S (for SECRET), and that of the telephone list files is UC (for UNCLASSIFIED). (When we refer to both subject clearances and object classifications, we use the term "classification.") The goal of the Bell-LaPadula security model is to prevent information flowing from objects at a security classification higher than a subject's clearance to that subject.

The Bell-LaPadula security model combines mandatory and discretionary access controls. In what follows, "S has discretionary read (or write) access to O" means that the access control matrix entry for S and O corresponding to the discretionary access control component contains a read (or write) right. In other words, were the mandatory controls not present, S would be able to read (or write) O.

Let $L(S) = l_s$ be the security clearance of subject S, and let $L(O) = l_o$ be the security classification of object O. For all security classifications l_i, $i = 0, ..., k-1$, $l_i < l_{i+1}$:

Simple Security Condition, Preliminary Version: S can read O if and only if $l_o \leq l_s$ and S has discretionary read access to O.

TOP SECRET (TS)	Tamara, Thomas	Personnel Files
SECRET (S)	Sally, Samuel	Electronic Mail Files
CONFIDENTIAL (C)	Claire, Clarence	Activity Log Files
UNCLASSIFIED (UC)	Ulaley, Ursula	Telephone List Files

Figure 5–1 At the left is the basic confidentiality classification system. The four security levels are arranged with the most sensitive at the top and the least sensitive at the bottom. In the middle are individuals grouped by their security clearances, and at the right is a set of documents grouped by their security levels.

In Figure 5–1, for example, Claire and Clarence cannot read personnel files, but Tamara and Sally can read the activity log files (and, in fact, Tamara can read any of the files, given her clearance), assuming that the discretionary access controls allow it.

Should Tamara decide to copy the contents of the personnel files into the activity log files and set the discretionary access permissions appropriately, Claire could then read the personnel files. Thus, for all practical purposes, Claire could read the files at a higher level of security. A second property prevents this:

***-Property (Star Property), Preliminary Version:** S can write O if and only if $l_o \geq l_s$ and S has discretionary write access to O.

Because the activity log files are classified C and Tamara has a clearance of TS, she cannot write to the activity log files.

If both the simple security condition, preliminary version, and the *-property, preliminary version, hold, call the system a *secure system*. A straightforward induction establishes the following theorem.

> **Theorem 5.1.** ***Basic Security Theorem, Preliminary Version:*** Let S be a system with a secure initial state s_0, and let T be a set of state transformations. If every element of T preserves the simple security condition, preliminary version, and the *-property, preliminary version, then every state $s_i, i \geq 0$, is secure.

Expand the model by adding a set of categories to each security classification. Each category describes a kind of information. Objects placed in multiple categories have the kinds of information in all of those categories. These categories arise from the "need to know" principle, which states that no subject should be able to read objects unless reading them is necessary for that subject to perform its functions. The sets of categories to which a person may have access is simply the power set of the set of categories. For example, if the categories are NUC, EUR, and US, someone can have access to any of the following sets of categories: $\varnothing$ (none), { NUC }, { EUR }, { US }, { NUC, EUR }, { NUC, US }, { EUR, US }, and { NUC, EUR, US }. These sets of categories form a lattice under the operation $\subseteq$ (subset of); see Figure 5–2. (Appendix A, "Lattices," discusses the mathematical nature of lattices.)

Each security clearance or classification and category forms a *security level*.[1] As before, we say that subjects *have clearance at* (or *are cleared into*, or *are in*) a security level and that objects are at the level of (or are in) a security level. For example, William may be cleared into the level (SECRET, { EUR }) and George

[1]This terminology is not fully agreed upon. Some call security levels "compartments." However, other use this term as a synonym for "categories." We follow the terminology of the unified exposition [150].

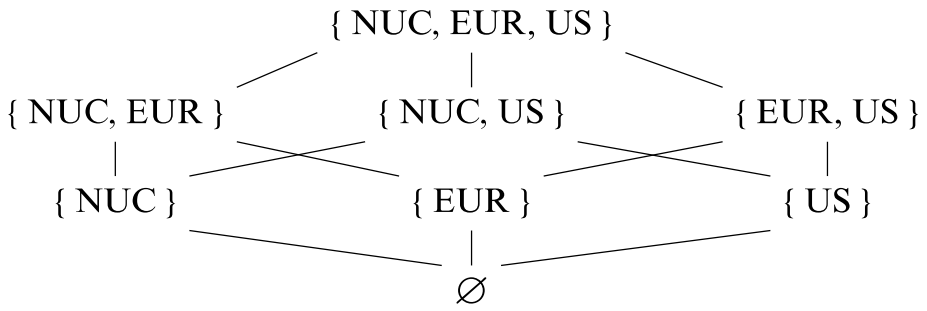

Figure 5–2 Lattice generated by the categories NUC, EUR, and US. The lines represent the ordering relation induced by $\subseteq$.

into the level (TOP SECRET, { NUC, US }). A document may be classified as (CONFIDENTIAL, { EUR }).

Security levels change access. Because categories are based on a "need to know," someone with access to the category set { NUC, US } presumably has no need to access items in the category EUR. Hence, read access should be denied, even if the security clearance of the subject is higher than the security classification of the object. But if the desired object is in any security level with category sets $\varnothing$, { NUC }, { US }, or { NUC, US } and the subject's security clearance is no less than the document's security classification, access should be granted because the subject is cleared into the same category set as the object. This suggests a new relation for capturing the combination of security classification and category set. Define the relation *dom* (dominates) as follows.

> **Definition 5–1.** The security level (L, C) dominates the security level (L', C'), written $(L, C)\ dom(L', C')$, if and only if $L' \leq L$ and $C' \subseteq C$.

We write $(L, C)\ \neg dom(L', C')$ when $(L, C)\ dom(L', C')$ is false. This relation also induces a lattice on the set of security levels [534].

EXAMPLE: George is cleared into security level (SECRET, { NUC, EUR }), DocA is classified as (CONFIDENTIAL, { NUC }), DocB is classified as (SECRET, { EUR, US }), and DocC is classified as (SECRET, { EUR }). Then:

- George *dom* DocA as CONFIDENTIAL $\leq$ SECRET and { NUC } $\subseteq$ { NUC, EUR }
- George $\neg$*dom* DocB as { EUR, US } $\not\subseteq$ { NUC, EUR }
- George *dom* DocC as SECRET $\leq$ SECRET and { EUR } $\subseteq$ { NUC, EUR }

Let $C(S)$ be the category set of subject S, and let $C(O)$ be the category set of object O. The simple security condition, preliminary version, is modified in the obvious way:

Simple Security Condition: S can read O if and only if $S\,dom\,O$ and S has discretionary read access to O.

In the previous example, George can read DocA and DocC but not DocB (again, assuming that the discretionary access controls allow such access).

Suppose Paul is cleared into security level (SECRET, { EUR, US, NUC }) and has discretionary read access to DocB. Paul can read DocB; were he to copy its contents to DocA and set its access permissions accordingly, George could then read DocB. The modified *-property prevents this:

***-Property:** S can write to O if and only if $O\,dom\,S$ and S has discretionary write access to O.

DocA *dom* Paul is false (because $C(Paul) \not\subseteq C(DocA)$), so Paul cannot write to DocA.

The simple security condition is often described as "no reads up" and the *-property as "no writes down."

Redefine a *secure system* to be a system in which both the simple security property and the *-property hold. The analogue to the Basic Security Theorem, preliminary version, can also be established by induction.

> **Theorem 5.2.** ***Basic Security Theorem:*** Let S be a system with a secure initial state s_0, and let T be a set of state transformations. If every element of T preserves the simple security condition and the *-property, then every state $s_i, i \geq 0$, is secure.

At times, a subject must communicate with another subject at a lower level. This requires the higher-level subject to write into a lower-level object that the lower-level subject can read.

EXAMPLE: A colonel with (SECRET, { NUC, EUR }) clearance needs to send a message to a major with (SECRET, { EUR }) clearance. The colonel must write a document that has at most the (SECRET, { EUR }) classification. But this violates the *-property, because (SECRET, { NUC, EUR }) *dom* (SECRET, { EUR }).

The model provides a mechanism for allowing this type of communication. A subject has a *maximum security level* and a *current security level*. The maximum security level must dominate the current security level. A subject may (effectively) decrease its security level from the maximum in order to communicate with entities at lower security levels.

EXAMPLE: The colonel's maximum security level is (SECRET, { NUC, EUR }). She changes her current security level to (SECRET, { EUR }). This is valid, because the maximum security level dominates the current security level. She can then create the document at the major's clearance level and send it to him.

How this policy is instantiated in different environments depends on the requirements of each environment. The conventional use is to define "read" as "allowing information to flow from the object being read to the subject reading," and "write" as "allowing information to flow from the subject writing to the object being written." Thus, "read" usually includes "execute" (because by monitoring the instructions executed, one can determine the contents of portions of the file) and "write" includes "append" (as the information is placed in the file, it does not overwrite what is already in the file, however). Other actions may be included as appropriate; however, those who instantiate the model must understand exactly what those actions are. Chapter 9, "Noninterference and Policy Composition," and Chapter 18, "Confinement Problem," will discuss this subject in considerably more detail.

5.2.2 Example: Trusted Solaris

Trusted Solaris [2140, 2241, 2244–2246] is based on a noninterference policy model (see Chapter 9), which appears as a Bell-LaPadula model for user and process interactions. The mandatory access control policy is based on labels of subjects and objects.

Labels consist of a classification and a set of categories. The security administrator defines the set of valid labels. A *sensitivity label* of a subject and an object is used for mandatory access control. A *clearance* is the least upper bound of all the sensitivity labels of a subject; the clearance need not be a valid label, though. All system objects that are available to users have the distinguished label ADMIN_LOW, which any other label dominates; the privileged administrative objects such as logs and configuration files have the distinguished label ADMIN_HIGH, which dominates any other label.

Each subject S has a controlling user U_S. In addition to a clearance and a sensitivity label S_L, a subject also has an attribute $privileged(S, P)$ indicating whether S can override or bypass part of a security policy P, and another attribute $asserted(S, P)$ indicating whether S is currently asserting $privileged(S, P)$.

The model defines six rules. In the following, a named object O has sensitivity label O_L, and C_L is the clearance of subject S. The policy elements involved are P_1, which is "change S_L," P_2, which is "change O_L," P_3, which is "override O's mandatory read access contol," and P_4, which is "override O's mandatory write access control."

- If $\neg privileged(S, P_1)$, then no sequence of operations can change S_L to a value that it has not previously assumed.
- If $\neg privileged(S, P_1)$, then $asserted(S, P_1)$ is always false.

- If $\neg privileged(S, \text{change } S_L)$, then no value of S_L can be outside the clearance of U_S.
- For all subjects S and named objects O, if $\neg privileged(S, P_2)$, then no sequence of operations can change O_L to a value that it has not previously assumed.
- For all subjects S and named objects O, if $\neg privileged(S, P_3)$, then write access to O is granted only if $S_L \; dom \; O_L$. This is the instantiation of the Bell-LaPadula simple security condition.
- For all subjects S and named objects O, if $\neg privileged(S, P_4)$, then read access to O is granted only if $O_L \; dom \; S_L$ and $C_L \; dom \; O_L$. This is the instantiation of the Bell-LaPadula *-property.

When a user logs into a Trusted Solaris system, the system determines whether the user's session is to be a single-level session. Each account is assigned a label range; the upper bound of the range is the user's clearance and the lower bound is the user's minimum label. If the two are the same, the user is given a single level session with that label. If not, the user is asked whether the session is to be a single-level or multilevel session, and if the latter the user can specify the session clearance (which must not be dominated by the user's minimum label, and must be dominated by the user's clearance). During a multilevel session, the user can change to any label in the range from the user's minimum label to the session clearance. This is helpful when a single user will define several workspaces, each with its own sensitivity level. Of course, the session clearance must dominate the sensitivity labels of all workspaces used during that session.

Unlike in the Bell-LaPadula model, writing is allowed only when the labels of the subject and the object are equal, or when the file is in a special *downgraded directory* that the administrator can create. In that case, the following must all be true for a subject S with sensitivity label S_L and clearance C_L to write to a file O with sensitivity label O_L, which is in a directory D with sensitivity label D_L:

- $S_L \; dom \; D_L$;
- S has discretionary read and search access to D;
- $O_L \; dom \; S_L$ and $O_L \neq S_L$;
- S has discretionary write access to O; and
- $C_L \; dom \; O_L$.

Note that the subject cannot read the object.

EXAMPLE: A process has clearance C_L and label S_L, a file has sensitivity label F_L, and the file is in a directory with sensitivity label D_L. The process has discretionary access to read and write the file and to read and search the directory. If $S_L \; dom \; F_L$, then the process can read the file. If $S_L = F_L$, then the process can write the file.

with a declassification policy, can be measured. This view treats declassification as exceptions to the security policy (although authorized ones). A government agency that never declassifies any information might be considered secure; should it declassify information, though, there is the potential for information that should remain confidential to be declassified by accident—making the system nonsecure. Thus, in some sense, declassification creates a potential "hole" in the system security policy, and weakens the system.

Tranquility plays an important role in the Bell-LaPadula Model, because it highlights the trust assumptions in the model. It raises other problems in the context of integrity that we will revisit in the next chapter.

5.4 The Controversy over the Bell-LaPadula Model

The Bell-LaPadula Model became the target of inquiries into the foundations of computer security. The controversy led to a reexamination of security models and a deeper appreciation of the complexity of modeling real systems.

5.4.1 McLean's †-Property and the Basic Security Theorem

In a 1985 paper [1296], McLean argued that the "value of the [Basic Security Theorem] is much overrated since there is a great deal more to security than it captures. Further, what is captured by the [Basic Security Theorem] is so trivial that it is hard to imagine a realistic security model for which it does not hold" [1296, p. 67]. The basis for McLean's argument was that, given assumptions known to be nonsecure, the Basic Security Theorem could prove a nonsecure system to be secure. He defined a complement to the *-property:

> **Definition 5–11.** A state (b, m, f, h) satisfies the *†-property* if and only if, for each $s \in S$, the following hold:
>
> (a) $b(s : \underline{a}) \neq \varnothing \Rightarrow [\forall o \in b(s : \underline{a})[f_c(o)\, dom\, f_o(s)]]$
> (b) $b(s : \underline{w}) \neq \varnothing \Rightarrow [\forall o \in b(s : \underline{w})[f_c(o) = f_o(s)]]$
> (c) $b(s : \underline{r}) \neq \varnothing \Rightarrow [\forall o \in b(s : \underline{r})[f_c(o)\, dom\, f_c(c)]]$

In other words, the †-property holds for a subject s and an object o if, whenever s has $\underline{w}$ rights over o, the clearance of s dominates the classification of o. This is exactly the reverse of the *-property, which holds that the classification of o would dominate the clearance of s. A state satisfies the †-property if and only if, for every triplet (s, o, p), where the right p involves writing (that is, $p = \underline{a}$ or $p = \underline{w}$), the †-property holds for s and o.

McLean then proved the analogue to Theorem 5.4:

Theorem 5.16. $\Sigma(R, D, W, z_0)$ satisfies the †-property relative to $S' \subseteq S$ for any secure state z_0 if and only if, for every action $(r, d, (b, m, f, h), (b', m', f', h'))$, W satisfies the following for every $s \in S'$:

(a) Every $(s, o, p) \in b - b'$ satisfies the †-property with respect to S'.
(b) Every $(s, o, p) \in b'$ that does not satisfy the †-property with respect to S' is not in b.

Proof See Exercise 5, with "*-property" replaced by "†-property."

From this theorem, and from Theorems 5.3 and 5.5, the analogue to the Basic Security Theorem follows.

Theorem 5.17. ***McLean's Basic Security Theorem:*** $\Sigma(R, D, W, z_0)$ is a secure system if and only if z_0 is a secure state and W satisfies the conditions of Theorems 5.3, 5.16, and 5.5.

However, the system $\Sigma(R, D, W, z_0)$ is clearly nonsecure, because a subject with HIGH clearance can write information to an object with LOW classification. Information can flow down, from HIGH to LOW. This violates the basic notion of security in the confidentiality policy.

Consider the role of the Basic Security Theorem in the Bell-LaPadula Model. The goal of the model is to demonstrate that specific rules, such as the *get-read* rule, preserve security. But what is security? The model defines that term using the Basic Security Theorem: an instantiation of the model is secure if and only if the initial state satisfies the simple security condition, the *-property, and the ds-property, and the transition rules preserve those properties. In essence, the theorems are assertions about the three properties.

The rules describe the changes in a *particular* system instantiating the model. Showing that the system is secure, as defined by the analogue of Definition 5–3, requires proving that the rules preserve the three properties. Given that they do, McLean's Basic Security Theorem asserts that reachable states of the system will also satisfy the three properties. The system will remain secure, given that it starts in a secure state.

LaPadula pointed out that McLean's statement does not reflect the assumptions of the Basic Security Theorem [1138]. Specifically, the Bell-LaPadula Model assumes that a transition rule introduces no changes that violate security, but does *not* assume that any *existing* accesses that violate security are eliminated. The rules instantiating the model do no elimination (see the *get-read* rule, Section 5.2.4.1, as an example).

Furthermore, the *nature* of the rules is irrelevant to the model. The model accepts a definition of "secure" as axiomatic. The specific *policy* defines "security" and is an instantiation of the model. The Bell-LaPadula Model uses a military

definition of security: information may not flow from a dominating entity to a dominated entity. The *-property captures this requirement. But McLean's variant uses a different definition: rather than meet the *-property, his policy requires that information not flow from a dominated entity to a dominating entity. This is not a confidentiality policy. Hence, a system satisfying McLean's policy will not satisfy a confidentiality policy.

However, the sets of properties in both policies (the confidentiality policy and McLean's variant) are inductive, and both Basic Security Theorems hold. The properties may not make sense in a real system, but this is irrelevant to the model. It is very relevant to the *interpretation* of the model, however. The confidentiality policy requires that information not flow from a dominating subject to a dominated object. McLean substitutes a policy that allows this. These are alternative instantiations of the model.

McLean makes these points by stating problems that are central to the use of any security model. The model must abstract the notion of security that the system is to support. For example, McLean's variant of the confidentiality policy does not provide a correct definition of security for military purposes. An analyst examining a system could not use this variant to show that the system implemented a confidentiality classification scheme. The Basic Security Theorem, and indeed all theorems, fail to capture this, because the definition of "security" is axiomatic. The analyst must establish an appropriate definition. All the Basic Security Theorem requires is that the definition of security be inductive.

McLean's second observation asks whether an analyst can prove that the system being modeled meets the definition of "security." Again, this is beyond the province of the model. The model makes claims based on hypotheses. The issue is whether the hypotheses hold for a real system.

5.4.2 McLean's System Z and More Questions

In a second paper [1297], McLean sharpened his critique. System transitions can alter any system component, including b, f, m, and h, as long as the new state does not violate security. McLean used this property to demonstrate a system, called System Z, that satisfies the model but is not a confidentiality security policy. From this, he concluded that the Bell-LaPadula Model is inadequate for modeling systems with confidentiality security policies.

System Z has the weak tranquility property and supports exactly one action. When a subject requests any type of access to any object, the system downgrades all subjects and objects to the lowest security level, adds access permission to the access control matrix, and allows the access.

Let System Z's initial state satisfy the simple security condition, the *-property, and the ds-property. It can be shown that successive states of System Z also satisfy those properties and hence System Z meets the requirements of the Basic Security Theorem. However, with respect to the confidentiality security policy requirements, the system clearly is not secure, because all entities are downgraded.

McLean reformulated the notion of a secure action. He defined an alternative version of the simple security condition, the *-property, and the ds-property. Intuitively, an action satisfies these properties if, given a state that satisfies the properties, the action transforms the system into a (possibly different) state that satisfies these properties, and *eliminates any accesses present in the transformed state that would violate the property in the initial state*. From this, he shows:

> **Theorem 5.18.** $\Sigma(R, D, W, z_0)$ is a secure system if z_0 is a secure state and each action in W satisfies the alternative versions of the simple security condition, the *-property, and the ds-property.
>
> **Proof** See [1297].

Under this reformulation, System Z is not secure because this rule is not secure. Specifically, consider an instantiation of System Z with two security clearances, (HIGH, { ALL }) and (LOW, { ALL }) (LOW < HIGH). The initial state has a subject s and an object o. Take $f_c(s)$ = (LOW, { ALL }), $f_o(o)$ = (HIGH, { ALL }), $m[s, o]$ = { $\underline{\text{w}}$ }, and b = { $(s, o, \underline{\text{w}})$ }. When s requests read access to o, the rule transforms the system into a state wherein $f_o'(o)$ = (LOW, { ALL }), $(s, o, \underline{\text{r}}) \in b'$, and $m'[s, o]$ = { $\underline{\text{r}}, \underline{\text{w}}$ }. However, because $(s, o, \underline{\text{r}}) \in b' - b$ and $f_o(o)\, dom\, f_s(s)$, an illegal access has been added. Yet, under the traditional Bell-LaPadula formulation, in the final state $f_c'(s) = f_o'(o)$, so the read access is legal and the state is secure, hence the system is secure.

McLean's conclusion is that proving that states are secure is insufficient to prove the security of a system. One must consider both states and transitions.

Bell [148] responded by exploring the fundamental nature of modeling. Modeling in the physical sciences abstracts a physical phenomenon to its fundamental properties. For example, Newtonian mathematics coupled with Kepler's laws of planetary motion provide an abstract description of how planets move. When observers noted that Uranus did not follow those laws, they calculated the existence of another, trans-Uranean planet. Adams and Lavoisier, observing independently, confirmed its existence. Refinements arise when the theories cannot adequately account for observed phenomena. For example, the precession of Mercury's orbit suggested another planet between Mercury and the sun. But none was found.[7] Einstein's theory of general relativity, which modified the theory of how planets move, explained the precession, and observations confirmed his theory.

Modeling in the foundations of mathematics begins with a set of axioms. The model demonstrates the consistency of the axioms. A model consisting of points, lines, planes, and the axioms of Euclidean geometry can demonstrate the

[7] Observers reported seeing this planet, called Vulcan, in the mid-1800s. The sighting was never officially confirmed, and the refinements discussed above explained the precession adequately. Willy Ley's book [1160] relates the charming history of this episode.

Chapter 6
Integrity Policies

ISABELLA: Some one with child by him? My cousin Juliet?
LUCIO: Is she your cousin?
ISABELLA: Adoptedly; as school-maids change their names
By vain, though apt affection.
— *Measure for Measure*, I, iv, 45–48.

An inventory control system may function correctly if the data it manages is released; but it cannot function correctly if the data can be randomly changed. So integrity, rather than confidentiality, is key. These policies are important because many commercial and industrial firms are more concerned with accuracy than disclosure. This chapter discusses the major integrity security policies and explores their design.

6.1 Goals

Commercial requirements differ from military requirements in their emphasis on preserving data integrity. Lipner [1193] identifies five commercial requirements:

1. Users will not write their own programs, but will use existing production programs and databases.
2. Programmers will develop and test programs on a nonproduction system; if they need access to actual data, they will be given production data via a special process, but will use it on their development system.
3. A special process must be followed to install a program from the development system onto the production system.
4. The special process in requirement 3 must be controlled and audited.
5. The managers and auditors must have access to both the system state and the system logs that are generated.

These requirements suggest several principles of operation.

First comes *separation of duty*. The principle of separation of duty states that if two or more steps are required to perform a critical function, at least two different people should perform the steps. Moving a program from the development system to the production system is an example of a critical function. Suppose one of the application programmers made an invalid assumption while developing the program. Part of the installation procedure is for the installer to certify that the program works "correctly," that is, as required. The error is more likely to be caught if the installer is a different person (or set of people) than the developer. Similarly, if the developer wishes to subvert the production data with a corrupt program, the certifier either must not detect the code to do the corruption, or must be in league with the developer.

Next comes *separation of function*. Developers do not develop new programs on production systems because of the potential threat to production data. Similarly, the developers do not process production data on the development systems. Depending on the sensitivity of the data, the developers and testers may receive sanitized production data. Further, the development environment must be as similar as possible to the actual production environment.

Last comes *auditing*. Commercial systems emphasize recovery and accountability. Auditing is the process of analyzing systems to determine what actions took place and who performed them. Hence, commercial systems must allow extensive auditing and thus have extensive logging (the basis for most auditing). Logging and auditing are especially important when programs move from the development system to the production system, since the integrity mechanisms typically do not constrain the certifier. Auditing is, in many senses, external to the model.

Even when disclosure is at issue, the needs of a commercial environment differ from those of a military environment. In a military environment, clearance to access specific categories and security levels brings the ability to access information in those compartments. Commercial firms rarely grant access on the basis of "clearance"; if a particular individual needs to know specific information, he or she will be given it. While this can be modeled using the Bell-LaPadula Model, it requires a large number of categories and security levels, increasing the complexity of the modeling. More difficult is the issue of controlling this proliferation of categories and security levels. In a military environment, creation of security levels and categories is centralized. In commercial firms, this creation would usually be decentralized. The former allows tight control on the number of compartments, whereas the latter allows no such control.

More insidious is the problem of information aggregation. Commercial firms usually allow a limited amount of (innocuous) information to become public, but keep a large amount of (sensitive) information confidential. By aggregating the innocuous information, one can often deduce much sensitive information. Preventing this requires the model to track what questions have been asked, and this complicates the model enormously. Certainly the Bell-LaPadula Model lacks this ability.

6.2 The Biba Model

In 1977, Biba [196] studied the nature of the integrity of systems. He proposed three policies, one of which was the mathematical dual of the Bell-LaPadula Model.

A system consists of a set S of subjects, a set O of objects, and a set I of integrity levels.[1] The levels are ordered. The relation $< \subseteq I \times I$ holds when the second integrity level dominates the first. The relation $\leq \subseteq I \times I$ holds when the second integrity level either dominates or is the same as the first. The function $min : I \times I \rightarrow I$ gives the lesser of the two integrity levels (with respect to $\leq$). The function $i : S \cup O \rightarrow I$ returns the integrity level of an object or a subject. The relation $\underline{r} \subseteq S \times O$ defines the ability of a subject to read an object; the relation $\underline{w} \subseteq S \times O$ defines the ability of a subject to write to an object; and the relation $\underline{x} \subseteq S \times S$ defines the ability of a subject to invoke (execute) another subject.

Some comments on the meaning of "integrity level" will provide intuition behind the constructions to follow. The higher the level, the more confidence one has that a program will execute correctly (or detect problems with its inputs and stop executing). Data at a higher level is more accurate and/or reliable (with respect to some metric) than data at a lower level. Again, this model implicitly incorporates the notion of "trust"; in fact, the term "trustworthiness" is used as a measure of integrity level. For example, a process at a level higher than that of an object is considered more "trustworthy" than that object.

Integrity labels, in general, are not also security labels. They are assigned and maintained separately, because the reasons behind the labels are different. Security labels primarily limit the flow of information; integrity labels primarily inhibit the modification of information. They may overlap, however, with surprising results (see Exercise 3).

Biba tests his policies against the notion of an information transfer path:

> **Definition 6–1.** An *information transfer path* is a sequence of objects $o_1, \ldots, o_{n+1}$ and a corresponding sequence of subjects $s_1, \ldots, s_n$ such that $s_i \ \underline{r} \ o_i$ and $s_i \ \underline{w} \ o_{i+1}$ for all i, $1 \leq i \leq n$.

Intuitively, data in the object o_1 can be transferred into the object o_{n+1} along an information flow path by a succession of reads and writes.

[1] The original model did not include categories and compartments. The changes required to add them are straightforward.

6.2.1 Low-Water-Mark Policy

Whenever a subject accesses an object, the low-water-mark policy [196] changes the integrity level of the subject to the lower of the subject and the object. Specifically:

1. $s \in S$ can write to $o \in O$ if and only if $i(o) \leq i(s)$.
2. If $s \in S$ reads $o \in O$, then $i'(s) = min(i(s), i(o))$, where $i'(s)$ is the subject's integrity level after the read.
3. $s_1 \in S$ can execute $s_2 \in S$ if and only if $i(s_2) \leq i(s_1)$.

The first rule prevents writing from one level to a higher level. This prevents a subject from writing to a more highly trusted object. Intuitively, if a subject were to alter a more trusted object, it could implant incorrect or false data (because the subject is less trusted than the object). In some sense, the trustworthiness of the object would drop to that of the subject. Hence, such writing is disallowed.

The second rule causes a subject's integrity level to drop whenever it reads an object at a lower integrity level. The idea is that the subject is relying on data less trustworthy than itself. Hence, its trustworthiness drops to the lesser trustworthy level. This prevents the data from "contaminating" the subject or its actions.

The third rule allows a subject to execute another subject provided the second is not at a higher integrity level. Otherwise, the less trusted invoker could control the execution of the invoked subject, corrupting it even though it is more trustworthy.

This policy constrains any information transfer path:

Theorem 6.1. [196] If there is an information transfer path from object $o_1 \in O$ to object $o_{n+1} \in O$, then enforcement of the low-water-mark policy requires that $i(o_{n+1}) \leq i(o_1)$ for all $n > 1$.

Proof If an information transfer path exists between o_1 and o_{n+1}, then Definition 6–1 gives a sequence of subjects and objects identifying the entities on the path. Without loss of generality, assume that each read and write was performed in the order of the indices of the vertices. By induction, for any $1 \leq k \leq n$, $i(s_k) = min\{i(o_j) \mid 1 \leq j \leq k\}$ after k reads. As the nth write succeeds, by rule 1, $i(o_{n+1}) \leq i(s_n)$. Thus, by transitivity, $i(o_{n+1}) \leq i(o_1)$.

This policy prevents direct modifications that would lower integrity labels. It also prevents indirect modification by lowering the integrity label of a subject that reads from an object with a lower integrity level.

The problem with this policy is that, in practice, the subjects change integrity levels. In particular, the level of a subject is nonincreasing, which means that it will soon be unable to access objects at a high integrity level. An alternative

policy is to decrease object integrity levels rather than subject integrity levels, but this policy has the property of downgrading object integrity levels to the lowest level.

6.2.2 Ring Policy

The ring policy [196] ignores the issue of indirect modification and focuses on direct modification only. This solves the problems described above. The rules are as follows:

1. Any subject may read any object, regardless of integrity levels.
2. $s \in S$ can write to $o \in O$ if and only if $i(o) \leq i(s)$.
3. $s_1 \in S$ can execute $s_2 \in S$ if and only if $i(s_2) \leq i(s_1)$.

The difference between this policy and the low-water-mark policy is simply that any subject can read any object.

6.2.3 Biba's Model (Strict Integrity Policy)

The strict integrity policy model [196] is the dual of the Bell-LaPadula Model, and is most commonly called "Biba's model." Its rules are as follows:

1. $s \in S$ can read $o \in O$ if and only if $i(s) \leq i(o)$.
2. $s \in S$ can write to $o \in O$ if and only if $i(o) \leq i(s)$.
3. $s_1 \in S$ can execute $s_2 \in S$ if and only if $i(s_2) \leq i(s_1)$.

Given these rules, Theorem 6.1 still holds, but its proof changes (see Exercise 1). Note that rules 1 and 2 imply that if both read and write are allowed, $i(s) = i(o)$.

Like the low-water-mark policy, this policy prevents indirect as well as direct modification of entities without authorization. By replacing the notion of "integrity level" with "integrity compartments," and adding the notion of discretionary controls, one obtains the full dual of Bell-LaPadula. Indeed, the rules require "no reads down" and "no writes up"—the exact opposite of the simple security condition and the *-property of the Bell-LaPadula Model.

EXAMPLE: Pozzo and Gray [1543,1544] implemented Biba's strict integrity model on the distributed operating system LOCUS [1533]. Their goal was to limit execution domains for each program to prevent untrusted software from altering data or other software. Their approach was to make the level of trust in software and data explicit. They have different classes of executable programs. Their *credibility ratings* (Biba's integrity levels) assign a measure of trustworthiness

on a scale from 0 (untrusted) to *n* (highly trusted), depending on the source of the software. Trusted file systems contain only executable files with the same credibility level. Associated with each user (process) is a risk level that starts out set to the highest credibility level at which that user can execute. Users may execute programs with credibility levels at least as great as the user's risk level. To execute programs at a lower credibility level, a user must use the *run-untrusted* command. This acknowledges the risk that the user is taking.

EXAMPLE: The FreeBSD system's implementation of the Biba model [2185] uses integers for both parts of an integrity label. The integrity level consists of a *grade*, the values of which are linearly ordered, and a *compartment*, the set of which is not ordered. A grade is represented by an integer value between 0 and 65,535 inclusive, with higher grades having higher numbers. A category is represented by an integer value between 0 and 255 inclusive. The labels are written as "biba/100:29+64+130," meaning the label has integrity grade 100 and is in integrity categories 29, 64, and 130. FreeBSD defines three distinguished labels: "biba/low," which is the lowest label; "biba/high," which is the highest label; and "biba/equal," which is equal to all labels.

Objects have a single label. Like the Trusted Solaris implementation of the Bell-LaPadula model, subjects have three. The first is the label at which the subject is currently; the other two represent the low and high labels of a range. These are written as biba/*currentlabel*(*lowlabel-highlabel*). The subject can change its current label to any label within that range. So, a subject with label biba/75:29+64(50:29-150:29+64+130+150) can read but not write to an object with label biba/100:29+64+130 as the object's label dominates the subject's label. If the subject changes its label to biba/100:29+64+130, it will be able to read from and write to the object.

6.3 Lipner's Integrity Matrix Model

Lipner returned to the Bell-LaPadula Model and combined it with the Biba model to create a model [1193] that conformed more accurately to the requirements of a commercial policy. For clarity, we consider the Bell-LaPadula aspects of Lipner's model first, and then combine those aspects with Biba's model.

6.3.1 Lipner's Use of the Bell-LaPadula Model

Lipner provides two security levels, in the following order (higher to lower):

- Audit Manager (AM): system audit and management functions are at this level.
- System Low (SL): any process can read information at this level.

fundamentally, the Bell-LaPadula Model restricts the flow of information. Lipner notes this, suggesting that combining his model with Biba's may be the most effective.

6.4 Clark-Wilson Integrity Model

In 1987, David Clark and David Wilson developed an integrity model [423] radically different from previous models. This model uses transactions as the basic operation, which models many commercial systems more realistically than previous models.

One main concern of a commercial environment, as discussed above, is the integrity of the data in the system and of the actions performed on that data. The data is said to be *in a consistent state* (or *consistent*) if it satisfies given properties. For example, let D be the amount of money deposited so far today, W the amount of money withdrawn so far today, YB the amount of money in all accounts at the end of yesterday, and TB the amount of money in all accounts so far today. Then the consistency property is

$$D + YB - W = TB$$

Before and after each action, the consistency conditions must hold. A *well-formed transaction* is a series of operations that transition the system from one consistent state to another consistent state. For example, if a depositor transfers money from one account to another, the transaction is the transfer; two operations, the deduction from the first account and the addition to the second account, make up this transaction. Each operation may leave the data in an inconsistent state, but the well-formed transaction must preserve consistency.

The second feature of a commercial environment relevant to an integrity policy is the integrity of the transactions themselves. Who examines and certifies that the transactions are performed correctly? For example, when a company receives an invoice, the purchasing office requires several steps to pay for it. First, someone must have requested a service, and determined the account that would pay for the service. Next, someone must validate the invoice (was the service being billed for actually performed?). The account authorized to pay for the service must be debited, and the check must be written and signed. If one person performs all these steps, that person could easily pay phony invoices; however, if at least two different people perform these steps, both must conspire to defraud the company. Requiring more than one person to handle this process is an example of the principle of separation of duty.

Computer-based transactions are no different. Someone must certify that the transactions are implemented correctly. The principle of separation of duty requires that the certifier and the implementors be different people. In order for the transaction to corrupt the data (either by illicitly changing the data or by leaving

the data in an inconsistent state), two different people must either make similar mistakes or collude to certify the well-formed transaction as correct.

6.4.1 The Model

The Clark-Wilson model defines data constrained by its integrity controls as *constrained data items*, or CDIs. Data not subject to the integrity controls are called *unconstrained data items*, or UDIs. For example, in a bank, the balances of accounts are CDIs since their integrity is crucial to the operation of the bank, whereas the gifts selected by the account holders when their accounts were opened would be UDIs, because their integrity is not crucial to the operation of the bank. The set of CDIs and the set of UDIs partition the set of all data in the system being modeled.

A set of *integrity constraints* (similar in spirit to the consistency constraints discussed above) constrain the values of the CDIs. In the bank example, the consistency constraint presented earlier would also be an integrity constraint.

The model also defines two sets of procedures. *Integrity verification procedures*, or IVPs, test that the CDIs conform to the integrity constraints at the time the IVPs are run. In this case, the system is said to be in a *valid state*. *Transformation procedures*, or TPs, change the state of the data in the system from one valid state to another; TPs implement well-formed transactions.

Return to the example of bank accounts. The balances in the accounts are CDIs; checking that the accounts are balanced, as described above, is an IVP. Depositing money, withdrawing money, and transferring money between accounts are TPs. To ensure that the accounts are managed correctly, a bank examiner must certify that the bank is using proper procedures to check that the accounts are balanced, to deposit money, to withdraw money, and to transfer money. Furthermore, those procedures may apply only to deposit and checking accounts; they might not apply to other types of accounts, such as petty cash. The Clark-Wilson model captures these requirements in two *certification rules*:

Certification rule 1 (CR1): When any IVP is run, it must ensure that all CDIs are in a valid state.

Certification rule 2 (CR2): For some associated set of CDIs, a TP must transform those CDIs in a valid state into a (possibly different) valid state.

CR2 defines as *certified* a relation that associates a set of CDIs with a particular TP. Let C be the certified relation. Then, in the bank example,

$$(\text{balance}, \text{account}_1), (\text{balance}, \text{account}_2), \ldots, (\text{balance}, \text{account}_n) \in C$$

CR2 implies that a TP may corrupt a CDI if it is not certified to work on that CDI. For example, the TP that invests money in the bank's stock portfolio would corrupt account balances even if the TP were certified to work on the portfolio,

because the actions of the TP make no sense on the bank accounts. Hence, the system must prevent TPs from operating on CDIs for which they have not been certified. This leads to the following *enforcement rule*:

Enforcement rule 1 (CR1): The system must maintain the *certified* relations, and must ensure that only TPs certified to run on a CDI manipulate that CDI.

Specifically, ER1 says that if a TP f operates on a CDI o, then $(f, o) \in C$. However, in a bank, a janitor is not allowed to balance customer accounts. This restriction implies that the model must account for the person performing the TP, or user. The Clark-Wilson model uses an enforcement rule for this:

Enforcement rule 2 (CR2): The system must associate a user with each TP and set of CDIs. The TP may access those CDIs on behalf of the associated user. If the user is not associated with a particular TP and CDI, then the TP cannot access that CDI on behalf of that user.

This defines a set of triples $(user, TP, \{CDIset\})$ to capture the association of users, TPs, and CDIs. Call this relation *allowed* A. Of course, these relations must be certified:

Certification rule 3 (CR3): The *allowed* relations must meet the requirements imposed by the principle of separation of duty.

Because the model represents users, it must ensure that the identification of a user with the system's corresponding user identification code is correct. This suggests:

Enforcement rule 3 (CR3): The system must authenticate each user attempting to execute a TP.

An interesting observation is that the model does not require authentication when a user logs into the system, because the user may manipulate only UDIs. But if the user tries to manipulate a CDI, the user can do so only through a TP; this requires the user to be certified as allowed (per ER2), which requires authentication of the user (per ER3).

Most transaction-based systems log each transaction so that an auditor can review the transactions. The Clark-Wilson model considers the log simply as a CDI, and every TP appends to the log; no TP can overwrite the log. This leads to:

Certification rule 4 (CR4): All TPs must append enough information to reconstruct the operation to an append-only CDI.

When information enters a system, it need not be trusted or constrained. For example, when one deposits money into an automated teller machine (ATM),

one need not enter the correct amount. However, when the ATM is opened and the cash or checks counted, the bank personnel will detect the discrepancy and fix it before they enter the deposit amount into one's account. This is an example of a UDI (the stated deposit amount) being checked, fixed if necessary, and certified as correct before being transformed into a CDI (the deposit amount added to one's account). The Clark-Wilson model covers this situation with certification rule 5:

Certification rule 5 (CR5): Any TP that takes as input a UDI may perform only valid transformations, or no transformations, for all possible values of the UDI. The transformation either rejects the UDI or transforms it into a CDI.

The final rule enforces the separation of duty needed to maintain the integrity of the relations in rules ER2 and ER3. If a user could create a TP and associate some set of entities and herself with that TP (as in ER3), she could have the TP perform unauthorized acts that violate integrity constraints. The final enforcement rule prevents this:

Enforcement rule 4 (CR4): Only the certifier of a TP may change the list of entities associated with that TP. No certifier of a TP, or of an entity associated with that TP, may ever have execute permission with respect to that entity.

This rule requires that all possible values of the UDI be known, and that the TP be implemented so as to be able to handle them. This issue arises again in both vulnerabilities analysis and secure programming.

This model contributed two new ideas to integrity models. First, it captured the way most commercial firms work with data. The firms do not classify data using a multilevel scheme, and they enforce separation of duty. Second, the notion of certification is distinct from the notion of enforcement, and each has its own set of rules. Assuming correct design and implementation, a system with a policy following the Clark-Wilson model will ensure that the enforcement rules are obeyed. But the certification rules require outside intervention, and the process of certification is typically complex and prone to error or to incompleteness (because the certifiers make assumptions about what can be trusted). This is a weakness in some sense, but it makes explicit assumptions that other models do not.

6.4.1.1 A UNIX Approximation to Clark-Wilson

Polk describes an implementation of Clark-Wilson under the UNIX operating system [1529]. He first defines "phantom" users that correspond to locked accounts. No real user may assume the identity of a phantom user.

Now consider the triple (*user*, *TP*, {*CDIset*}). For each TP, define a phantom user to be the owner. Place that phantom user into the group that owns each of the CDIs in the CDI set. Place all real users authorized to execute the TP on the CDIs in the CDI set into the group owner of the TP. The TPs are setuid to the

TP owner,[2] and are executable by the group owner. The CDIs are owned either by *root* or by a phantom user.

EXAMPLE: Suppose access to each CDI is constrained by user only—that is, in the triple, *TP* can be any TP. In this case, the CDI is owned by a group containing all users who can modify the CDI.

EXAMPLE: Now, suppose access to each CDI is constrained by TP only—that is, in the triple, *user* can be any user. In this case, the CDIs allow access to the owner, a phantom user *u*. Then each TP allowed to access the CDI is owned by *u*, setuid to *u*, and world-executable.

Polk points out three problems. Two different users cannot use the same TP to access two different CDIs. This requires two separate copies of the TP, one for each user and associated CDI. Secondly, this greatly increases the number of setuid programs, which increases the threat of improperly granted privileges. Proper design and assignment to groups minimizes this problem. Finally, the superuser can assume the identity of any phantom user. Without radically changing the nature of the *root* account, this problem cannot be overcome.

6.4.2 Comparison with the Requirements

We now consider whether the Clark-Wilson model meets the five requirements in Section 6.1. We assume that production programs correspond to TPs and that production data (and databases) are CDIs.

1. If users are not allowed to perform certifications of TPs, but instead only "trusted personnel" are, then CR5 and ER4 enforce this requirement. Because ordinary users cannot create certified TPs, they cannot write programs to access production databases. They must use existing TPs and CDIs—that is, production programs and production databases.
2. This requirement is largely procedural, because no set of technical controls can prevent a programmer from developing and testing programs on production systems. (The standard procedural control is to omit interpreters and compilers from production systems.) However, the notion of providing production data via a special process corresponds to using a TP to sanitize, or simply provide, production data to a test system.
3. Installing a program from a development system onto a production system requires a TP to do the installation and "trusted personnel" to do the certification.

[2] That is, the TPs execute with the rights of the TP owner, and not of the user executing the TP.

4. CR4 provides the auditing (logging) of program installation. ER3 authenticates the "trusted personnel" doing the installation. CR5 and ER4 control the installation procedure (the new program being a UDI before certification and a CDI, as well as a TP in the context of other rules, after certification).
5. Finally, because the log is simply a CDI, management and auditors can have access to the system logs through appropriate TPs. Similarly, they also have access to the system state.

Thus, the Clark-Wilson model meets Lipner's requirements.

6.4.3 Comparison with Other Models

The contributions of the Clark-Wilson model are many. We compare it with the Biba model to highlight these new features.

Recall that the Biba model attaches integrity levels to objects and subjects. In the broadest sense, so does the Clark-Wilson model, but unlike the Biba model, each object has two levels: constrained or high (the CDIs) and unconstrained or low (the UDIs). Similarly, subjects have two levels: certified (the TPs) and uncertified (all other procedures). Given this similarity, can the Clark-Wilson model be expressed fully using the Biba model?

The critical distinction between the two models lies in the certification rules. The Biba model has none; it asserts that "trusted" subjects exist to ensure that the actions of a system obey the rules of the model. No mechanism or procedure is provided to verify the trusted entities or their actions. But the Clark-Wilson model provides explicit requirements that entities and actions must meet; in other words, the method of upgrading an entity is itself a TP that a security officer has certified. This underlies the assumptions being made and allows for the upgrading of entities within the constructs of the model (see ER4 and CR5). As with the Bell-LaPadula Model, if the Biba model does not have tranquility, trusted entities must change the objects' integrity levels, and the method of upgrading need not be certified.

Handling changes in integrity levels is critical in systems that receive input from uncontrolled sources. For example, the Biba model requires that a trusted entity, such as a security officer, pass on every input sent to a process running at an integrity level higher than that of the input. This is not practical. However, the Clark-Wilson model requires that a trusted entity (again, perhaps a security officer) certify the method of upgrading data to a higher integrity level. Thus, the trusted entity would not certify each data item being upgraded; it would only need to certify the method for upgrading data, and the data items could be upgraded. This is quite practical.

Can the Clark-Wilson model emulate the Biba model? The relations described in ER2 capture the ability of subjects to act on objects. By choosing TPs appropriately, the emulation succeeds (although the certification rules constrain

trusted subjects in the emulation, whereas the Biba model imposes no such constraints). The details of the construction are left as an exercise for the reader (see Exercise 12).

6.5 Trust Models

Integrity models deal with changes to entities. They state conditions under which the changes preserve those properties that define "integrity." However, they do not deal with the confidence one can have in the initial values or settings of that entity. Put another way, integrity models deal with the *preservation* of trustworthiness, but not with the *initial* evaluation of whether the contents can be trusted.

Trust models, on the other hand, deal with exactly that problem. They provide information about the credibility of data and entities. Because trust is subjective, trust models typically express the trustworthiness of one entity in terms of another. Interestingly, the term "trust" is difficult to define, and much work treats it as axiomatic.

We use the following definition:

> **Definition 6–2.** [734] Anna *trusts* Bernard if Anna believes, with a level of subjective probability, that Bernard will perform a particular action, both before the action can be monitored (or independently of the capacity of being able to monitor it) and in a context in which it affects Anna's own action.

This defines trust in terms of actors, but it also can apply to the credibility of information. Asking whether the data is "trusted" is really asking if a reader of the data believes to some level of subjective probability that the entity providing the data obtained it accurately and without error, and is providing it accurately and without error. Hence, in the above definition, the reader is Anna, the provider is Bernard, and the "particular action" is that of gathering and providing the data.

This definition captures three important points about trust [6]. First, it includes the subjective nature of trust. Second, it captures the idea that trust springs from belief in that which we do not, or cannot, monitor. Third, the actions of those we trust affects our own actions. This also leads to the notion of transitivity of trust.

> **Definition 6–3.** *Transitivity of trust* means that, if a subject Anna trusts a second subject Bernard, and Bernard trusts a third subject Charlene, then Anna trusts Charlene.

In practice, trust is not absolute, so whether trust is transitive depends on Anna's assessment of Bernard's judgment. This leads to the notion of *conditional transitivity of trust* [6], which says that Anna can trust Charlene when:

- Bernard recommends Charlene to Anna;
- Anna trusts Bernard's recommendations;

- Anna can make judgments about Bernard's recommendations; and
- Based on Bernard's recommendation, Anna may trust Charlene less than Bernard does.

If Anna establishes trust in Charlene based on her observations and other interactions, the trust is *direct*. If it is established based on Anna's acceptance of Bernard's recommendation of Charlene, then the trust is *indirect*. Indirect trust may take a path involving many intermediate entities. This is called *trust propagation* because the trust propagates among many entities.

Castelfranchi and Falcone [362] argue that trust is a cognitive property, so only agents with goals and beliefs can trust another agent. This requires the trusting agent, Anna, to estimate risk and then decide, based on her willingness to accept (or not accept) the risk, whether to rely on the one to be trusted, Bernard. This estimation arises from social and technological sources, as well as Anna's observations and her taking into account recommendations. They identify several belief types:

- *Competence belief*: Anna believes Bernard to be competent to aid Anna in reaching her goal.
- *Disposition belief*: Anna believes that Bernard will actually carry out what Anna needs to reach her goal.
- *Dependence belief*: Anna believes she needs what Bernard will do, depends on what Bernard will do, or that it is better for Anna to rely on Bernard than not to rely on him.
- *Fulfillment belief*: Anna believes the goal will be achieved.
- *Willingness belief*: Anna believes that Bernard has decided to take the action she desires.
- *Persistence belief*: Anna believes that Bernard will not change his mind before carrying out the desired action.
- *Self-confidence belief*: Anna believes that Bernard knows that he can take the desired action.

Parsons et al. [1498] provide a set of schemes to evaluate arguments about trust. Trust coming from experience will be based either on Anna's personal experience about Bernard ("direct experience") or on her observation of evidence leading her to conclude Bernard is reliable ("indirect experience"). Trust coming from validation requires that, due to his particular knowledge ("expert opinion"), position ("authority"), or reputation ("reputation"), Bernard be considered an expert in the domain of the goals or actions that Anna wants Bernard to perform to reach that goal. Trust can also come from Anna's observations about Bernard's character ("moral nature") or her belief that Bernard's being untrustworthy would be to Bernard's disadvantage ("social standing"). Finally, Anna can trust Bernard because of factors strictly external to any knowledge of Bernard, for example that most people from Bernard's community are trustworthy ("majority behavior"),

that not trusting Bernard poses an unacceptable risk ("prudence"), or that it best serves Anna's current interests ("pragmatism").

These humanistic traits have analogues in the technological world, but ultimately the trust models frame the technology to provide evidence to support the evaluation of arguments about trust. By using a set of predetermined belief rules, much of the trust analysis can be automated. However, the automation is an attempt to mimic the way the relevant authority, whatever that may be, would evaluate the arguments based on the belief types to determine the appropriate level of trust.

Trust management systems provide a mechanism for instantiating trust models. They use a language to express relationships about trust, often involving assertions or claims about the properties of trust in the model. They also have an evaluation mechanism or engine that takes data and the trust relationships (called the *query*), and provides a measure of the trust in an entity, or determines whether an entity should be trusted or an action taken. The result of the evaluation is rarely complete trust or complete distrust; more often, it is somewhere between.

We distinguish between two basic types of trust models: policy-based models and recommendation-based models.

6.5.1 Policy-Based Trust Management

Policy-based trust models use credentials to instantiate policy rules that determine whether to trust an entity, resource, or information. The credentials themselves are information, so they too may be input to these rules. Trusted third parties often vouch for these credentials. For example, Kerberos (see Section 11.2.2) allows users to verify identity, effectively producing an identity credential. Similarly, certificates (see Section 11.4) encode information about identity and other attributes of the entity, and are often used as credentials. Complicating this is that many agents are automated, particularly web-based agents, and they act on behalf of users to access services and take other actions. In order to do so, they must decide which servers to use and which actions to perform. Thus, trust models generally assume that the agents will act autonomously.

The statement of policies requires a language in which to express those policies. The differing goals of trust models have led to different languages. Some, particularly those intended for the Semantic Web, include negotiation protocols; others simply supply a language to express the rules. The expressiveness of a language determines the range of policies it can express. Usability of the language speaks to the ease of users defining polices in that language, as well as the ability to analyze policies. The languages should also be easily mapped into enforcement mechanisms, so the policies they describe can be enforced.

EXAMPLE: The Keynote trust management system [245] is based on PolicyMaker [246], but is extended to support applications that use public keys. It is designed for simplicity, expressivity, and to be extensible. Its basic units are the assertion and the action environment.

Assertions are either policy assertions or credential assertions. Policy assertions make statements about policy; credential assertions make statements about credentials, and these assertions describe the actions allowed to possessors of the stated credentials. An action environment is a set of attributes that describe an action associated with a set of credentials. An evaluator takes a set of policy assertions describing a local policy, a set of credentials, and an action environment, and determines whether a proposed action is consistent with the local policy by applying the assertions to the action environment.

An assertion is composed of a set of fields listed in Figure 6–5. The "KeyNote-Version" must appear first if it present; similarly, the "Signature" field must come last, if present. If the value of the `Authorizer` field is `"POLICY"`, the assertion is a policy assertion. If that value is a credential, the assertion is a credential assertion. The evaluator returns a result from a set of values called the Compliance Values.

As an example, consider an email domain [244]. The following policy authorizes the holder of credential `mastercred` for all actions:

```
Authorizer: "POLICY"
Licensees: "mastercred"
```

When the evaluator evaluates this policy and the credential assertion

```
KeyNote-Version: 2
Local-Constants: Alice="cred1234", Bob="credABCD"
Authorizer: "authcred"
Licensees: Alice || Bob
Conditions: (app_domain == "RFC822-EMAIL") &&
            (address ~= "^.*@keynote\\.ucdavis\\.edu$")
Signature: "signed"
```

Field	Meaning
Authorizer	Principal making the assertion
Comment	Annotation describing something about the assertion
Conditions	Conditions under which Authorizer trusts Licensee to perform action
KeyNote-Version	Version of KeyNote used for writing assertions
Licensees	Principals authorized by the assertion
Local-Contents	Adds or changes attributes in the current assertion
Signature	Encoded digital signature of Authorizer

Figure 6–5 KeyNote fields and their meanings.

the evaluator enables the entity with the credential identified by "authcred" to trust the holders of either credential "cred1234" or credential "credABCD" to issue credentials for users in the email domain (app_domain == "RFC822-EMAIL") when the address involved ends in "@keynote.ucdavis.edu". So Alice and Bob might be issuers of certificates for members of that domain, and the holder of the credential "authcred" will trust them for those certificates. The evaluator's Compliance Values are { _MIN_TRUST, _MAX_TRUST }. If the action environment is

```
_ACTION_AUTHORIZERS=Alice
app_domain = "RFC822-EMAIL"
address = "snoopy@keynote.ucdavis.edu"
```

then the action satisfies the policy and the evaluator would return _MAX_TRUST (meaning it is trusted because it satisfies the assertions). Conversely, the action environment

```
_ACTION_AUTHORIZERS=Bob
app_domain = "RFC822-EMAIL"
address = "opus@admin.ucdavis.edu"
```

does not satisfy the policy, and the evaluator returns _MIN_TRUST (meaning it is an untrusted action).

As a second example, consider separation of duty for a company's invoicing system. The policy delegates authority for payment of invoices to the entity with credential fundmgrcred:

```
Authorizer: "POLICY"
Licensee: "fundmgecred"
Conditions: (app_domain == "INVOICE" && @dollars < 10000)
```

To implement the separation of duty requirement, the following credential assertion requires at least two signatures on any expenditure:

```
KeyNote-Version: 2
Comment: This credential specifies a spending policy
Authorizer: "authcred"
Licensees: 2-of("cred1", "cred2", "cred3", "cred4",
           "cred5")
Conditions: (app_domain=="INVOICE") # note nested clauses
            -> { (@dollars) < 2500) -> _MAX_TRUST;
                 (@dollars < 7500) -> "ApproveAndLog";
               };
Signature: "signed"
```

This says that the authorizer with credential "`authcred`" (probably a financial officer of the company) allows any two people with any of the five listed credentials to approve payment of invoices under $7,500, but the approval of any invoice for $2,500 or more will be logged. So in this context, the Compliance Value set is { "`Reject`", "`ApproveAndLog`", "`Approve`" }. Thus, if the action environment is

```
_ACTION_AUTHORIZERS = "cred1,cred4"
app_domain = "INVOICE"
dollars = "1000"
```

then the evaluator returns "`Approve`" because it satisfies the policy. This assertion

```
_ACTION_AUTHORIZERS = "cred1,cred2"
app_domain = "INVOICE"
dollars = "3541"
```

causes the evaluator to return "`ApproveandLog`". And these assertions

```
_ACTION_AUTHORIZERS = "cred1"
app_domain = "INVOICE"
dollars = "1500"
```

and

```
_ACTION_AUTHORIZERS = "cred1,cred5"
app_domain = "INVOICE"
dollars = "8000"
```

cause the evaluator to return "`Reject`".

The simplicity of the KeyNote language allows it to be used in a wide variety of environments. This makes KeyNote very powerful, allowing the delegation of trust as needed. By changing the evaluation engine, the assertions may be augmented to express arbitrary conditions. Indeed, KeyNote's predecessor, PolicyMaker, allowed assertions to be arbitrary programs, so it supported both the syntax used in KeyNote and a much more general tool.

Ponder (see Section 4.5.1) can be used to express trust relationships. Rei [986] is a language for expressing trust in a pervasive computing environment, and KAoS [1915] focuses on grid computing and web services. Cassandra [144] is a trust management system tailored for electronic health records.

6.5.2 Reputation-Based Trust Management

Reputation-based models use past behavior, especially during interactions, and information gleaned from other sources to determine whether to trust an entity. This may include recommendations from other entities.

Abdul-Rahman and Hailes [6] base trust on the recommendations of other entities. They distinguish between direct trust relationships (where Amy trusts Boris) and recommender trust relationships (where Amy trusts Boris to make recommendations about another entity). Trust categories refine the nature of the trust; for example, Amy may trust Boris to recommend trustworthy web services, but not to vouch for another entity's identity. Each entity maintains its own list of relationships.

Trust is computed based on the protocol flow through the system. The direct and recommender trust values have specific semantics. For example, Abdul-Rahman and Hailes use for direct trust the values -1 as representing untrustworthy, the integers from 1 to 4 inclusive representing the lowest trust level to completely trustworthy, and 0 as the inability to make trust judgments. For recommender trust values, the integers -1 and 0 represent the same as for direct trust and the meaning of the integers 1 to 4 inclusive represent how close the judgment of the recommender is to the entity being recommended to. Together with an agent's identification and trust category, this forms a *reputation*. Formally, a *recommendation* is trust information that contains one or more reputations.

EXAMPLE: Suppose Amy wants to get Boris's recommendation about Danny, specifically about his ability to write a program, because Amy needs to hire a good programmer. Amy knows Boris and trusts his recommendations with (recommender) trust value 2 (his judgment is somewhat close to hers). She sends Boris a request for a recommendation about Danny's programming abilities. Boris does not know Danny, so he sends a similar request to Carole, who does know Danny. Carole believes Danny is an above average programmer, so she replies to Boris with a recommendation of 3 (more trustworthy than most programmers). Bob adds his own name to the end of the recommendation and forwards it to Amy.

Amy can now compute a trust value for the path used to find out about Danny using the following formula:

$$t(T,P) = tv(T)\prod_{i=1}^{n}\frac{tv(R_i)}{4} = 3 \times \frac{2}{4} \times \frac{3}{4} = 1.125 \tag{6.1}$$

where T is the entity about which information is sought (in this example, Danny), P the path taken (here, the path nodes are Boris and Carole), $tv(x)$ the trust value of x, and t the overall trust in the path.

The metrics or evaluating recommendations are critical; they are also poorly understood. The previous formula, for example, "was derived largely by intuition" [6, p. 57]. The advantage to a simple formula is that it is easily understood; the disadvantage is that the recommendation score is very coarse.

Recommendation systems use many different types of metrics. Statistical models are common, as are belief models (in which the probabilities involved may not sum up to 1, due to uncertainties of belief) and fuzzy models (in which reasoning involves degrees of trustworthiness, rather than being trustworthy or not trustworthy). Previous experience with interactions can also be factored in.

EXAMPLE: The PeerTrust recommendation system [2036] uses a trust metric based on complaints as feedback. Let $u \in P$ be a node in a peer-to-peer network P; let $p(v, t) \in P$ be the node that u interacts with in transaction t. Let $S(u, t)$ be the amount of satisfaction u gets from $p(u, t)$. Let $I(u)$ be the total number of transactions that u performs. Then the trust value of u is computed by the formula

$$T(u) = \sum_{t=1}^{I(u)} S(u, t) Cr(p(u, t))$$

$Cr(v)$ is the credibility of node v's feedback. One of the proposed measures for it is

$$Cr(v) = \sum_{t=1}^{I(v)} S(v, i) \frac{T(p(v, t))}{\sum_{x=1} I(v) T(p(v, x))}$$

The credibility of v therefore depends on its prior trust values.

6.6 Summary

Integrity models are gaining in variety and popularity. The problems they address arise from industries in which environments vary wildly. They take into account concepts (such as separation of privilege) from beyond the scope of confidentiality security policies. This area will continue to increase in importance as more and more commercial firms develop models or policies to help them protect their data.

Although the policy and reputation trust models are presented separately, trust management systems can combine them. The SULTAN system [810] does this using four components. The specification editor and the analysis tool provide policy-based analyses, and the risk and monitoring services feed information about experience back into the system.

Underlying most trust models is some form of logic. Various logics such as fuzzy logics and belief logics incorporate the lack of certainty in general trust and enable reasoning about trust in the framework of that logic. The result is very similar to reputation models, in that one obtains various metrics for trust.

6.7 Research Issues

Central to the maintenance of integrity is an understanding of how trust affects integrity. A logic for analyzing trust in a model or in a system would help analysts understand the role of trust. The problem of constructing such a logic that captures realistic environments is an open question.

The development of realistic integrity models is also an open research question, as are the analysis of a system to derive models and the generation of mechanisms to enforce them. Although these issues arise in all modeling, integrity models are particularly susceptible to failures to capture the underlying processes and entities on which systems are built.

Models for analyzing software and systems to determine whether they conform to desired integrity properties is another critical area, and much of the research on "secure programming" is relevant here. In particular, has the integrity of a piece of software, or of data on which that software relies, been compromised? In the most general form, this question is undecidable; in particular cases, with software that exhibits specific properties, this question is decidable.

The quantification of trust is an open problem. Part of the problem is an understanding of exactly what "trust" means, as it depends not only on environment but also on the psychological, organizational, and sociological forces in the system being modeled. Even when the entities involved are automated, a human or group of humans must provide the judgment on which the trust metrics are based. Perhaps this is why policy-based models are seen as more definitive—the rules of the policy give a straightforward answer, whereas reputation-based trust modeling requires feedback to be effective. But how to evaluate and integrate that feedback to produce results deemed to be accurate is another complex, and open, problem.

6.8 Further Reading

Nash and Poland discuss realistic situations in which mechanisms are unable to enforce the principle of separation of duty [1425]. Other studies of this principle include its use in role-based access control [1111, 1167, 1754], databases [1463], and multilevel security [1463]. Notargiacomo, Blaustein, and McCollum [1461] present a generalization of Clark-Wilson suitable for trusted database management systems that includes dynamic separation of duty. Foley [695] presents two formal definitions of integrity and uses them to reason about separation of duty and other mechanisms.

Integrity requirements arise in many contexts. The SELinux example policy has been analyzed with respect to certain of the Clark-Wilson rules [954]. Saltman [1640] and Neumann [1446] provide an informative survey of the requirements for secure electronic voting, and other papers discuss models that provide integrity in voting [16, 389–391, 1321, 1595]. Chaum's classic paper on electronic payment [386] raises issues of confidentiality and shows that integrity and anonymity can coexist; Bitcoin [1419] used his, and other, ideas to develop a widely used instantiation of digital cash [1420]. Integrity in databases is crucial to their correctness [93, 754, 812, 1966]. The analysis of trust in software is also an issue of integrity [46, 101, 1379].

Chalmers compares commercial policies with governmental ones [372]. Lee [1148] discusses an alternative to Lipner's use of mandatory access controls

Chapter 7
Availability Policies

LADY MACBETH: Alack, I am afraid they have awak'd,
And 'tis not done. Th' attempt and not the deed
Confounds us. Hark! I laid their daggers ready;
He could not miss 'em.
— *Macbeth*, II, ii, 10–13.

Confidentiality and integrity policies describe what can be done once a resource or information is accessed. Availability policies describe when, and for how long, the resource can be accessed. Violations of these policies may occur unintentionally or deliberately.

7.1 Goals of Availability Policies

An availability policy ensures that a resource can be accessed in some way in a timely fashion. This is often expressed in terms of "quality of service." As an example, a commercial website selling merchandise will need to display details of items for customer requests in a matter of seconds or, at worst, a minute. The goal of the customer is to see what the website is selling, and the goal of the site is to make information available to the customer. However, the site does not want customers to alter prices displayed on the website, so there is no availability for altering information. As another example, a website enabling students to upload homework must allow some alterations (students must be able to upload their homework, possibly multiple times per assignment) quickly and no access for the students to read other students' assignments. As these examples show, an availability policy defines the type of access and what a "timely fashion" means. These depend on the nature of the resource and the goals of the accessing entity.

When a resource or service is not available, a denial of service occurs. This problem is closely related to the problems of safety and liveness. A denial of service that results from the service giving incorrect responses means the service is not performing the functions that the client is expecting; this is a safety property.

Similarly, a denial of service that prevents users from accessing the service is a liveness problem. But other problems can cause a denial of service, such as assignment of inadequate resources to a process.

The difference between the mechanisms used to support availability in general, and availability as a security requirement, lies in the assumptions underlying the failures. In the general case, lack of accessibility can be modeled using an average case model, in which this condition occurs following a (known or unknown) statistical model. The failures occur naturally. For example, the failure rates of disk drives depends upon many factors such as the age, the manufacturer, and environment [1526] and can be statistically modeled, although the precise model to be used is unclear [1693]. But the mechanisms used to support availability as a security requirement use a worst-case model, in which an adversary deliberately tries to make the resource or information unavailable. Because attackers induce this condition, models used in computer security describe failures that are nonrandom, and indeed may well be nonstatistical.

7.2 Deadlock

Perhaps the simplest form of availability is that of disallowing deadlocks.

> **Definition 7–1.** A *deadlock* is a state in which some set of processes block, each waiting for another process in the set to take some action.

Deadlock can occur if four conditions hold simultaneously:

1. The resource is not shared (*mutual exclusion*).
2. An entity must hold the resource and block, waiting until another resource becomes available (*hold and wait*).
3. A resource being held cannot be released (*no preemption*).
4. A set of entities must be holding resources such that each entity is waiting for a resource held by another entity in the set (*circular wait*).

There are three approaches to handling deadlock: preventing it, avoiding it, or detecting and recovering from it.

EXAMPLE: Preventing deadlock requires that the system prevent at least one of the above conditions from holding. Early methods required a process to request and obtain all resources necessary to complete its task before starting. An alternative method was to require the process to relinquish all resources when it needed a new one; then it would simply request the new one and all the resources it just released. These invalidated the hold and wait condition, but both have many disadvantages. Linearly ordering resource types breaks the circular wait

condition. In this scheme, each resource type r_i is assigned a number i. In order to acquire resources of type r_a, the process must first release all resources with a lower or equal number. It can reacquire them by adding them to the request for r_a. Mutual exclusion is necessary for some resources, as is not preempting their use.

Dijkstra's Banker's Algorithm is an example of a deadlock avoidance technique. A system can be in either a *safe state*, in which deadlock cannot occur, or an *unsafe state*, which may (but need not) lead to deadlock. Initially, each process states the maximum number and type of resources it will need to complete. A process needing a resource requests it from the resource manager. That manager applies the Banker's Algorithm to determine whether granting the request will place the system in an unsafe state; if so, the process blocks until the request can be satisfied. The algorithm guarantees that all requests will be satisfied in a finite time. Further, the processes must release allocated resources in a finite time. But the term "finite" is indefinite; it could be a very long time, but not infinite. These and other considerations, such as the difficulty of knowing the maximum number and type of resources a process will use, mean that the algorithm is not used in practice.

Deadlock detection techniques allow deadlocks to occur, but detect them and then recover from them. Resource graphs represent processes and resources as nodes, requests as directed edges from a process node to a resource node, and an assignment as a directed edge from a resource node to a process node. When cycles occur, the property of circular wait holds and the processes are deadlocked. Recovery techniques include simply terminating one of the processes, or suspending it and releasing its resources; either of these methods breaks the cycle. Some systems such as distributed databases periodically capture their current state as a checkpoint, and when deadlock occurs they roll back to the most recent checkpoint. Other systems assume deadlocks are infrequent, and rely on the user to detect them and initiate recovery (usually by terminating a process).

Deadlock is a specific example of a situation in which a process is denied service, such as access to a resource. It is usually not due to an attack. A process can acquire multiple resources needed by other processes, and then not release them. In this situation, the process is not blocked (although the others are), so it is not a deadlock.

Denial of service models generalize this problem to include attacks.

7.3 Denial of Service Models

A basic requirement for the use of systems and networks is access. The access must occur in a time frame suitable for the type of access and the use to which the accessed resource, and any associated data, will be put.

Definition 7–2. [772] A *denial of service* occurs when a group of authorized users of a service makes that service unavailable to a (disjoint) group of authorized users for a period of time exceeding a defined maximum waiting time.

The term "authorized user" here must be read expansively. If a user is not authorized, then in theory access control mechanisms that protect the server will block the unauthorized users from accessing the server. But in practice, the access control mechanisms may be ineffective. An intruder may compromise a user's account to gain access to a server. The policy controlling access to a network server may be unworkable, such as one stating that only customers interested in the products sold may access the server—but the access control mechanisms could not tell whether a remote user accessing the server was interested in the products, or trying to block access by others. Hence the first "group of authorized users" is simply the group of users with access to the service, whether the security policy grants them access or not.

Underlying all models and mechanisms is the assumption that, in the absence of other processes, there are sufficient resources to enable a process requesting those resources to proceed. If those resources cannot be allocated to the process, then the security problem is one of inadequate resources, a management problem not dealt with here.

Denial of service models have two essential components. The first is a *waiting time policy*. This controls the time between a request for a resource and the allocation of that resource to the requesting process. A denial of service occurs when the bound set by this policy is exceeded. The environment in which the request is made influences the policy. The acceptable waiting time for a pacemaker to take action affecting a patient's heart beating is considerably different than the acceptable waiting time for a purchase from an Internet website to be acknowledged.

The second is a *user agreement* that establishes constraints a process ("user") must meet in order to ensure service. These are designed to ensure that a process will receive service within the waiting time. For example, a user agreement for parallel processes accessing a mutually exclusive resource would be that, once a process acquires the resource, it must (eventually) release that resource and when released, there are enough unallocated resources to enable a process waiting for those resources to proceed.

When combined, these two components ensure that a process meets the conditions needed to receive the resources it needs and not create a denial of service. It will receive those resources after an acceptable waiting time. Thus, the process can proceed and not itself be denied service.

Two types of models, constraint-based models and state-based models, formalize these notions.

7.3.1 Constraint-Based Model

Some models of denial of service rely on the enforcement of constraints to ensure availability. The Yu-Gligor model of denial of service [2065] has two parts, a

user agreement and a *finite waiting time* policy. The former focuses on undesirable invocation sequences, and the latter on the sharing policies and mechanisms.

7.3.1.1 User Agreement

The goal of the user agreement is to describe the properties that the users of the server must meet. It is not a part of the specification of a service because it involves actions taken that do not involve the particular service. For example, user interaction with multiple services requires an agreement covering those services. An enforcement mechanism is necessary because users may not follow the agreement, and indeed may be unaware of it. The deadlock avoidance technique is an example of this.

> **Definition 7–3.** [2065] A *user agreement* is a set of constraints designed to prevent denial of service.

Let S_{seq} be the set of sequences of all possible invocations of a service, and U_{seq} the set of sequences of all possible invocations by a user. The set $U_{i,seq}$ is the subset of U_{seq} that user U_i can invoke. The use of a service involves commands to consume a resource produced or controlled by that service. This means there are two types of operations. Let C be the set of operations that user U_i performs to consume the service, and let P be the set of operations that produce the resources consumed by user U_i. As resources must be produced before they can be consumed, there is a partial ordering of operations. The partial order relation $p < c$ means that operation p must precede operation c. Call A_i the set of allowed operations for user U_i. Then the set R_i is the set of relations between every pair of allowed operations for that user.

EXAMPLE: Consider the problem of a mutually exclusive resource, in which only one process at a time may use the resource. The operations here are $C = \{acquire\}$ and $P = \{release\}$. Each process may execute both operations; hence, for two processes p_1 and p_2, $A_i = \{acquire_i, release_i\}$ for $i = 1, 2$. Given that a process must acquire the resource before it can release the resource, $R_i = \{(acquire_i < release_i\}$ for $i = 1, 2$.

Next, let $U_i(k)$ represent the initial subsequence of U_i of length k and let $n_o(U_i(k))$ be the number of times operation o occurs in $U_i(k)$. Then $U_i(k)$ is said to be *safe* if the following two conditions hold:

- if $o \in U_{i,seq}$, then $o \in A_i$; and
- for all k, if $(o_1 < o_2) \in R_i$, then $n_{o_1}(U_i(k)) \geq n_{o_2}(U_i(k))$.

This simply says that if operation o is in the sequence of commands that user U_i executes, it is an allowed operation, and that if one operation must precede a second, the number of times the first operation is executed must be at least as great as the number of times the second operation is executed.

monitor allocates the CPU resource to p; call this time t_{CPU}. It does this between each quantum. When the process completes, the resource monitor deallocates the resources given to p; this takes time t_d. Thus, as p satisfies (C1), the time needed to run p and deallocate all resources is $t_a + \lfloor \frac{M}{d} + 1 \rfloor (q + t_{\text{CPU}}) + t_d$. So the maximum time the cycle will take is $(t_a + \lfloor \frac{M}{d} + 1 \rfloor (q + t_{\text{CPU}}) + t_d)n$. Thus, there is a maximum time for each round robin cycle.

These two results will prevent a denial of service.

7.4 Example: Availability and Network Flooding

The SYN flood is the most common type of flooding attack. It is based on the initiation of a connection using the TCP protocol (see Figure 7–1) [615]. The attacker sends the SYN packet to the target, which replies with a SYN/ACK message. But the target fails to receive the ACK packet. As a result, the resources used to hold information about the (pending) connection are held for a period of time before they are released.

The attacker can suppress the sending of the ACK in a number of ways. First, the SYN packet might contain the (spoofed) source address of a nonexistent or nonresponsive host, so the host either never receives the SYN/ACK packet or never responds to it. The attacker could also use the IP address of a system under the attacker's control, so the attacker can block the ACK packet from being sent. This is particularly attractive if the attacker controls a large number of systems such as a botnet (see Section 23.5). If the packets come from multiple sources but have the same destination, the attack is a *distributed denial of service* attack.

In what follows, the term "legitimate handshake" refers to a connection attempt that is not part of a SYN flood. If the client in a legitimate handshake receives the SYN/ACK packet from the server, it will respond with the appropriate ACK to complete the handshake and begin the connection. The term "attack handshake," on the other hand, refers to a connection attempt that is part of a

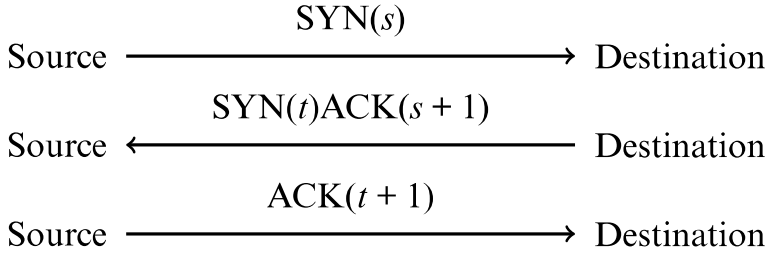

Figure 7–1 The TCP three-way handshake. The SYN packet is a TCP packet with sequence number *s* (or *t*) and the SYN flag set. Likewise, the ACK packet is a TCP packet with acknowledgment number *s + 1* (or *t + 1*) and the ACK flag set. The middle message is a single TCP packet with both SYN and ACK flags set.

SYN flood. The client in an attack handshake will never send an ACK packet to complete the handshake. When the first step in the handshake completes, the server has a "pending connection" and once the handshake completes, the server opens a connection. A critical observation is that the server cannot distinguish between a legitimate handshake and an attack handshake. Both follow the same steps. The only difference lies in whether the third part of the handshake is sent (and received).

7.4.1 Analysis

There are two aspects of SYN flooding. The first is the consumption of bandwidth. If the flooding is more than the capacity of the physical network medium, or of intermediate nodes, legitimate handshakes may be unable to reach the target. The second is the use of resources—specifically, memory space—on the target. If the flooding absorbs all the memory allocated for half-open connections, then the target will discard the SYN packets from legitimate handshake attempts.

Placing the SYN flood attack in the context of the models of denial of service illuminates why the attack works, and how countermeasures compensate for it. The key observation is that the waiting time policy has a maximum wait time, which is the time that the sending process will wait for a SYN/ACK message from the receiver. Specifically, the fairness policy component must assure that a process waiting for the resources will acquire them. But this does not hold; indeed, the point of the attack is to ensure this does not happen. In terms of the Yu-Gligor model, the finite wait time does not hold; in terms of the Millen model, requirement (D2) does not hold. Further, the (implicit) user agreement that traffic from one client will not prevent that of other clients from reaching the server, and that once begun the client will complete the three-way TCP handshake, is also violated. So in both models, the user agreements are violated.

Countermeasures thus focus on ensuring the resources needed for the legitimate handshake to complete are available, and every legitimate client gets access to the server. But the focus of the countermeasures differ. Some manipulate the opening of the connections at the end point; others control which packets, or the rate at which packets, are sent to the destination. In the latter case, the goal of the countermeasures is to ensure the implicit user agreements are enforced. In the former, if the focus is ensuring that client connection attempts will succeed after some time, the focus is the waiting time policy; otherwise, it is the user agreement.

Methods to provide these properties may involve intermediate systems, the server, or both.

7.4.2 Intermediate Systems

This approach tries to reduce the consumption of resources on the target by using routers to divert or eliminate illegitimate traffic. The key observation here is that

initially a subject is free to access all objects. The Chinese Wall model's constraints grow as the subject accesses more objects. However, from the initial state, the Bell-LaPadula Model constrains the set of objects that a subject can access. This set cannot change unless a trusted authority (such as a system security officer) changes subject clearances or object classifications. The obvious solution is to clear all subjects for all categories, but this means that any subject can read any object, which violates the CW-simple security condition.

Hence, the Bell-LaPadula Model cannot emulate the Chinese Wall model faithfully. This demonstrates that the two policies are distinct.

However, the Chinese Wall model can emulate the Bell-LaPadula Model; the construction is left as an exercise for the reader. (See Exercise 3.)

8.1.5 Clark-Wilson and Chinese Wall Models

The Clark-Wilson model deals with many aspects of integrity, such as validation and verification, as well as access control. Because the Chinese Wall model deals exclusively with access control, it cannot emulate the Clark-Wilson model fully. So, we consider only the access control aspects of the Clark-Wilson model.

The representation of access control in the Clark-Wilson model is the second enforcement rule, ER2. That rule associates users with transformation procedures and constrained data items on which they can operate. If one takes the usual view that "subject" and "process" are interchangeable, then a single person could use multiple processes to access objects in multiple CDs in the same COI class. Because the Chinese Wall model would view processes independently of who was executing them, no constraints would be violated. However, by requiring that a "subject" be a specific individual and including all processes executing on that subject's behalf, the Chinese Wall model is consistent with the Clark-Wilson model.

8.2 Clinical Information Systems Security Policy

Medical records require policies that combine confidentiality and integrity, but in a very different way than for brokerage firms. Conflicts of interest among doctors treating patients are not normally a problem. Patient confidentiality, authentication of both records and the personnel making entries in those records, and assurance that the records have not been changed erroneously are critical. Anderson [57] presents a model for such policies that illuminates the combination of confidentiality and integrity to protect patient privacy and record integrity.

Anderson defines three types of entities in the policy.

Definition 8–6. A *patient* is the subject of medical records, or an agent for that person who can give consent for the person to be treated.

Definition 8–7. *Personal health information* is information about a patient's health or treatment enabling that patient to be identified.

In more common parlance, the "personal health information" is contained in a medical record. We will refer to "medical records" throughout, under the assumption that all personal health information is kept in the medical records.

Definition 8–8. A *clinician* is a health-care professional who has access to personal health information while performing his or her job.

The policy also assumes that personal health information concerns one individual at a time. Strictly speaking, this is not true. For example, obstetrics/gynecology records contain information about both the father and the mother. In these cases, special rules come into play, and the policy does not cover them.

The policy is guided by principles similar to the certification and enforcement rules of the Clark-Wilson model. These principles are derived from the medical ethics of several medical societies, and from the experience and advice of practicing clinicians.[1]

The first set of principles deals with access to the medical records themselves. It requires a list of those who can read the records, and a list of those who can append to the records. Auditors are given access to copies of the records, so the auditors cannot alter the original records in any way. Clinicians by whom the patient has consented to be treated can also read and append to the medical records. Because clinicians often work in medical groups, consent may apply to a set of clinicians. The notion of groups abstracts this set well. Thus:

Access Principle 1: Each medical record has an access control list naming the individuals or groups who may read and append information to the record. The system must restrict access to those identified on the access control list.

Medical ethics require that only clinicians and the patient have access to the patient's medical record. Hence:

Access Principle 2: One of the clinicians on the access control list (called the *responsible clinician*) must have the right to add other clinicians to the access control list.

Because the patient must consent to treatment, the patient has the right to know when his or her medical record is accessed or altered. Furthermore, if a clinician who is unfamiliar to the patient accesses the record, the patient should be notified of the leakage of information. This leads to another access principle:

Access Principle 3: The responsible clinician must notify the patient of the names on the access control list whenever the patient's medical record is opened. Except for situations given in statutes, or in cases of emergency, the responsible clinician must obtain the patient's consent.

---

[1] The principles are numbered differently in Anderson's paper.

Erroneous information should be corrected, not deleted, to facilitate auditing of the records. Auditing also requires that all accesses be recorded, along with the date and time of each access and the name of each person accessing the record.

Access Principle 4: The name of the clinician, the date, and the time of the access of a medical record must be recorded. Similar information must be kept for deletions.

The next set of principles concern record creation and information deletion. When a new medical record is created, the clinician creating the record should have access, as should the patient. Typically, the record is created as a result of a referral. The referring clinician needs access to obtain the results of the referral, and so is included on the new record's access control list.

Creation Principle: A clinician may open a record, with the clinician and the patient on the access control list. If the record is opened as a result of a referral, the referring clinician may also be on the access control list.

How long the medical records are kept varies with the circumstances. Normally, medical records can be discarded after 8 years, but in some cases—notably cancer cases—the records are kept longer.

Deletion Principle: Clinical information cannot be deleted from a medical record until the appropriate time has passed.

Containment protects information, so a control must ensure that data copied from one record to another is not available to a new, wider audience. Thus, information from a record can be given only to those on the record's access control list.

Confinement Principle: Information from one medical record may be appended to a different medical record if and only if the access control list of the second record is a subset of the access control list of the first.

A clinician may have access to many records, possibly in the role of an advisor to a medical insurance company or department. If this clinician were corrupt, or could be corrupted or blackmailed, the secrecy of a large number of medical records would be compromised. Patient notification of the addition limits this threat.

Aggregation Principle: Measures for preventing the aggregation of patient data must be effective. In particular, a patient must be notified if anyone is to be added to the access control list for the patient's record and if that person has access to a large number of medical records.

Finally, systems must implement mechanisms for enforcing these principles.

Enforcement Principle: Any computer system that handles medical records must have a subsystem that enforces the preceding principles. The effectiveness of this enforcement must be subject to evaluation by independent auditors.

Anderson developed guidelines for a clinical computer system based on his model [54].

8.2.1 Bell-LaPadula and Clark-Wilson Models

Anderson notes that the Confinement Principle imposes a lattice structure on the entities in this model, much as the Bell-LaPadula Model imposes a lattice structure on its entities. Hence, the Bell-LaPadula protection model is a subset of the Clinical Information Systems security model. But the Bell-LaPadula Model focuses on the subjects accessing the objects (because there are more subjects than security labels), whereas the Clinical Information Systems model focuses on the objects being accessed by the subjects (because there are more patients, and medical records, than clinicians). This difference does not matter in traditional military applications, but it might aid detection of "insiders" in specific fields such as intelligence.

The Clark-Wilson model provides a framework for the Clinical Information Systems model. Take the constrained data items to be the medical records and their associated access control lists. The transaction procedures are the functions that update the medical records and their access control lists. The integrity verification procedures certify several items:

- A person identified as a clinician is a clinician (to the level of assurance required by the system).
- A clinician validates, or has validated, information in the medical record.
- When someone (the patient and/or a clinician) is to be notified of an event, such notification occurs.
- When someone (the patient and/or a clinician) must give consent, the operation cannot proceed until the consent is obtained.

Finally, the requirement of auditing (Clark-Wilson certification rule CR4) is met by making all records append-only, and notifying the patient whenever the access control list changes.

8.3 Originator Controlled Access Control

Mandatory and discretionary access controls (MACs and DACs) do not handle environments in which the originators of documents retain control over them even after those documents are disseminated. Graubart [813] developed a policy called ORGCON or ORCON (for "ORiginator CONtrolled") in which a subject can give another subject rights to an object only with the approval of the creator of that object.

EXAMPLE: The Secretary of Defense of the United States drafts a proposed policy document and distributes it to her aides for comment. The aides are not allowed to distribute the document any further without permission from the secretary. The secretary controls dissemination; hence, the policy is ORCON. The trust in this policy is that the aides will not release the document illicitly—that is, without the permission of the secretary.

In practice, a single author does not control dissemination; instead, the organization on whose behalf the document was created does. Hence, objects will be marked as ORCON on behalf of the relevant organization. The controller disseminating the object is called the *originator*, and the ones who receive copies of the objects are the *owners* of those objects.

Suppose a subject $s \in S$ marks an object $o \in O$ as ORCON on behalf of organization X. Organization X allows o to be disclosed to subjects acting on behalf of a second organization, Y, subject to the following restrictions:

1. The object o cannot be released to subjects acting on behalf of other organizations without X's permission.
2. Any copies of o must have the same restrictions placed on it.

Discretionary access controls are insufficient for this purpose, because the owner of an object can set any permissions desired. Thus, X cannot enforce condition 2.

Mandatory access controls are theoretically sufficient for this purpose, but in practice have a serious drawback. Associate a separate category C containing o, X, and Y and nothing else. If a subject $y \in Y$ wishes to read o, $x \in X$ makes a copy o' of o. The copy o' is in C, so unless $z \in Z$ is also in category C, y cannot give z access to o'. This demonstrates adequacy.

Suppose a member w of an organization W wants to provide access to a document d to members of organization Y, but the document is not to be shared with members of organization X or Z. So, d cannot be in category C because if it were, members $x \in X$ and $z \in Z$ could access d. Another category containing d, W, and Y must be created. Multiplying this by several thousand possible relationships and documents creates an unacceptably large number of categories.

A second problem with mandatory access controls arises from the abstraction. Organizations that use categories grant access to individuals on a "need to know" basis. There is a formal, written policy determining who needs the access based on common characteristics and restrictions. These restrictions are applied at a very high level (national, corporate, organizational, and so forth). This requires a central clearinghouse for categories. The creation of categories to enforce ORCON implies local control of categories rather than central control, and a set of rules dictating who has access to each compartment.

ORCON abstracts none of this. ORCON is a decentralized system of access control in which each originator determines who needs access to the data. No

centralized set of rules controls access to data; access is at the complete discretion of the originator. Hence, the MAC representation of ORCON is not suitable.

A solution is to combine features of the MAC and DAC models. The rules are as follows:

1. The owner of an object cannot change the access controls of the object.
2. When an object is copied, the access control restrictions of that source are copied and bound to the target of the copy.
3. The creator (originator) can alter the access control restrictions on a per-subject and per-object basis.

The first two rules are from mandatory access controls. They say that the system controls all accesses, and no one may alter the rules governing access to those objects. The third rule is discretionary and gives the originator power to determine who can access the object. Hence, this hybrid scheme is neither MAC nor DAC.

The critical observation here is that the access controls associated with the object are under the control of the *originator* and not the owner of the object. Possession equates to only some control. The owner of the object may determine to whom he or she gives access, but only if the originator allows the access. The owner may not override the originator.

8.3.1 Digital Rights Management

The owner of content, such as a movie or a book, may wish to control its distribution. When the content is given to a purchaser, the owner of the content may not want the purchaser to distribute it further without permission. The ORCON model describes this situation.

Definition 8–9. *Digital rights management* (DRM) is the persistent control of digital content.

This issue arises most often when dealing with copyrights. DRM technology controls what the recipient of a copyrighted work can, and cannot, do with that work.

EXAMPLE: A movie studio produces a new movie. It wishes to sell copies of the movie over the web. If the studio simply allowed people to purchase and download the movie from the studio's website, the purchaser could then redistribute it freely. To prevent this, the studio uses a DRM scheme. That scheme does not prevent the owner from further distributing the movie; however, the people to whom the owner distributes the movie cannot play it.

A DRM scheme has several elements [1968]. The basic ones are as follows:

- The *content* is the information being protected. It may be simple (as a single movie or book) or composite (in which several pieces of content, from different sources and with different rights, are combined).
- The *license* is the token that describes the uses to which the content may be put.
- A *grant* is that part of a license that gives specific authorizations to one or more entities. It may also include conditions that constrain the use of the grant.
- The *issuer* is the entity that issues the license. It may be the creator or a distributor.
- A *principal* is an identification of an entity. It is typically used in a license to identify to whom the license applies.
- A *device* is a mechanism used to view the content. It manages the licenses, principals, and any copies of the resource.

EXAMPLE: In the previous example, the content is the movie itself. The license is the token binding the playing of that movie to the specific copy that is downloaded. It includes a grant allowing the movie to be played on some set of equipment, with the condition that geographically the equipment be located within a particular country. The issuer is the movie studio or its authorized distributor. The principal is the user who downloaded the movie.

DRM schemes provide relationships among these elements. These relationships must satisfy three basic properties [1086]:

1. The system must implement controls on the use of the content. These controls constrain what clients can do with the content, so for example simply distributing the content encrypted and providing the keys to those authorized to see the content is insufficient (see Exercise 7).
2. The rules that constrain the users of the content must be associated with the content itself, and not the users.
3. The controls and rules must persist throughout the life of the content, even when the content is distributed in unauthorized ways or to unauthorized recipients.

EXAMPLE: Some music on Apple's iTunes store is protected by a DRM system called FairPlay.[2] The scheme is based upon cryptographic licensing of the system and the user.

[2]In 2007, Apple made some music available without DRM for a higher price. It subsequently incorporated this feature into music on the Apple cloud.

The user must authorize that particular system to play music. The iTunes program provides this capability. It first generates a globally unique number for the computer system and sends that to Apple's servers. The servers then add it to the list of systems authorized to play music for that user. At most five computer systems can be authorized per user; when the limit is reached, further authorizations are denied.

Suppose Sage purchases a song from iTunes. The song file is enciphered with the AES cipher (see Section 10.2.5) using a master key, which is in turn locked by a randomly generated user key from iTunes. iTunes then sends the user key to the Apple servers, which store it for future use. The key is encrypted on both the Apple server and on the local computer.

When Sage plays the song, iTunes first decrypts the user key, and then uses that to decipher the master key. It can then use the master key to decipher the song file and play it. Note that iTunes need not contact the Apple servers for authorization to play the song.

When Sage authorizes a new system, the Apple server sends that system all the user keys stored on the server so it can immediately play the music controlled by FairPlay. When she deauthorizes a system, it deletes all the locally stored user keys and notifies Apple to delete its global unique ID from the list of authorized computers.

When the content is protected, it can only be used on an authorized system. If the content is copied to an unauthorized system, the user keys will not be available and so the content cannot be used.

Conditions for use and for further distribution are stated using a *rights expression language*. Vendors often develop their own proprietary language to meet their needs. Other languages provide ways to express a wide variety of policies [1968, 1969].

EXAMPLE: Microsoft's PlayReady DRM [2187, 2188] uses a different model than Apple's FairPlay. It provides finer-grained control over use. The content is first enciphered using the AES algorithm (see Section 10.2.5). The cryptographic key is made available to a license server, and the content is made available for distribution to clients. To play the content, the PlayReady client downloads the content and requests a license from a PlayReady license server. The license server authenticates the client and, if successful, returns a license. The client then checks the constraints in the license, and if they allow playback, the client uses the license key to decipher the content and play it.

The client's request for a license includes both an identifier for the content to be played and the client's public key (see Section 10.3). The license server authenticates the client and verifies both the user and client are authorized to play the content. It then constructs a license containing the content key and usage constraints, enciphers this using the client's public key, and sends it to the client. The client then decrypts the license with its private key.

The rights expression language supports several different types of constraints. Temporal constraints allow the content to be viewed over a specific period of time, enabling the renting of the content. They also allow a validity period, after which the license must be renewed; this allows subscription-based services. Purchasing constraints allow the consumer to buy content, and the language provides means to express constraints on copying, transferring, or converting the content. Some applications, such as for streaming live television content, require constraints based on geographical location and on availability (as for example when a sporting event is not to be available near where the game is played). PlayReady supports these constraints, too [2189].

Some DRM technologies create unanticipated problems, especially when the software implementing the DRM modifies system programs or the kernel without the user's understanding of the effects of such modification. The example of Sony's DRM mechanism on page 778 serves as a warning of how not to implement DRM.

8.4 Role-Based Access Control

The ability, or need, to access information may depend on one's job functions.

EXAMPLE: Allison is the bookkeeper for the Department of Mathematics. She is responsible for balancing the books and keeping track of all accounting for that department. She has access to all departmental accounts. She moves to the university's Office of Admissions to become the head accountant (with a substantial raise). Because she is no longer the bookkeeper for the Department of Mathematics, she no longer has access to those accounts. When that department hires Sally as its new bookkeeper, she will acquire full access to all those accounts. Access to the accounts is a function of the job of bookkeeper, and is not tied to any particular individual.

This suggests associating access with the particular job of the user [668].

Definition 8–10. A *role* is a collection of job functions. Each role r is authorized to perform one or more transactions (actions in support of a job function). The set of authorized transactions for r is written $trans(r)$.

Definition 8–11. The *active role of a subject s*, written $actr(s)$, is the role that s is currently performing.

Definition 8–12. The authorized roles of a subject s, written $authr(s)$, is the set of roles that s is authorized to assume.

Definition 8–13. The predicate $canexec(s, t)$ is true if and only if the subject s can execute the transaction t at the current time.

Three rules reflect the ability of a subject to execute a transaction.

Axiom 8.7. Let S be the set of subjects and T the set of transactions. The *rule of role assignment* is

$$(\forall s \in S)(\forall t \in T)[canexec(s, t) \rightarrow actr(s) \neq \varnothing]$$

This axiom simply says that if a subject can execute any transaction, then that subject has an active role. This binds the notion of execution of a transaction to the role rather than to the user.

Axiom 8.8. Let S be the set of subjects. Then the *rule of role authorization* is

$$(\forall s \in S)[actr(s) \subseteq authr(s)]$$

This rule means that the subject must be authorized to assume its active role. It cannot assume an unauthorized role. Without this axiom, any subject could assume any role, and hence execute any transaction.

Axiom 8.9. Let S be the set of subjects and T the set of transactions. The *rule of transaction authorization* is

$$(\forall s \in S)(\forall t \in T)[canexec(s, t) \rightarrow t \in trans(actr(s))]$$

This rule says that a subject cannot execute a transaction for which its current role is not authorized.

The forms of these axioms restrict the transactions that can be performed. They do not ensure that the allowed transactions can be executed. This suggests that role-based access control (RBAC) is a form of mandatory access control. The axioms state rules that must be satisfied before a transaction can be executed. Discretionary access control mechanisms may further restrict transactions.

EXAMPLE: Some roles subsume others. For example, a trainer can perform all actions of a trainee, as well as others. One can view this as containment. This suggests a hierarchy of roles, in this case the trainer role containing the trainee role. As another example, many operations are common to a large number of roles. Instead of specifying the operation once for each role, one specifies it for a role containing all other roles. Granting access to a role R implies that access is granted for all roles contained in R. This simplifies the use of the RBAC model (and of its implementation).

If role r_i contains role r_j, we write $r_i > r_j$. Using our notation, the implications of containment of roles may be expressed as

$$(\forall s \in S)[r_i \in authr(s) \wedge r_i > r_j \rightarrow r_j \in authr(s)]$$

EXAMPLE: RBAC can model the separation of duty rule [1111]. Our goal is to specify separation of duty centrally; then it can be imposed on roles through containment, as discussed in the preceding example. The key is to recognize that the users in some roles cannot enter other roles. That is, for two roles r_1 and r_2 bound by separation of duty (so the same individual cannot assume both roles)

$$(\forall s \in S)[r_1 \in authr(s) \rightarrow r_2 \notin authr(s)]$$

Capturing the notion of mutual exclusion requires a new predicate.

Definition 8–14. Let r be a role, and let s be a subject such that $r \in auth(s)$. Then the *mutually exclusive authorization* set $meauth(r)$ is the set of roles that s cannot assume because of the separation of duty requirement.

Putting this definition together with the above example, the principle of separation of duty can be summarized as [1111]

$$(\forall r_1, r_2 \in R)[r_2 \in meauth(r_1) \rightarrow [(\forall s \in S)[r_1 \in authr(s) \rightarrow r_2 \notin authr(s)]]]$$

Sandhu et al. [1655] have developed a family of models for RBAC. RBAC_0 is the basic model. RBAC_1 adds role hierarchies; RBAC_2 adds constraints; and RBAC_3 adds both hierarchies and constraints by combining RBAC_1 and RBAC_2.

RBAC_0 has four entities. *Users* are principals, and *roles* are job functions. A *permission* is an access right. A *session* is a user interaction with the system during which the user may enter any role that she is authorized to assume. A user may be in multiple roles at one time. More formally:

Definition 8–15. RBAC_0 has the following components:

- A set of users U, a set of roles R, a set of permissions P, and a set of sessions S
- A relation $PA \subseteq P \times R$ mapping permissions to roles
- A relation $UA \subseteq U \times R$ mapping users to roles
- A function $user : S \rightarrow U$ mapping each session $s \in S$ to a user $u \in U$
- A function $roles : S \rightarrow 2^R$ mapping each session $s \in S$ to a set of roles $roles(s) \subseteq \{r \in R | (user(s), r) \in UA\}$, where s has the permissions $\bigcup_{r \in roles(s)} \{p \in P | (p, r) \in PA\}$

The last means that when a user assumes role r during session s, r and hence the user assuming r acquires a set of permissions associated with r.

RBAC_1 adds role hierarchies. A hierarchy is a means for structuring roles using containment, and each role is less powerful than those higher in the hierarchy. Each interior role contains the job functions, and hence permissions, of its subordinate roles. Formally:

Definition 8–16. RBAC_1 makes the following changes to the definition of RBAC_0:

- Add a partial order $RH \subseteq R \times R$ called the role hierarchy
- Change the function $roles : S \rightarrow 2^R$ to map each session $s \in S$ to a set of roles $roles(s) \subseteq \{r \in R | (\exists r' \geq r)(user(s), r') \in UA\}$, where s has the permissions $\bigcup_{r \in roles(s)} \{p \in P | (\exists r'' \geq r)(p, r'') \in PA\}$, and for $r_1, r_2 \in R$, $r_1 \geq r_2$ means $(r_2, r_1) \in RH$.

The last means that when user u assumes a role r with subordinate roles, u can establish a session with any combination of the subordinate roles, and that session receives the permissions it would acquire in RBAC_0, plus any permissions that subordinate roles have. So, in effect, the user gets the permissions assigned to the role assumed, as well as the permissions of all roles subordinate to that role.

Role hierarchies can limit the inheritance of permissions through *private roles*. Figure 8–3 shows a hierarchy where the employees report to line management. But employees in this organization have the right to air grievances to an ombudsman who does not answer to line management, and indeed will not report to them. In this case, the ombudsman role is private with respect to the line management.

RBAC_2 is based on RBAC_0, but adds constraints on the values that the components can assume. For example, two roles may be mutually exclusive; this constrains the values that can be in the sets UA and PA. Another example is to constrain the function $roles$ to allow a user to be in exactly one role at a time. Formally:

Definition 8–17. RBAC_2 adds to the RBAC_0 model constraints that determine allowable values for the relations and functions.

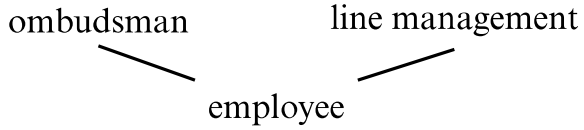

Figure 8–3 An example of a private role.

$RBAC_3$ combines all three models. One can think of it as adding hierarchies to $RBAC_2$:

> **Definition 8–18.** $RBAC_3$ combines the $RBAC_1$ and $RBAC_2$ models, providing both role hierarchies and constraints that determine allowable values for the relations and functions.

An interesting use for RBAC is to manage the role assignments and privilege assignments—in essence, have the model manage its own components. To do this, a set of administrative roles AR and a set of administrative permissions AP are disjoint from the ordinary roles R and permissions P. Constraints allow nonadministrative permissions to be assigned to nonadministrative roles only, and administrative permissions to be assigned to administrative roles only. The ARBAC97 and the ARBAC02 models [1471, 1654] extend $RBAC_3$ to cover role-based administration of roles for RBAC.

EXAMPLE: Dresdner Bank developed an enterprise-wide role-based access control system called FUB to manage access rights based upon a combination of job function and position within the bank [1674]. When an employee launches an application, the application transmits a request contains identifying information for both the user and the application to the FUB. The FUB returns the appropriate profile that contains the rights for that application and that employee role.

Dresdner Bank has 65 official positions such as Clerk, Branch Manager, and Member of the Board of Directors. There are 368 job functions. A role is defined by the position and job function, but only about 1300 roles are in use. The Human Resources Department creates the roles and the users, and assigns users to roles. Thus, when a user leaves, part of the exit processing by the Human Resources Department is to delete the user's assignment to roles. The Application Administrator assigns access rights to an application. These rights are represented by numbers the meaning of which is known only to the Application Administrator. The Application Administrator then passes the numbers to the FUB administrator, who assigns the roles that can access the application using the numbers.

In the definition of roles, the positions form a partial order. Thus, one role is superior to another when the first role's job function is higher than the second's, and the job functions are the same. An alternative approach is to have the job functions be ordered hierarchically, for example by saying that the function auditor includes the functions of an accountant. In this context, the positions could be ignored, and the hierarchy based solely on the partial ordering of job functions.

The analysis of the FUB system using RBAC as a model increased the bank's confidence that their approach was sound.

A problem that often arises in practice is defining useful roles and determining the permissions they need. This process is called *role engineering* [472].

A similar, often more complex, problem arises when two organizations that both use RBAC merge, as the roles each defines are rarely compatible with the roles the other defines—yet many job functions will overlap. Role mining is the process of analyzing existing role and user permission to determine an optimal assignment of permissions to roles. This is an NP-complete problem in theory, but in practice near-optimal, or even optimal, solutions can be produced [637, 714, 1210, 1916, 1917, 2088].

8.5 Break-the-Glass Policies

Sometimes security requirements conflict. Consider a health-care policy that controls access to medical records. In an emergency, doctors may need to override restrictions to get immediate access to a patient's medical record, for example if the patient is unconscious and unable to give consent. Povey [1540] proposed a control to handle this situation:

> **Definition 8–19.** A *break-the-glass* policy allows access controls to be overridden in a controlled manner.

The term comes from "breaking-the-glass" to activate an emergency alarm. These policies are added to standard access control policies, and enable them to be overridden. The overriding is logged for future analysis.

Break-the-glass schemes are invoked when a user attempts to access information and the access is denied. The system either informs the user of the break-the-glass option, or the user knows about it through external sources. In either case, the user can override the denial. Should she do so, the system immediately notifies those whom it is supposed to notify, and logs the notification and the user's actions [671].

EXAMPLE: The Rumpole policy [1252] implements a break-the-glass policy. *Evidential rules* define how evidence is assembled to create the context in which a break-the-glass request is made. *Break-glass rules* define permissions, which may include constraints such as imposing an obligation to justify the need for the break-the-glass action at a later time. *Grant policies* define how the break-glass rules are combined to determine whether to grant the override.

Rumpole's enforcement model consists of a policy decision point and an enforcement point. A break-the-glass request consists of a subject, the desired action, the resource, and obligations that the subject will accept should the override be granted. The decision point grants the request unconditionally, denies the request, or returns the request with a set of obligations that the subject must

accept for the break-the-glass request to be granted. The subject then sends a new request with the new obligations.

8.6 Summary

The goal of this chapter was to show that policies typically combine features of both integrity and confidentiality policies. The Chinese Wall model accurately captures requirements of a particular business (brokering) under particular conditions (the British law). The Clinical Information Systems model does the same thing for medical records. Both models are grounded in current business and clinical practice.

ORCON and RBAC take a different approach, focusing on which entities *will* access the data rather than on which entities *should* access the data. ORCON allows the author (individual or corporate) to control access to the document; RBAC restricts access to individuals performing specific functions. The latter approach can be fruitfully applied to many of the models discussed earlier.

Break-the-glass policies provide conditions under which normal access control rules are to be violated. They are useful in cases where unanticipated situations requiring human intervention may occur.

8.7 Research Issues

Policies for survivable systems, which continue functioning in the face of massive failures, are critical to the secure and correct functioning of many types of banking, medical, and governmental systems. Of particular interest is how to enable such systems to reconfigure themselves to continue to work with a limited or changed set of components.

ORCON provides controls that are different from DAC and MAC. Are other controls distinct enough to be useful in situations where DAC, MAC, and ORCON don't work? How can integrity and consistency be integrated into the model?

Most DRM schemes are developed for the specific organization that wants to use DRM. The multiplicity of these systems inhibits acceptance. For example, a consumer who wishes to buy several digital movies and books from four movie studios and three publishers may have to use seven different DRM schemes, each with its own user interface. Thus, the development of interoperable DRM schemes is an area of active research. Another area is the usability of DRM mechanisms, because they must balance this with the protection of the rights of the content owners. This also introduces privacy concerns, another fertile area of research.