CHAPTER 2

# Access Control Fundamentals

An *access enforcement mechanism* authorizes requests (e.g., system calls) from multiple *subjects* (e.g., users, processes, etc.) to perform *operations* (e.g., read, write, etc.) on objects (e.g., files, sockets, etc.). An operating system provides an access enforcement mechanism. In this chapter, we define the fundamental concepts of access control: a *protection system* that defines the access control specification and a *reference monitor* that is the system's access enforcement mechanism that enforces this specification. Based on these concepts, we provide an ideal definition for a secure operating system. We use that definition to evaluate the operating systems security of the various systems examined in this book.

## 2.1 PROTECTION SYSTEM

The security requirements of a operating system are defined in its *protection system*.

**Definition 2.1.** A *protection system* consists of a *protection state*, which describes the operations that system subjects can perform on system objects, and a set of *protection state operations*, which enable modification of that state.

A protection system enables the definition and management of a protection state. A *protection state* consists of the specific system subjects, the specific system objects, and the operations that those subjects can perform on those objects. A protection system also defines *protection state operations* that enable a protection state to be modified. For example, protection state operations are necessary to add new system subjects or new system objects to the protection state.

### 2.1.1 LAMPSON'S ACCESS MATRIX

Lampson defined the idea that a protection state is represented by an *access matrix*, in general, [176].

**Definition 2.2.** An *access matrix* consists of a set of subjects $s \in S$, a set of objects $o \in O$, a set of operations $op \in OP$, and a function $ops(s, o) \subseteq OP$, which determines the operations that subject $s$ can perform on object $o$. The function $ops(s, o)$ is said to return a set of operations corresponding to cell $(s, o)$.

Figure 2.1 shows an access matrix. The matrix is a two-dimensional representation where the set of subjects form one axis and the set of objects for the other axis. The cells of the access matrix store the operations that the corresponding subject can perform on the corresponding object. For example, subject `Process 1` can perform `read` and `write` operations on object `File 2`.

| | File 1 | File 2 | File 3 | Process 1 | Process 2 |
|---|---|---|---|---|---|
| Process 1 | Read | Read, Write | Read, Write | Read | - |
| Process 2 | - | Read | Read, Write | - | Read |

**Figure 2.1:** Lampson's Access Matrix

If the subjects correspond to processes and the objects correspond to files, then we need protection state operations to update the protection state as new files and processes are created. For example, when a new file is created, at least the creating process should gain access to the file. In this case, a protection state operation `create_file(process, file)` would add a new column for the new file and add `read` and `write` operations to the cell $(process, file)$.

Lampson's access matrix model also defines operations that determine which subjects can modify cells. For example, Lampson defined an `own` operation that defines ownership operations for the associated object. When a subject is permitted for the `own` operation for an object $o$, that subject can modify the other cells associated with that object $o$. Lampson also explored delegation of ownership operations to other subjects, so others may manage the distribution of permissions.

The access matrix is used to define the *protection domain* of a process.

**Definition 2.3.** A *protection domain* specifies the set of resources (objects) that a process can access and the operations that the process may use to access such resources.

By examining the rows in the access matrix, one can see all the operations that a subject is authorized to perform on system resources. This determines what information could be read and modified by a processes running on behalf of that subject. For a secure operating system, we will want to ensure that the protection domain of each process satisfies system security goals (e.g., secrecy and integrity).

A process at any time is associated with one or more subjects that define its protection domain. That is, the operations that it is authorized to perform are specified by one or more subjects. Systems that we use today, see Chapter 4, compose protection domains from a combination of subjects, including users, their groups, aliases, and ad hoc permissions. However, protection domains can also be constructed from an intersection of the associated subjects (e.g., Windows 2000 Restricted Contexts [303]). The reason to use an intersection of subjects' permissions is to restrict the protection domain to permissions shared by all, rather than giving the protection domain subjects extra permissions that they would not normally possess.

Because the access matrix would be a sparse data structure in practice (i.e., most of the cells would not have any operations), other representations of protection states are used in practice. One representation stores the protection state using individual object columns, describing which subjects have access to a particular object. This representation is called an *access control list* or ACL. The other representation stores the other dimension of the access matrix, the subject rows. In this case, the objects that a particular subject can access are stored. This representation is called a *capability list* or C-List.

There are advantages and disadvantages to both the C-List and ACL representations of protection states. For the ACL approach, the set of subjects and the operations that they can perform are stored with the objects, making it easy to tell which subjects can access an object at any time. Administration of permissions seems to be more intuitive, although we are not aware of any studies to this effect. C-Lists store the set of objects and operations that can be performed on them are stored with the subject, making it easy to identify a process's protection domain. The systems in use today, see Chapter 4, use ACL representations, but there are several systems that use C-Lists, as described in Chapter 10.

### 2.1.2 MANDATORY PROTECTION SYSTEMS

This access matrix model presents a problem for secure systems: untrusted processes can tamper with the protection system. Using protection state operations, untrusted user processes can modify the access matrix by adding new subjects, objects, or operations assigned to cells. Consider Figure 2.1. Suppose `Process 1` has ownership over `File 1`. It can then grant any other process read or write (or potentially even ownership) access over `File 1`. A protection system that permits untrusted processes to modify the protection state is called a *discretionary access control* (DAC) system. This is because the protection state is at the discretion of the users and any untrusted processes that they may execute.

The problem of ensuring that particular protection state and all possible future protection states derivable from this state will not provide an unauthorized access is called the *safety problem* [130] [1]. It was found that this problem is undecidable for protection systems with compound protection state operations, such as for `create_file` above which both adds a file column and adds the operations to the owner's cell. As a result, it is not possible, in general, to verify that a protection state in such a system will be secure (i.e., satisfy security goals) in the future. To a secure operating system designer, such a protection system cannot be used because it is not tamperproof; an untrusted process can modify the protection state, and hence the security goals, enforced by the system.

We say that the protection system defined in Definition 2.1 aims to enforce the requirement of *protection*: one process is protected from the operations of another only if both processes behave benignly. If no user process is malicious, then with some degree of certainly, the protection state will still describe the true security goals of the system, even after several operations have modified the protection state. Suppose that a `File 1` in Figure 2.1 stores a secret value, such as a private key in a public key pair [257], and `File 2` stores a high integrity value like the corresponding public key. If `Process 1` is non-malicious, then it is unlikely that it will leak the private key to `Process 2` through either `File 1` or File 2 or by changing the `Process 2`'s permissions to `File 1`. However, if `Process 1` is malicious, it is quite likely that the private key will be leaked. To ensure that the

[1]For a detailed analysis of the *safety problem* see Bishop's textbook [29].

secrecy of `File 1` is enforced, all processes that have access to that file must not be able to leak the file through the permissions available to that process, including via protection state operations.

Similarly, the access matrix protection system does not ensure the integrity of the public key file `File 2`, either. In general, an attacker must not be able to modify any user's public key because this could enable the attacker to replace this public key with one whose private key is known to the attacker. Then, the attacker could masquerade as the user to others. Thus, the integrity compromise of `File 2` also could have security ramifications. Clearly, the access matrix protection system cannot protect `File 2` from a malicious `Process 1`, as it has write access to `File 2`. Further, a malicious `Process 2` could enhance this attack by enabling the attacker to provide a particular value for the public key. Also, even if `Process 1` is not malicious, a malicious `Process 2` may be able to trick `Process 1` into modifying `File 2` in a malicious way depending on the interface and possible vulnerabilities in `Process 1`. Buffer overflow vulnerabilities are used in this manner for a malicious process (e.g., `Process 2`) to take over a vulnerable process (e.g., `Process 1`) and use its permissions in an unauthorized manner.

Unfortunately, the protection approach underlying the access matrix protection state is naive in today's world of malware and connectivity to ubiquitous network attackers. We see in Chapter 4 that today's computing systems are based on this protection approach, so they cannot be ensure enforcement of secrecy and integrity requirements. Protection systems that can enforce secrecy and integrity goals must enforce the requirement of *security*: *where a system's security mechanisms can enforce system security goals even when any of the software outside the trusted computing base may be malicious.* In such a system, the protection state must be defined based on the accurate identification of the secrecy and integrity of user data and processes, and no untrusted processes may be allowed to perform protection state operations. Thus, the dependence on potentially malicious software is removed, and a concrete basis for the enforcement of secrecy and integrity requirements is possible.

This motivates the definition of a mandatory protection system below.

**Definition 2.4.** A *mandatory protection system* is a protection system that can only be modified by trusted administrators via trusted software, consisting of the following state representations:

- A *mandatory protection state* is a protection state where subjects and objects are represented by *labels* where the state describes the operations that subject labels may take upon object labels;
- A *labeling state* for mapping processes and system resource objects to labels;
- A *transition state* that describes the legal ways that processes and system resource objects may be relabeled.

For secure operating systems, the subjects and objects in an access matrix are represented by system-defined *labels*. A label is simply an abstract identifier—the assignment of permissions to a label defines its security semantics. Labels are tamperproof because: (1) the set of labels is defined by trusted administrators using trusted software and (2) the set of labels is immutable. Trusted administrators define the access matrix's labels and set the operations that subjects of particular labels can perform on objects of particular labels. Such protection systems are *mandatory access control* (MAC) systems because the protection system is immutable to untrusted processes [2]. Since the set of labels cannot be changed by the execution of user processes, we can prove the security goals enforced by the access matrix and rely on these goals being enforced throughout the system's execution.

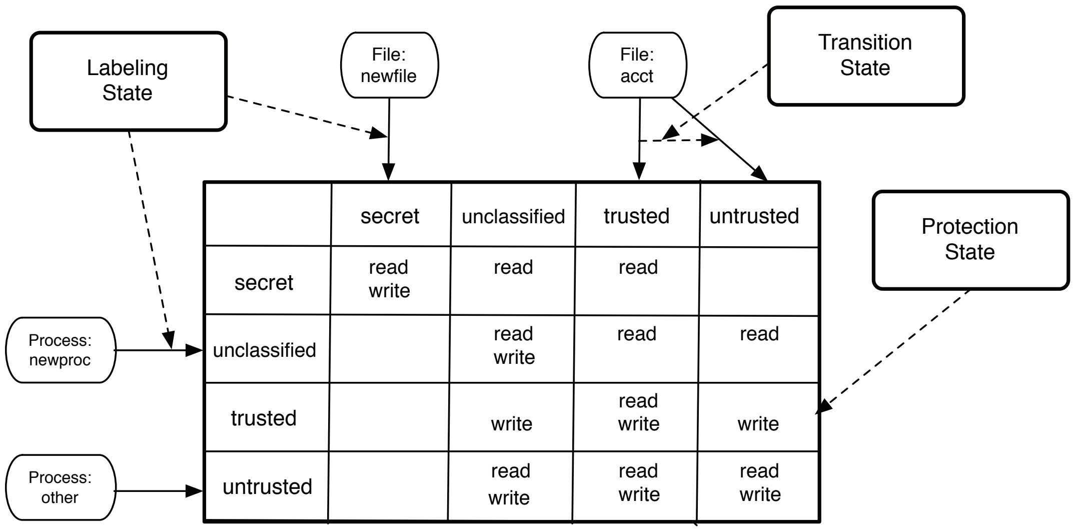
Of course, just because the set of labels are fixed does not mean that the set of processes and files are fixed. Secure operating systems must be able to attach labels to dynamically created subjects and objects and even enable label transitions.

A *labeling state* assigns labels to new subjects and objects. Figure 2.2 shows that processes and files are associated with labels in a fixed protection state. When `newfile` is created, it must be assigned one of the object labels in the protection state. In Figure 2.2, it is assigned the `secret` label. Likewise, the process `newproc` is also labeled as `unclassified`. Since the access matrix does not permit `unclassified` subjects with access to `secret` objects, `newproc` cannot access `newfile`. As for the protection state, in a secure operating system, the labeling state must be defined by trusted administrators and immutable during system execution.

A *transition state* enables a secure operating system to change the label of a process or a system resource. For a process, a label transition changes the permissions available to the process (i.e., its protection domain), so such transitions are called *protection domain transitions* for processes. As an example where a protection domain transition may be necessary, consider when a process executes a different program. When a process performs an `execve` system call the process image (i.e., code and data) of the program is replaced with that of the file being executed. Since a different program is run as a result of the `execve` system call, the label associated with that process may need to be changed as well to indicate the requisite permissions or trust in the new image.

A transition state may also change the label of a system resource. A label transition for a file (i.e., object or resource) changes the accessibility of the file to protection domains. For example, consider the file `acct` that is labeled `trusted` in Figure 2.2. If this file is modified by a process with an `untrusted` label, such as `other`, a transition state may change its label to `untrusted` as well. The Low-Water Mark (LOMAC) policy defines such kind of transitions [101, 27] (see Chapter 5). An alternative would be to change the protection state to prohibit `untrusted` processes from modifying `trusted` files, which is the case for other policies. As for the protection state and labeling state, in a secure operating system, the transition state must be defined by trusted administrators and immutable during system execution.

[2] Historically, the term *mandatory access control* has been used to define a particular family of access control models, lattice-based access control models [271]. Our use of the terms *mandatory protection system* and *mandatory access control system* are meant to include historical MAC models, but our definition aims to be more general. We intend that these terms imply models whose sets of labels are immutable, including these MAC models and others, which are administered only by trusted subjects, including trusted software and administrators. We discuss the types of access control models that have been used in MAC systems in Chapter 5.



| | secret | unclassified | trusted | untrusted |
|---|---|---|---|---|
| secret | read write | read | read | |
| unclassified | | read write | read | read |
| trusted | | write | read write | write |
| untrusted | | read write | read write | read write |

**Figure 2.2:** A Mandatory Protection System: The *protection state* is defined in terms of labels and is immutable. The immutable *labeling state* and *transition state* enable the definition and management of labels for system subjects and objects.

## 2.2 REFERENCE MONITOR

A *reference monitor* is the classical access enforcement mechanism [11]. Figure 2.3 presents a generalized view of a reference monitor. It takes a request as input, and returns a binary response indicating whether the request is *authorized* by the reference monitor's access control policy. We identify three distinct components of a reference monitor: (1) its interface; (2) its authorization module; and (3) its policy store. The interface defines where the authorization module needs to be invoked to perform an authorization query to the protection state, a labeling query to the labeling state, or a transition query to the transition state. The authorization module determines the exact queries that are to be made to the policy store. The policy store responds to authorization, labeling, and transition queries based on the protection system that it maintains.

*Reference Monitor Interface* The reference monitor interface defines where protection system queries are made to the reference monitor. In particular, it ensures that all security-sensitive operations are authorized by the access enforcement mechanism. By a *security-sensitive operation*, we mean an *operation* on a particular *object* (e.g., file, socket, etc.) whose execution may violate the system's security requirements. For example, an operating system implements file access operations that would allow one user to read another's secret data (e.g., private key) if not controlled by the operating system. Labeling and transitions may be executed for authorized operations.

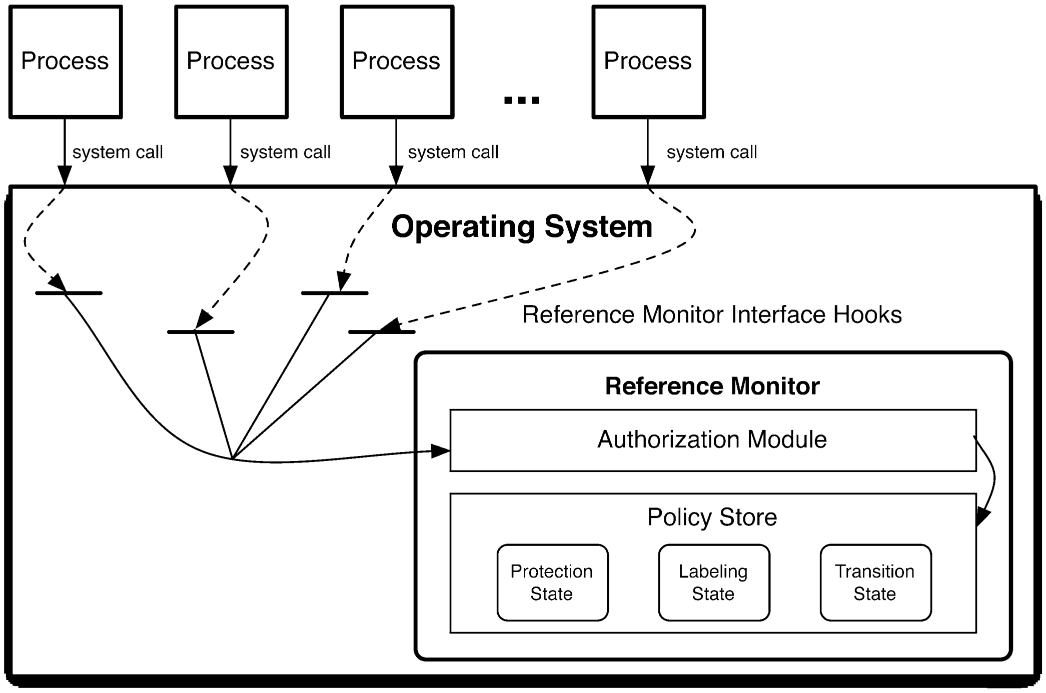


**Figure 2.3:** A *reference monitor* is a component that authorizes access requests at the *reference monitor interface* defined by individual *hooks* that invoke the reference monitor's *authorization module* to submit an authorization query to the *policy store*. The policy store answers authorization queries, labeling queries, and label transition queries using the corresponding states.

The reference monitor interface determines where access enforcement is necessary and the information that the reference monitor needs to authorize that request. In a traditional UNIX file `open` request, the calling process passes a file path and a set of operations. The reference monitor interface must determine what to authorize (e.g., directory searches, link traversals, and finally the operations for the target file's `inode`), where to perform such authorizations (e.g., authorize a directory search for each directory inode in the file path), and what information to pass to the reference monitor to authorize the open (e.g., an `inode` reference). Incorrect interface design may allow an unauthorized process to gain access to a file.

*Authorization Module* The core of the reference monitor is its authorization module. The authorization module takes interface's inputs (e.g., process identity, object references, and system call name), and converts these to a query for the reference monitor's policy store. The challenge for the authorization module is to map the process identity to a subject label, the object references to an object label, and determine the actual operations to authorize (e.g., there may be multiple operations per interface). The protection system determines the choices of labels and operations, but the authorization module must develop a means for performing the mapping to execute the "right" query.

For the `open` request above, the module responds to the individual authorization requests from the interface separately. For example, when a directory in the file path is requested, the authorization module builds an authorization query. The module must obtain the label of the subject responsible for the request (i.e., requesting process), the label of the specified directory object (i.e., the directory `inode`), and the protection state operations implied the request (e.g., read or search the directory). In some cases, if the request is authorized by the policy store, the module may make subsequent requests to the policy store for labeling (i.e., if a new object were created) or label transitions.

*Policy Store* The policy store is a database for the protection state, labeling state, and transition state. An authorization query from the authorization module is answered by the policy store. These queries are of the form `{subject_label, object_label, operation_set}` and return a binary authorization reply. Labeling queries are of the form `{subject_label, resource}` where the combination of the subject and, optionally, some system resource attributes determine the resultant resource label returned by the query. For transitions, queries include the `{subject_label, object_label, operation, resource}`, where the policy store determines the resultant label of the resource. The resource may be either be an active entity (e.g., a process) or a passive object (e.g., a file). Some systems also execute queries to authorize transitions as well.

## 2.3 SECURE OPERATING SYSTEM DEFINITION

We define a *secure operating system* as a system with a reference monitor access enforcement mechanism that satisfies the requirements below when it enforces a mandatory protection system.

**Definition 2.5.** A *secure operating system* is an operating system where its access enforcement satisfies the *reference monitor concept* [11].

**Definition 2.6.** The *reference monitor concept* defines the necessary and sufficient properties of any system that securely enforces a mandatory protection system, consisting of three guarantees:

1. **Complete Mediation**: The system ensures that its access enforcement mechanism mediates all security-sensitive operations.

2. **Tamperproof**: The system ensures that its access enforcement mechanism, including its protection system, cannot be modified by untrusted processes.

3. **Verifiable**: The access enforcement mechanism, including its protection system, "must be small enough to be subject to analysis and tests, the completeness of which can be assured" [11]. That is, we must be able to prove that the system enforces its security goals correctly.

The reference monitor concept defines the *necessary* and *sufficient* requirements for access control in a secure operating system [145]. First, a secure operating system must provide complete mediation of all security-sensitive operations. If all these operations are not mediated, then a security requirement may not be enforced (i.e., a secret may be leaked or trusted data may be modified by an untrusted process). Second, the reference monitor system, which includes its implementation and the protection system, must all be tamperproof. Otherwise, an attacker could modify the enforcement function of the system, again circumventing its security. Finally, the reference monitor system, which includes its implementation and the protection system, must be small enough to verify the correct enforcement of system security goals. Otherwise, there may be errors in the implementation or the security policies that may result in vulnerabilities.

A challenge for the designer of secure operating system is how to precisely achieve these requirements.

*Complete Mediation* Complete mediation of security-sensitive operations requires that all program paths that lead to a security-sensitive operation be mediated by the reference monitor interface. The trivial approach is to mediate all system calls, as these are the entry points from user-level processes. While this would indeed mediate all operations, it is often insufficient. For example, some system calls implement multiple distinct operations. The `open` system call involves opening a set of directory objects, and perhaps file links, before reaching the target file. The subject may have different permission for each of these objects, so several, different authorization queries would be necessary. Also, the directory, link, and file objects are not available at the system call interface, so the interface would have to compute them, which would result in redundant processing (i.e., since the operating system already maps file names to such objects). But worst of all, the mapping between the file name passed into an `open` system call and the directory, link, and file objects may be changed between the start of the system call and the actual open operation (i.e., by a well-timed rename operation). This is called a *time-of-check-to-time-of-use* (TOCTTOU) attack [30], and is inherent to the `open` system call.

As a result, reference monitors require interfaces that are embedded in the operating system itself in order to enforce complete mediation correctly. For example, the Linux Security Modules (LSM) framework [342] (see Chapter 9), which defines the mediation interface for reference monitors in Linux does not authorize the `open` system call, but rather each individual directory, link, and file open after the system object reference (i.e., the `inode`) has been retrieved. For LSM, tools have been built to find bugs in the complete mediation demanded of the interface [351, 149], but it is difficult to verify that a reference monitor interface is correct.

*Tamperproof* Verifying that a reference monitor is tamperproof requires verifying that all the reference monitor components, the reference monitor interface, authorization module, and policy store, cannot be modified by processes outside the system's *trusted computing base* (TCB) (see Chapter 1). This also implies that the TCB itself is high integrity, so we ultimately must verify that the entire TCB cannot be modified by processes outside the TCB. Thus, we must identify all the ways that the TCB can be modified, and verify that no untrusted processes (i.e., those outside the TCB) can perform such modifications. First, this involves verifying that the TCB binaries and data files are unmodified. This can be accomplished by a multiple means, such as file system protections and binary verification programs. Note that the verification programs themselves (e.g., Tripwire [169]) must also be protected. Second, the running TCB processes must be protected from modification by untrusted processes. Again, system access control policy may ensure that untrusted processes cannot communicate with TCB processes, but for TCB processes that may accept inputs from untrusted processes, they must protect themselves from malicious inputs, such as buffer overflows [232, 318], format string attacks [305], and return-to-libc [337]. While defenses for runtime vulnerabilities are fundamental to building tamperproof code, we do not focus on these software engineering defenses in this book. Some buffer overflow defenses, such as StackGuard [64] and stack randomization [121], are now standard in compilers and operating systems, respectively.

Second, the policy store contains the mandatory protection system which is a MAC system. That is, only trusted administrators are allowed to modify its states. Unfortunately, access control policy is deployment-specific, so administrators often will need to modify these states. While administrators may be trusted they may also use untrusted software (e.g., their favorite editor). The system permissions must ensure that no untrusted software is used to modify the mandatory protection system.

Tamperproofing will add a variety of specific security requirements to the system. These requirements must be included in the verification below.

*Verifiable* Finally, we must be able to verify that a reference monitor and its policy really enforce the system security goals. This requires verifying the correctness of the interface, module, and policy store software, and evaluating whether the mandatory protection system truly enforces the intended goals. First, verifying the correctness of software automatically is an unsolved problem. Tools have been developed that enable proofs of correctness for small amounts of code and limited properties (e.g., [18]), but the problem of verifying a large set of correctness properties for large codebases appears intractable. In practice, correctness is evaluated with a combination of formal and manual techniques which adds significant cost and time to development. As a result, few systems have been developed with the aim of proving correctness, and any comprehensive correctness claims are based on some informal analysis (i.e., they have some risk of being wrong).

Second, testing that the mandatory protection system truly enforces the intended security goals appears tractable, but in practice, the complexity of systems makes the task difficult. Because the protection, labeling, and transition states are immutable, the security of these states can be assessed.

For protection states, some policy models, such as Bell-LaPadula [23] and Biba [27], specify security goals directly (see Chapter 5), but these are idealizations of practical systems. In practice, a variety processes are trusted to behave correctly, expanding the TCB yet further, and introducing risk that the security goals cannot be enforced. For operating systems that have fine-grained access control models (i.e., lots of unique subjects and objects), specifying and verifying that the policy enforces the intended security goals is also possible, although the task is significantly more complex.

For the labeling and transition states, we must consider the security impact of the changes that these states enable. For example, any labeling state must ensure that any label associated with a system resource does not enable the leakage of data or the modification of unauthorized data. For example, if a `secret` process is allowed to create `public` objects (i.e., those readable by any process), then data may be leaked. The labeling of some objects, such as data imported from external media, presents risk of incorrect labeling as well.

Likewise, transition states must ensure that the security goals of the system are upheld as processes and resources are relabeled. A challenge is that transition states are designed to enable *privilege escalation*. For example, when a user wants to update their password, they use an unprivileged process (e.g., a shell) to invoke privileged code (e.g., the `passwd` program) to be run with the privileged code's label (e.g., UNIX `root` which provides full system access). However, such transitions may be insecure if the unprivileged process can control the execution of the privileged code. For example, unprivileged processes may be able to control a variety of inputs to privileged programs, including libraries, environment variables, and input arguments. Thus, to verify that the system's security goals are enforced by the protection system, we must examine more than just the protection system's states.

## 2.4 ASSESSMENT CRITERIA

For each system that we examine, we must specify precisely how each system enforces the reference monitor guarantees in order to determine how an operating system aims to satisfy these guarantees. In doing this, it turns out to be easy to expose an insecure operating system, but it is difficult to define how close to "secure" an operating system is. Based on the analysis of reference monitor guarantees above, we list a set of dimensions that we use to evaluate the extent to which an operating system satisfies these reference monitor guarantees.

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?

   In this answer, we describe how the system ensures that the subjects, objects, and operations being mediated are the ones that will be used in the security-sensitive operation. This can be a problem for some approaches (e.g., *system call interposition* [3, 6, 44, 84, 102, 115, 171, 250]), in which the reference monitor does not have access to the objects used by the operating system. In some of these cases, a race condition may enable an attacker to cause a different object to be accessed than the one authorized by reference monitor [30].

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

   We describe how the mediation interface described above mediates all security-sensitive operations.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

   We describe any formal means for verifying the complete mediation described above.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, from modification?

   In modern systems, the reference monitor and its protection system are protected by the operating system in which they run. The operating system must ensure that the reference monitor cannot be modified and the protection state can only be modified by trusted computing base processes.

5. **Tamperproof**: Does the system's protection system protect the trusted computing base programs?

   The reference monitor's tamperproofing depends on the integrity of the entire trusted computing base, so we examine how the trusted computing base is defined and protected.

6. **Verifiable**: What is basis for the correctness of the system's trusted computing base?

   We outline the approach that is used to justify the correctness of the implementation of all trusted computing base code.

7. **Verifiable**: Does the protection system enforce the system's security goals?

   Finally, we examine how the system's policy correctly justifies the enforcement of the system's security goals. The security goals should be based on the models in Chapter 5, such that it is possible to test the access control policy formally.

While this is undoubtedly an incomplete list of questions to assess the security of a system, we aim to provide some insight into why some operating systems cannot be secure and provide some means to compare "secure" operating systems, even ones built via different approaches.

We briefly list some alternative approaches for further examination. An alternative definition for penetration-resistant systems by Gupta and Gligor [122, 123] requires tamperproofing and complete mediation, but defines "simple enough to verify" in terms of: (1) consistency of system global variables and objects; (2) timing consistency of condition checks; and (3) elimination of undesirable system/user dependencies. We consider such goals in the definition of the tamperproofing requirements (particularly, number three) and the security goals that we aim to verify, although we do not assess the impact of timing in this book in detail. Also, there has been a significant amount of work on formal verification tools as applied to formal specifications of security for assessing the information flows among system states [79, 172, 331]. For example, Ina Test and Ina Go are symbolic execution tools that interpret the formal specifications of a system and its initial conditions, and compare the resultant states to the expected conditions of those states. As the formal specification of systems and expectations are complex, such tools have not achieved mainstream usage, but remains an area of exploration for determining practical methods for verifying systems (e.g., [134, 132]).

## 2.5 SUMMARY

In this chapter, we define the fundamental terminology that we will use in this book to describe the secure operating system requirements.

First, the concept of a *protection system* defines the system component that enforces the access control in an operating system. A protection system consists of a *protection state* which describes the operations that are permitted in a system and *protection state operations* which describe how the protection state may be changed. From this, we can determine the operations that individual processes can perform.

Second, we identify that today's commercial operating systems use protection systems that fail to truly enforce security goals. We define a *mandatory protection system* which will enforce security in the face of attacks.

Third, we outline the architecture of an access enforcement mechanism that would be implemented by a protection system. Such enforcement mechanisms can enforce a mandatory protection state correctly if they satisfy the guarantees required of the *reference monitor concept*.

Finally, we define requirements for a secure operating system based on a reference monitor and mandatory protection system. We then describe how we aim to evaluate the operating systems described in this book against those secure operating system requirements.

Such a mandatory protection system and reference monitor within a mediating, tamperproof, and verifiable TCB constitute the *trust model* of a system, as described in Chapter 1. This trust model provides the basis for the enforcement of system security goals. Such a trust model addresses the system *threat model* based on achievement of the reference monitor concept. Because the reference monitor mediates all security-sensitive operations, it and its mandatory protection state are tamperproof, and both are verified to enforce system security goals, then is it possible to have a comprehensive *security model* that enforces a system's security goals.

Although today's commercial operating systems fail to achieve these requirements in many ways, a variety of operating systems designs in the past and that are currently available are working toward meeting these requirements. Because of the interest in secure operating systems and the variety of efforts being undertaken, it is more important than ever to determine how an operating system design aims to achieve these requirements and whether the design approaches will actually satisfy these secure operating system requirements.