CHAPTER 5

Verifiable Security Goals

In this chapter, we examine access control models that satisfy the *mandatory protection system* of Definition 2.4 in Chapter 2. A mandatory protection system provides a tamperproof description of the system's access control policy. A mandatory protection system consists of: (1) a *protection state* that defines the operations that a fixed set of subject labels can perform on a fixed set of object labels; (2) a *labeling state* that maps system processes and resources to their subject and object labels, respectively; and (3) a *transition state* that defines the legal ways that system processes and resources may be assigned to new labels. As such, it manages the access rights that all system processes will ever obtain.

A mandatory protection system is necessary for a secure operating system to implement two of the requirements of the *reference monitor concept* as defined in Definition 2.6: tamperproofing and verifiability. A mandatory protection system is tamperproof from untrusted processes as the system defines the labeling of subjects and objects, transitions in these labels, and the resulting protection state. Because these access control models in mandatory protection systems only allow the system to modify the protection state, they are called *mandatory access control models* [179] [1]. Also, a mandatory protection system describes the access control policy that we use to verify the enforcement of the system's security goals. Often, such models support the definition of policies that describe concrete security goals, which we will define below.

In this chapter, we describe mandatory protection systems and the security goals that they imply. In general, a secure operating system should ensure the enforcement of secrecy goals, including for covert channels, and integrity goals. We present the basic concept for expressing secrecy and integrity goals, information flows, and then describe models for expressing these goals in mandatory access control policies. The models that we present here mainly focus on either secrecy or integrity, not both, but there are several mandatory models that combine both facets, such as Type Enforcement [33], Chinese Wall [37], and Caernarvon [278]. Unifying secrecy and integrity effectively in systems is an ongoing challenge in practice, as we will see.

5.1 INFORMATION FLOW

Secure operating systems use *information flow* as the basis for specifying secrecy and integrity security requirements. Conceptually, information flow is quite simple.

Definition 5.1. An *information flow* occurs between a subject $s \in S$ and an object $o \in O$ if the subject performs a *read* or *write* operation on the object. The information flow $s \to o$ is from the

[1] Historically, the term *mandatory access control model* has been used to describe specific access control models, such as the multilevel secrecy models, but we use the broader definition in this book that is based on how the policies in these models are administered.

subject to the object if the subject writes to the object. The information flow $s \leftarrow o$ is from the object to the subject if the subject reads from the object.

Information flow represents how data moves among subjects and objects in a system. When a subject (e.g., process) reads from an object (e.g., a file), the data from the object flows into the subject's memory. If there are secrets in the object, then information flow shows that these secrets may flow to the subject when the subject reads the object. However, if the subject holds the secrets, then information flow also can show that the subject may leak these secrets if the subject writes to the object.

Note that every operation on an object is either an information flow read (i.e., extracts data from the object), an information flow write (i.e., updates the object with new data), or a combination of both. For example, *execute* reads the data from a file to prepare it for execution, so the process reads from the file. When we *delete* a file from a directory, the directory's set of files is changed. The result is that an entire protection state of a mandatory protection system can be represented by a set of information flow reads and writes.

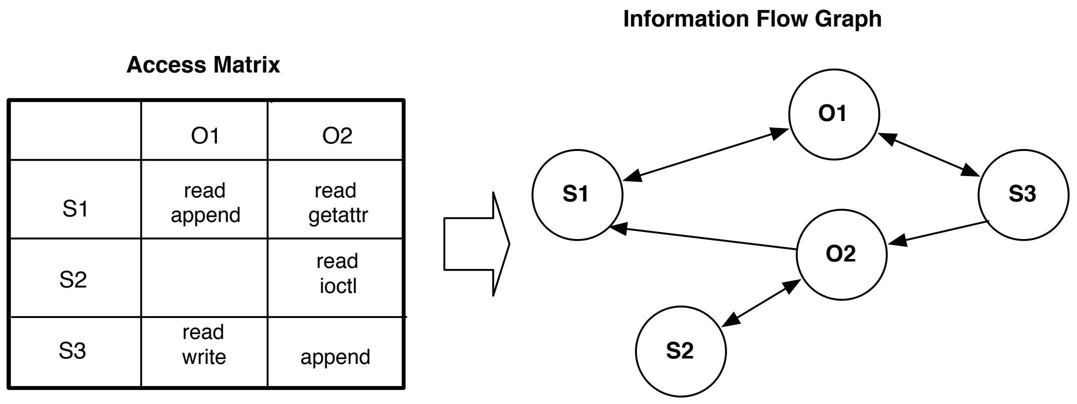
Thus, any protection state can be represented by an information flow graph.

Definition 5.2. An *information flow graph* for a protection state is a directed graph $G = (V, E)$ where: (1) the set of vertices V consists of the union of the set of subjects and set of objects in the protection state and (2) the set of directed edges E consists of each read and write information flow in the protection state.

An information flow graph for a protection state can be constructed as follows. First, we create a vertex for each subject and object in the protection state. Then, we add the information flow edges. To do this, we determine whether each operation in the protection state results in a read, write, or combination information flow. Then, we add an information flow edge from a subject vertex to an object vertex when the subject has permission to a write information flow operation for the object in the protection state. Likewise, we add an information flow edge from an object vertex to a subject vertex when the subject has permission to a read information flow operation in the protection state.

Example 5.3. Consider the access matrix shown in Figure 5.1. It defines the set of operations that the subjects $S1$, $S2$, and $S3$ can perform on objects $O1$ and $O2$. Note that some operations, such as `append`, `getattr`, and `ioctl` have to be mapped to their resultant information flows, write, read, and both, respectively. As a result, this access matrix represents the corresponding information flow graph shown in Figure 5.1.

Information flow is used in secure operating systems as an approximation for secrecy and integrity. For secrecy, the information flow edges in the graph indicate all the paths by which data may be *leaked*. We can use the graph to determine whether a secret object o may be leaked to an



	O1	O2
S1	read append	read getattr
S2		read ioctl
S3	read write	append

Figure 5.1: The information flow graph on the right represents the information flows described by the access control matrix on the left.

unauthorized subject s. If there exists a path in the information flow graph from o to s, then there is an unauthorized leak in the corresponding protection state.

For integrity, we require that no high integrity subject *depends* on any low integrity subjects or objects. We can use the graph to determine whether a high integrity subject $s1$ receives input from a low integrity subject $s2$ (e.g., an attacker). If there exists a path in the information flow graph from $s2$ to $s1$, then the high integrity subject $s1$ receives input from the low integrity subject $s2$. When a high integrity subject receives input from a low integrity subject, it is assumed that it depends on that low integrity subject.

5.2 INFORMATION FLOW SECRECY MODELS

For information flow secrecy, we want to ensure that no matter which programs a user runs, she cannot leak information to an unauthorized subject. The classical problem is that the user may be coerced into running a program that contains malware that actively wants to leak her information. For example, a *Trojan horse* is a type of malware that masquerades as a legitimate program, but contains a malicious component that tries to leak data to the attacker.

The access control models of UNIX and Windows cannot prevent such an attack, because: (1) they do not account for all the information flows that may be used to leak information and (2) the policies are discretionary, so the malware can modify the policy to introduce illegal information flows. First, UNIX and Windows policies often define some files shared among all users, but any user's secret data may also be leaked through such public files by well-crafted malware. For example, the UNIX model permits the sharing of files with all users by granting read access to the `others`. However, if a user has write access to any file that `others` can read, then the malware can leak secrets by writing them to this file. Second, discretionary access control (DAC) protection systems, such as those used by UNIX and Windows, permit the users to modify access control policies. However, any program that the user runs can modify the permission assignments to files that the user owns. Thus, any of the user's files may be leaked by malware simply by changing the file permissions.

The security policy models used in mandatory protection systems aim to solve these problems. First, they explicitly restrict information flows to not leak secret data. Second, such models do not permit users or their programs to modify the information flows permitted. In addition to the secrecy models presented here, see the High-Water Mark model of Weissman [329].

5.2.1 DENNING'S LATTICE MODEL

Denning refined the general information flow graph to express information flow secrecy requirements [70, 271] based partly on the work of Fenton [93].

Definition 5.4. An *information flow model* is a quintuple $\{N, P, SC, \oplus, \rightarrow\}$, where: (1) N is the set of storage *objects* (e.g., files); P is the set of *subjects* (e.g., processes) that cause information flows; SC is a set of *security classes*; (2) $\rightarrow \subseteq SC \times SC$ is a binary *can-flow* relation on SC; and (3) $\oplus$: $SC \times SC \rightarrow SC$ is a binary *join* operator on SC.

In a information flow model, each subject and object is assigned a security class. Secure classes are labels in the mandatory protection system defined in Definition 2.4, and both subjects and objects may share security classes. For example, a subject and object may both be assigned to security class X. However, another subject, to whom X data must not be leaked, is assigned security class Y. The *can-flow* relation $\rightarrow$ defines the legal information flows in the model. That is, $Y \rightarrow X$ specifies that information at security class Y can flow to subjects and objects in security class X [2]. Since we have a secrecy requirement that information in security class X not be leaked to subjects and objects of security class Y, $X \not\rightarrow Y$. The *join* operator determines the security class that results from combining data of two distinct security classes $X \oplus Y = Z$. In this case, the combination of data from X and Y security classes is labeled Z.

Example 5.5. Figure 5.2 shows two information flow model policies. In (a), this policy isolates users $u_1, u_2, u_3, ..., u_i$ by assigning them to distinct security classes. Any data in security class u_i cannot be read or written by any process running with security class u_j where $i \neq j$.

Figure 5.2 (b) shows an information flow model policy that totally orders security classes, such that data in higher classes will not be leaked to lower security class. These security classes represent the traditional governmental secrecy classes, *top-secret*, *secret*, *confidential*, and *unclassified*. In this policy, *top-secret* data may not be read by processes running in the lower classes in the information flow model policy. Further, processes running in *top-secret* may not write to objects in the lower security classes. Lower security classes, such as *secret*, may have information flow by permitting higher security classes (e.g., *top-secret*) to read their data or by writing up to objects in the higher

[2] Infix notation.

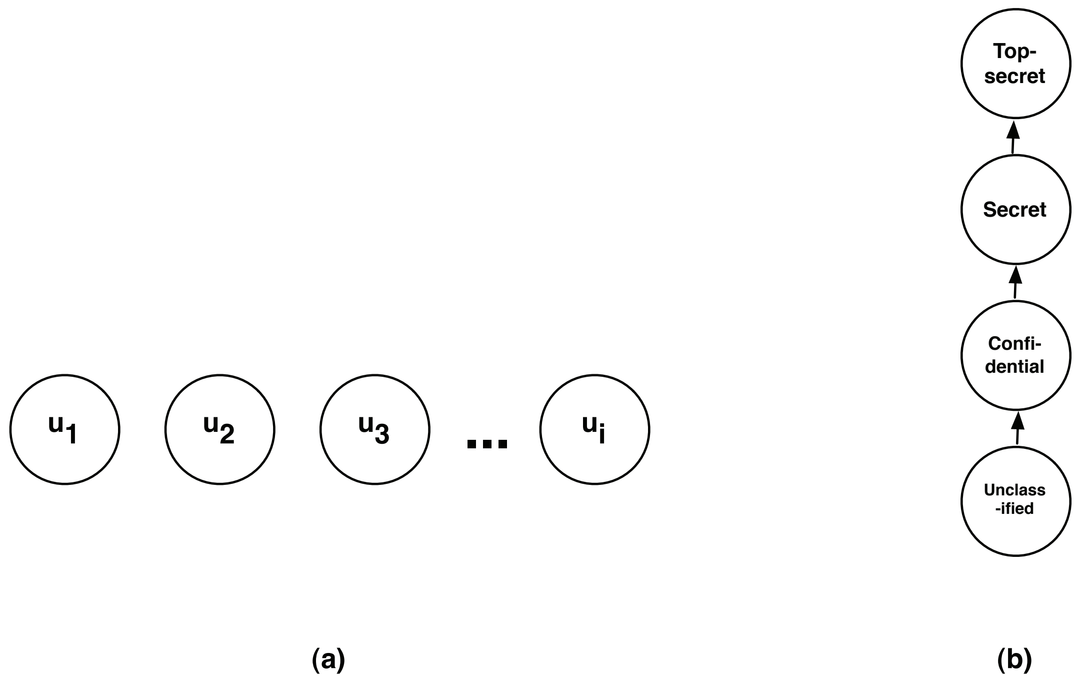


Figure 5.2: Two information flow model policies: (a) consists of isolated security class where no information flows among them and (b) is a totally-ordered sequence of security classes where information flows upwards only.

security classes. Since processes in the lower security classes do not even know the name of objects in higher security classes, such writing is implemented by a *polyinstantiated* file system where the files have instances at each security level, so the high security process can read the lower data and update the higher secrecy version without leaking whether there is a higher secrecy version of the file.

Such information flow model policies actually can be represented by a finite *lattice*. Denning defines four axioms required for lattice policies.

Definition 5.6. An information flow model forms a *finite lattice* if it satisfies the following axioms.

1. The set of security classes SC is finite.
2. The *can-flow* relation $\rightarrow$ is a partial order on SC.
3. SC has a lower bound with respect to $\rightarrow$.

4. The *join* operator $\oplus$ is a totally defined least upper bound operator.

The key result comes from axiom 4. In a finite lattice, the *join* operator is defined for any combination of security classes. Thus, for $X_1 \oplus X_2 \oplus ... \oplus X_n = Z$, the security class Z that results from a combination of data from any security classes in SC must also be in SC. For lattice policies, the results of any join operation can be modeled because the security class of the operation can always be computed.

Note that the Example 5.5(b) satisfies the Denning axioms, so it is a finite lattice. Any combination of data can be assigned a security class. For example, when a *secret* process generates data read from *confidential* and *unclassified* inputs, the data generated is labeled *secret*, which is the least upper bound of the three security classes. However, we can see that Example 5.5(a) does not satisfy axioms 3 and 4, so we cannot label the results from any operation that uses the data from two different users.

Finally, a useful concept is the inverse of the *can-flow* relation, called the *dominance* relation. Dominance is typically used in the security literature.

Definition 5.7. $A \geq B$ (read as A *dominates* B) if and only if $B \rightarrow A$. The *strictly dominates* relation $>$ is defined by $A > B$ if and only if $A \geq B$ and $A \neq B$. We say that A and B are *comparable* if $A \geq B$ or $B \geq A$. Otherwise, A and B are *incomparable*.

Dominance indicates which security class is more sensitive (i.e., contains data that is more secret). From a security perspective, dominance defines the information flows that are not allowed. That is, if $A > B$, then A's data must not flow to B or this constitutes a leak.

5.2.2 BELL-LAPADULA MODEL

The most common information flow model in secure operating systems for enforcing secrecy requirements is the Bell-LaPadula (BLP) model [23]. There are a variety of models associated with Bell and LaPadula, but we describe a common variant here, known as the Multics interpretation. This BLP model is a finite lattice model where the security classes represent two dimensions of secrecy: *sensitivity level* and *need-to-know*. The sensitive level of data is a total order indicating secrecy regardless of the type of data. In the BLP model, these levels consist of the four governmental security classes mentioned previously: *top-secret*, *secret*, *confidential*, and *unclassified*. However, it was found that not everyone with a particular security class "needs to know" all the information labeled for that class. The BLP model includes a set of *categories* that describe the topic areas for data, defining the need-to-know access. The BLP model assigns a sensitivity level that defines the secrecy level that the subject is authorized for, and also a set of categories, called a *compartment*, to each subject and object. The combination of sensitivity level and compartment for a subject are often called its *clearance*. For objects, their combination of sensitivity level and compartment are called its *access class*.

Example 5.8. Figure 5.3 shows a Bell-LaPadula policy with two sensitivity levels and three categories. The edges show the direction of information flow authorized by the Bell-LaPadula policy. If

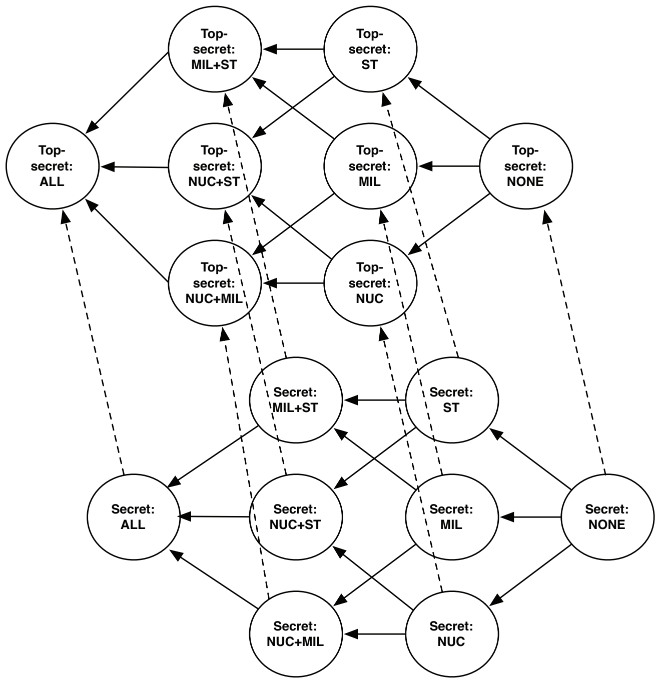


Figure 5.3: This a Haase diagram (with the information flow direction added in edges) of a Bell-LaPadula policy consisting of two sensitivity levels (*top-secret* and *secret* where *top-secret* dominates) and three categories (*NUC*, *MIL*, and *ST*). The edges show the information flows authorized by the Bell-LaPadula model for this lattice.

a subject is cleared for `top-secret:MIL`, it is able to read from this class, `top-secret:none`, and `secret:MIL`. However, information cannot flow to the `top-secret:MIL` class from `top-secret:MIL+ST` or others that include categories besides `MIL`. Even information that is labeled with the `secret` sensitivity level, but has additional categories may not flow to `top-secret:MIL`. Of course, subjects at the `top-secret:MIL` clearance can write to any `top-secret` class that includes the category `MIL`, but none of `secret` classes. The latter is not possible because the sensitivity level `top-secret` dominates or is incomparable to any `secret` class. Writes may only be allowed to classes that dominate the subject's clearance.

The BLP model defines two key properties for information flow secrecy enforcement.

Definition 5.9. The *simple-security property* states that subject s can read an object o only if $SC(s) \geq SC(o)$. Thus, a subject can only read data that at their security class or is less secret. Second, the

$\star$-*security property* states that subject s can write an object o only if $SC(s) \leq SC(o)$. Thus, a subject can only write data that is at their security class or is more secret.

The simple-security property solves the obvious problem that subjects should not read data that is above their security class. That is, the BLP policy identifies unauthorized subjects for data as subjects whose security class is dominated by the object's security class. Thus, the simple-security property prevents unauthorized subjects from receiving data.

The $\star$-security property handles the more subtle case that results when the user runs malware, such as a Trojan horse. This property prevents any process from writing secrets to a security class that they dominate, so even if the process is a Trojan horse, it cannot leak data to unauthorized subjects.

The BLP model and its variants are also called multilevel security models and mandatory access control models. A *multilevel security* (MLS) model is a lattice model consisting of multiple sensitivity levels. While the BLP models are simply instances of MLS models, they are MLS models used predominantly in practice. Thus, the BLP models are synonymous with MLS models.

The Bell-LaPadula model implements a *mandatory protection system* (see Definition 2.4 in Chapter 2. First, this model implements a *mandatory protection state*. A fixed set of security classes (labels), consisting of sensitivity levels and categories, are defined by trusted administrators. The dominance relationship among security classes is also fixed at policy definition. Since information flow in the Bell-LaPadula model is determined by the dominance relation, the set of accesses that are possible are fixed.

Second, the Bell-LaPadula model defines a *labeling state* where subjects and objects are labeled based on the label of the process that created them. At create time, a subject or object may be labeled at a security class that dominates the security class of the creating process. Once the subject or object is created and labeled, its label is static.

Third, the Bell-LaPadula model defines a null *transition state*. That is, once a subject or object is labeled (i.e., when it is created), the label may not change. The assumption that the assignment of security classes to subjects and objects does not change is called *tranquility*. Thus, the Bell-LaPadula model satisfies the requirements for a mandatory protection system.

5.3 INFORMATION FLOW INTEGRITY MODELS

Secure operating systems sometimes include policies that explicitly protect the integrity of the system. Integrity protection is more subtle than confidentiality protection, however. The integrity of a system is often described in more informal terms, such as "it behaves as expected." A common practical view of integrity in the security community is: *a process is said to be high integrity if it does not* depend *on any low integrity inputs*. That is, if the process's code and data originate from known, high integrity sources, then we may assume that the process is running in a high integrity manner (e.g., as we would expect).

Like data leakage, dependence can also be mapped to information flows. In this case, if a high integrity process reads from an object that may be written to by a low integrity process, then the high integrity process may be compromised. For example, if an attacker can modify the configuration files, libraries, or code of a high integrity process, then the attacker can take control of the process, compromising its integrity.

5.3.1 BIBA INTEGRITY MODEL

Based on this view, an information flow model was developed by Biba [27], now called the Biba integrity model [3]. The Biba model is a finite lattice model, as described above, but the model defines properties for enforcing information flow integrity.

Definition 5.10. The *simple-integrity property* states that subject s can read an object o only if $SC(s) \leq SC(o)$. Thus, a subject can only read data that is at their security class or is higher integrity. Second, the $\star$-*integrity property* states that subject s can write an object o only if $SC(s) \geq SC(o)$. Thus, a subject can only write data that is at their security class or is lower integrity.

Example 5.11. A Biba lattice model only uses *integrity levels*, not categories. Like the Bell-LaPadula model, the integrity levels are typically totally-ordered. However, unlike the Bell-LaPadula model, there is no commonly-agreed set of levels. An example Biba lattice could include the integrity levels of `trusted`, `system`, `application`, `user`, and `untrusted` where `trusted` is the is highest integrity level and `untrusted` is the lowest. In the Biba model, information flows are only allowed from the higher integrity levels to the lower integrity levels. Thus, subjects and objects that are labeled `untrusted` should not be able to write to subjects or objects in the other levels.

Thus, the flows allowed in a Biba policy are reverse of the flows allowed in the BLP model. Where BLP allows a subject to read objects of a security class dominated by the subject, Biba does not because the objects in a lower security class in Biba are lower integrity. As a result, if we combine BLP and Biba using a single set of security classes, subjects can only read and write data in their security class, which is too limiting for most systems. Also, the integrity and secrecy of a particular object are not necessarily the same, so, in practice two lattices, one for secrecy and one for integrity are created. Further, these lattices contain two distinct sets of security classes. From here, we will call the nodes in a secrecy lattice SC, *secrecy classes*, and the nodes in an integrity lattice SC_i, *integrity classes*. The two sets of security classes are disjoint.

Example 5.12. Figure 5.4 shows Biba and Bell-LaPadula lattices (levels only) where both must be authorized before an operation is permitted. This joint model enables the enforcement of secrecy and integrity goals. Each subject and object is assigned to a secrecy class and an integrity class which then defines the allowed information flows.

[3]There were actually three different integrity models proposed in this paper, called *low-water mark integrity*, *ring integrity*, and *strict integrity*. The strict integrity model became the most prominent of the three models, and it is the one we now call the Biba integrity model. The *low-water mark integrity* gained renewed interest later under the acronym LOMAC, see Section 5.3.2.

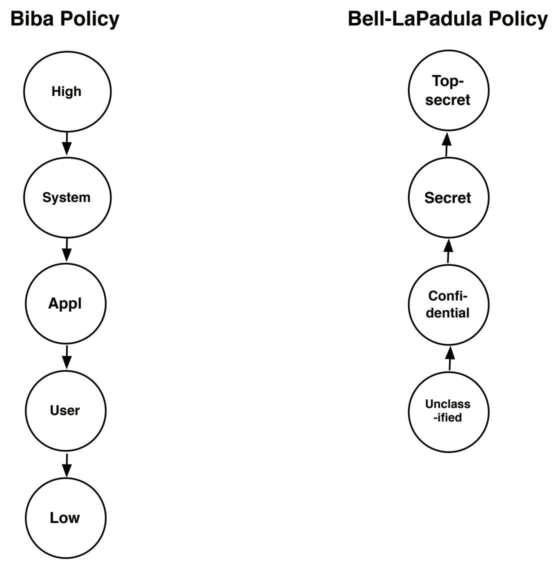


Figure 5.4: For a system that enforces both secrecy and integrity goals, Biba and Bell-LaPadula can be jointly applied. Subjects and objects will be assigned both Biba integrity classes and Bell-LaPadula secrecy classes from the set as shown.

Suppose that an object is a `top-secret, user` object (i.e., secrecy class, then integrity class). Only subjects that are authorized to read both `top-secret` objects according to the Bell-LaPadula policy and `user` objects according to the Biba policy are permitted to read the object. For example, neither `secret, low` nor `top-secret, appl` are allowed to read this object because both the Biba and Bell-LaPadula requirements are not satisfied for these subjects. A subject must be able to both read the object in Bell-LaPadula (i.e., be `top-secret`) and read the object in Biba (i.e., be `low` or `user`).

As for reading, a subject's integrity and secrecy classes must individually permit the subject to write to the object for writes to be authorized.

Lipner defined specific integrity and secrecy levels and categories could be chosen with the aim of constructing a useful composite model for commercial systems [190]. Lipner chose two secrecy levels, *audit manager* and *system low*, where audit manager is more secret (i.e., logs cannot be read once they are written). Also, there are three secrecy categories, *production*, *development*, *system development*. Production is used for production code and data, and the development categories separate system and user program development. In modern systems, we probably need at least one more secrecy level or category for untrusted programs (e.g., to prevent them from downloading the commercial entity's data). For integrity, the lattice has three levels, *system programs*, *operational*, and *system low*. The first two levels separate system code from user applications. There are still two integrity categories, *production* and *development*. Given the wider range of sources of data, more integrity levels are probably necessary today (e.g., Vista defines six integrity levels [152], although it is not a mandatory integrity system).

Biba is derived from Bell-LaPadula in such a way that it is also a mandatory protection system. The Biba protection state is mandatory, and, like Bell-LaPadula, its labeling state only permits the labeling to dominate integrity classes (i.e., lower in the Biba lattice) at creation time. Also, Biba has a null transition state, as integrity class assignments are fixed at creation time.

While the Biba model makes some intuitive sense, it differs from the BLP model in that there are no practical analogues for its use. The BLP model codifies the paper mechanism used by government agencies to protect the secrecy of documents. Thus, there was a practical application of the BLP model, so its application to computerized documents satisfied some users. We note that commercial systems do not use BLP in most cases.

A question is whether equating reading and writing with dependence in the Biba model is a practical way to enforce integrity. Many processes whose integrity must be protected, such as system services, databases, and web servers, accept input from untrusted sources. Biba explicitly prohibits such communication unless a formally-assured (see Chapter 12) *guard* process is inserted to filter such untrusted input. Since such guards must be application-specific, the development of such guards is expensive. Thus, the Biba model has not been applied to the extent that the BLP model has.

5.3.2 LOW-WATER MARK INTEGRITY

An alternative view of integrity is the Low-Water Mark integrity or LOMAC model [27, 101]. LOMAC differs from Biba in that the integrity of a subject or object is set equal to the lowest integrity class input. For example, a subject's integrity starts at the highest integrity class, but as code, libraries, and data are input, its integrity class drops to the lowest class of any of these inputs. Similarly, a file's integrity class is determined by the lowest integrity class of a subject that has written data to the file.

Example 5.13. Suppose a process p uses four inputs whose integrity levels are defined in the integrity lattice of Example 5.12 in Figure 5.4: (1) its code is at the `application` integrity level; (2) its libraries are also at the `system` integrity level; (3) its configuration files are at `application` integrity; and (4) it receives input from untrusted network subjects at the `low` integrity level. As a result, process p's integrity level will be `low`. If process p does not receive inputs from untrusted network subjects, then its integrity level will be `application`. However, if it is tricked into using a library that was provided by a untrusted network subject (i.e., the library is at the `low` integrity level), then process p will also be `low` protecting the integrity of application data.

LOMAC differs from BLP and Biba in that the integrity class of a subject or object may change as the system runs. That is, a LOMAC transition state is nonnull, as the protection state is not tranquil (see Section 5.2.2). LOMAC relaxes the tranquility requirement securely because it only lowers the security class of a subject or object. For a subject, the lowering of its security class reduces the set of objects that it can modify, reducing the number of objects whose integrity is dependent on the subject. For objects, the lowering of its security class reduces the number of subjects that can read the object, reducing the risk of subjects reading a low integrity object. Tranquility may be relaxed in other ways that still preserve the information flow requirements.

Like Biba, LOMAC does not correspond to how high integrity programs are built in practice, so its use has been limited. Since most high integrity processes receive some low integrity input, a LOMAC policy will result in most processes running at a low integrity class. In practice, systems depend on high integrity programs to protect themselves from malicious input. Despite the fact that there are many instances where this is not the case (e.g., buffer overflows in web server and network daemon software), most applications aim for self-protection.

5.3.3 CLARK-WILSON INTEGRITY

While some secure operating system architectures advocate the extensive use of guards (e.g., MILS, see Chapter 6), it is still an unresolved question whether the expensive of separate guards is justified. In the Clark-Wilson integrity model [54], no such external guard processes are required.

Ten years after the Biba model, Clark and Wilson aimed to bring integrity back into the focus of security enforcement. Clark-Wilson specified that high integrity data, called *constrained data items* (CDIs), must be validated as high integrity by special processes, called *integrity verification procedures* (IVPs), and could only be modified by high integrity processes, called *transformation procedures* (TPs). IVPs ensure that CDIs satisfy some known requirements for integrity (analogously to double-bookkeeping in accounting), so that the system can be sure that it starts with data that meets its integrity requirements. TPs are analogous to high integrity processes in Biba in that only they may modify high integrity data. That is, low integrity processes may not write data of a higher integrity level (i.e., CDI data). These two requirements are defined in two *certification rules* of the model, CR1 and CR2.

The Clark-Wilson model also includes *enforcement rules* that limit the users and TPs that may access CDIs, ER1 and ER2. When a CDI is accessed, it can be accessed only using a TP authorized for that CDI (ER1), and only by a user authorized to run that TP to access that CDI (ER2).

The Clark-Wilson model is comprehensive in that it defines rules for authentication, auditing, and administration. Clark-Wilson requires that all users must be authenticated before they can run a TP (ER3). Also, auditing is enforced by the rule that states that all TPs must append operational information sufficient to reconstruct any operation in an append-only CDI (CR4). Administration is enforced via two rules. First, Clark-Wilson restricts the certifier of a TP to be an entity who does not have execute permission for that TP (ER4). Second, administration of permissions assigning users and TPs to CDIs must satisfy the principle of *separation of duty*, defined by Clark and Wilson. According to this principle, no single principal can execute all the transactions in a separation transaction set. For example, a subject cannot both execute a payment and authorize that payment.

However, the most significant part of the model was the portion concerned with integrity protection. The Clark-Wilson model also includes a rule (CR5) to prevent the high integrity processes (IVPs and TPs) from depending on low integrity data. A high integrity process may read low integrity data, called *unconstrained data items* (UDIs), but it must either *upgrade or discard* that data upon receipt. That is, Clark-Wilson does not require a separate guard process like Biba, but it requires that the high integrity process guard itself. In order for a high integrity process to justify its ability to both write CDIs correctly and protect itself when it reads UDIs, the Clark-Wilson model requires that TPs be fully assured themselves. As we have discuss in Chapter 12, methodologies for comprehensive assurance are expensive, so few applications have ever been assured at this level.

5.3.4 THE CHALLENGE OF TRUSTED PROCESSES

With MLS and Biba, we can formally show that a system's information flows adhere to these policies, but they assume that there are no processes that would ever require illegal information flows. For MLS, some processes may be required to leak some information processed at a secret clearance to processes that can only read public information. This is analogous to a general holding a news conference or submitting orders to subordinates. For Biba, some trusted processes may be required to process inputs received from untrusted processes. This is analogous to an oracle providing responses to questions. The integrity of the oracle cannot be impacted by the integrity of the questions.

Such computations must be implemented by trusted processes. In Multics, it was found [333] that system processes could "not operate at a single clearance level," so these processes must be trusted "never [to] violate the fundamental security rules." In the design of KSOS, the set of trusted user-level programs including 10 different groups of programs [97].

While MLS and Biba models support the use of trusted processes (e.g., guards), the number of programs that need to be trusted became nontrivial and remain so today. For example, SELinux/MLS (see Chapter 9) specifies over 30 trusted subjects. One approach has been to reduce the scope of trust required of such programs. GEMSOS models permitted processes to work within ranges of access classes [277, 290]. The Caernarvon model [278] permits processes to work within access class ranges encompassing both secrecy and integrity. To obtain such authorization, programs must be certified in some manner analogous to Common Criteria assurance.

In lieu of formal assurance models, other approaches for building and using trusted processes have emerged. For example, *security-typed languages* ensure that compiled programs satisfy associated secrecy and integrity requirements [291, 219]. Such languages depend on trust in compilers and runtime environments that are not presently assured. Nonetheless, the idea of leveraging language level guarantees is gaining momentum.

Presuming that a basis for trust in programs emerges, a variety of recent work is already exploring how to manage such trust. A number of researchers have reverted to information flow as the ideal basis for security, and the problem is now to define the parameters of trust over information flow.

Some recent models leverage the concepts of the *decentralized label model* (DLM) [218] used in security-typed languages at the system-level (i.e., per process) to ensure information flow and define trust in processes. The key features of the DLM model are its representation of both secrecy and integrity in information flow policies, and its inclusion of specifications of trust in downgrading secret information (*declassifiers*) and upgrading low integrity information (*endorsers*). The Asbestos label model [316] expresses both secrecy and integrity requirements, and provides a special level value $\star$ that designates trust, but this trust can be limited to specific categories of data. Subsequent generalization of the Asbestos model into the *decentralized information flow control* (DIFC) model even more closely resembles the DLM model [174, 349, 350]. Processes may change the label of data under the limitations of information flow, unless the process has a privilege that permits it to downgrade or upgrade per specific labels. For example, a privilege t^- permits a holding process to remove the label t from its data, thus permitting it to downgrade such data.

Other recent integrity models define how processes may manage low integrity data, yet be trusted to maintain their high integrity [182, 285, 300]. For example, CW-Lite [285] leverages the semantics of the Clark-Wilson where a high integrity process may receive low integrity data if it immediately discarded or upgrades such data. In CW-Lite, trusted, filtering interfaces are identified as the only locations that such discard/upgrade is permitted for a process. Only such interfaces must be trusted. Usable Mandatory Integrity Protection (UMIP) [182] has a similar approach, but it defines trust in terms of the types of information flows (e.g., network, IPC) that a process may be entrusted. Finally, Practical Proactive Integrity (PPI) [300] permits definition of all previous integrity model semantics into policy options that enable flexible definition of trust in a process's handling of low integrity data.

5.4 COVERT CHANNELS

Lampson identified the problem that systems contain a variety of implicit communication channels enabled by access to shared physical resources [177, 189]. For example, if two processes share access to a disk device, they can communicate by testing the state of the device (i.e., whether it is full or not). These channels are now called *covert channels* because they are not traditionally intended for communication. Such channels are present in most system hardware (e.g., keyboards [284], disks [160], etc.) and across the network (e.g., [110, 294, 338]). Millen has provided a summary of covert channels in multilevel security systems [212].

From a security perspective, the problem is that these communication mechanisms are outside the control of the reference monitor. Assume a BLP policy. When a secret process writes to a secret file, this is permitted by BLP. When a unclassified process writes to an unclassified file this is also permitted. However, if the two files are stored on the same disk device, when the secret process fills the disk (e.g., via a Trojan horse) the unclassified process can see this. Thus, a covert communication is possible. Different covert channels have different data rates, so the capacity of the channel is an important issue in determining the vulnerability of the system [211, 215].

5.4.1 CHANNEL TYPES

Covert channels are classified into two types: storage and timing channels. A *storage covert channel* requires the two communicating processes to have access to a shared physical resource that may be modified by the sending party and viewed by the receiving party. For example, the shared disk device is an example of a storage channel because a process may modify the disk device by adding data to the disk. This action is observable by another party with access to this disk because the disk contents may be consumed. For example, a full disk may signal the binary value of one, and an available disk may signal the binary value of zero. Uses of the disk as a storage channel have been identified [160]. Additionally, the communicating parties also need a synchronized clock in order to know when to test the disk. For example, how do we differentiate between the transmission of a single "one" and two consecutive "ones." Of course, a system has many forms of time keeping available, such as dates, cycle counters, etc. As a result, storage covert channels are quite practical for the attacker [315].

A *timing covert channel* requires that the communicating party be able to affect the timing behavior of a resource [341]. For example, if a communicating process has high priority to a network device, then it can communicate by using the device for a certain among of time. For example, if the communicating process is using the device, then it is transmitting a binary "one," otherwise it is transmitting a binary "zero." Reliable timing channels are a bit harder to find because often there are many processes that may affect the timing of a device, and the scheduling of the device may not always be so advantageous to communication. In addition to the source whose timing behavior may be affected, timing channels also require synchronized clock.

Covert channels are problematic because they may be built into a system unknowingly, and they may be difficult to eradicate completely. A storage channel is created whenever a shared resource of limited size is introduced to the system. This may be a storage device, or it may be artificially created. For example, limiting the number of sockets that a system may have open at one time would introduce a storage channel. The challenge is to identify the presence of such channels. Once they are found, they are easily removed by prohibiting the processes from using the same resource.

Initially, the techniques to identify covert channels in software was ad hoc, but researchers developed systematic techniques to enable the discovery of covert channels. First, Kemmerer defined the Shared Resource Matrix [162, 163] (SRM) where shared resources and the operations that may access these resources are identified. A matrix is then constructed that shows how the resources are accessed by the operations. Given the matrix, we can determine whether a high process can use the resource and operations to leak data to the low process. The SRM technique is a manual technique for reviewing source code, but later techniques analyzed source code directly [40, 164]. For example, Kemmerer later defined Covert Flow Trees [164] (CFT) in which trees represent the flow of information in a program that serves shared resources, such as a file system. See Bishop [29] for a more detailed description of these approaches as well.

While it is at least theoretically possible to remove all covert storage channels from a system, timing channels cannot be completely removed. This is because the timing behavior of a system is available to all processes. For example, a process can cause a longer page fault handling time by consuming a large amount of pages [4]. Techniques to address timing channels, such as *fuzzy time* [140, 311], involve reducing the bandwidth of channels by randomizing their timing behavior. In a fuzzy time system, the response time of an operation is modified to prevent the operation's process from communicating. In theory, the time of every operation must be the same in order to prevent any communication. Thus, operations must be delayed in order to hide their performance (i.e., slow operations cannot be made faster), thus effecting the overall performance of the system. Other techniques to address timing channels have similar negative performance effects [160].

A question is whether BLP enforcement could be extended to control covert channels. If an *unclassified* process stores files on a disk, then the disk must be assigned the *unclassified* access class because the unclassified process can "read" the state of the disk. Thus, we would not store secret files on the disk, so a Trojan horse running in a *secret* process could not write to this disk. Unfortunately, this analogy cannot be carried out completely, as some resources, such as CPU, must be shared, and the cost of having a device per access class is sometimes impractical (e.g., one network device per access class). As a result, the covert channel problem is considered as an implementation issue that requires analysis of the system [249].

5.4.2 NONINTERFERENCE

An alternative for controlling covert channels is to use models that express the input-output requirements of a system. These models are based on the notion of *noninterference* [113]. Intuitively, noninterference among processes requires that the actions of any process have no effect on what any other process sees. For example, the actions of one process writing to a file should not be seen by any other process should the system enforce noninterference between them.

More formally, noninterference has been assessed as follows [113, 202, 203]. Consider a system in which the output of user u is given by the function $out(u, hist.read(u))$ where $hist.read(u)$ is the trace of inputs to u and $read(u)$ was the latest input. Noninterference is defined based on what part of a trace can be purged from the input of other users u' (and their processes) whose security class $SC(u')$ is dominated by $SC(u)$.

Definition 5.14. Let $purge$ be a function from $users \times traces$ to $traces$ where $purge(u', hist.command(u)) = purge(u'.hist)$ if $SC(u') < SC(u)$.

That is, $purge$ ensures that lower secrecy subjects are not impacted by the commands run by higher secrecy subjects.

In general, noninterference is not comparable to BLP. Since BLP does not prevent covert channels, BLP enforcement is weaker than noninterefeference. However, noninterference does not prevent the inputs of lower secrecy subjects from impacting higher secrecy subjects, implying that lower secrecy subjects may learn the state of higher secrecy data (i.e., illegally read up). In addition,

[4]This attack has a combination of storage and timing issues. The size of memory is finite which implies a storage channel, but the measurement of value involve the timing of the memory access.

noninterference traces do not address problems caused by timing channels, as such channels are not represented in the traces. On the other hand, noninterference prevents the more secret subject from passing encrypted data to the secret subject [302], so noninterference, like the other security properties that we have discussed, is also a more conservative approximation of the security than necessary, in practice. There is a large body of formal modeling of noninterference and assessment of its properties [70, 203, 218, 261, 144, 55, 91, 246], but this beyond the scope of this book. In practice, noninterefeence properties are too complex for manual analysis, and while research applications have had some successes [33, 125], support for enforcing noninterference requirements, in general, has not been developed.

5.5 SUMMARY

In this chapter, we examined the policy models that have been used in mandatory protection systems. Such policies must provide mandatory protection states that define the desired secrecy and integrity protection required, labeling states that securely assign system processes and resources to security classes, and transition states that ensure that any change in a process or resource's security class is also secure. These models define security goals, labeling, and transitions in terms of information flow. Information flow describes how any data in the system could flow from one subject to another. Information flow is conservative because it shows the possible flow paths, but these may or may not actually be used. The information flow abstraction works reasonably well for secrecy, at least for the government sector where a version of information flow was employed prior to computerization. However, information flow has not been practical for integrity, as the abstraction is too conservative.

Secrecy is defined in terms of a finite lattice of security classes where the security class determines who is authorized to access the data. It turns out that the more secret the data, the fewer the subjects that can read it. This has been codified in the Bell-LaPadula (BLP) model. Only subjects whose security class dominates or is equal to that of the data may read it (*simple-security property*). However, in order to prevent leakage via malware, subjects can only write to objects whose security class dominates that of the subject (⋆*-security property*).

For integrity, the information flow requirements are reversed. This is because we are concerned about high integrity processes reading less trusted data. Thus, the *simple-integrity* and ⋆*-integrity* properties are defined with reverse flows. The Biba model defines these requirements. Since it was found that integrity was largely independent of secrecy, a second set of security classes is generally used. In practice, information flow has been too conservative an approximation for integrity because data input does not imply *dependence*. Few secure operating systems use formal integrity management at present, despite the many vulnerabilities that result from the improper handling of malicious input.

Finally, we examined the problem of covert information flows. In these, access to shared resources is used to convey information. If a subject can modify a shared resource, the others that use this resource can receive this "signal." In theory, we cannot prevent two processes running on the same system from using a covert channel availed by the system, but we can reduce the bitrate. The *noninterference* property formalizes information flow in terms of input and output, such that a covert channel must convey some input that affects the behavior of an unauthorized subject. However, practical application of noninterference is complex and supporting tools are not available.