NCSC-TG-030
Library No. S-240,572
Version 1

FOREWORD

A Guide to Understanding Covert Channel Analysis of Trusted Systems provides a set of good practices related to covert channel analysis. We have written this guide to help the vendor and evaluator communities understand the requirements for covert channel analysis as described in the *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. In an effort to provide guidance, we make recommendations in this technical guide that are not cited in the *TCSEC*.

This guide is the latest in a series of technical guidelines published by the National Computer Security Center. These publications provide insight to the *TCSEC* requirements for the computer security vendor and technical evaluator. The goals of the Technical Guideline Program are to discuss each feature of the *TCSEC* in detail and to provide the proper interpretations with specific guidance.

The National Computer Security Center has established an aggressive program to study and implement computer security technology. Our goal is to encourage the widespread availability of trusted computer products for use by any organization desiring better protection of its important data. One way we do this is by supporting the Trusted Product Evaluation Program. This program focuses on the security features of commercially produced and supported computer systems. We evaluate the protection capabilities against the established criteria presented in the *TCSEC*. This program, and an open and cooperative business relationship with the computer and telecommunications industries, will result in the fulfillment of our country's information systems security requirements. We resolve to meet the challenge of identifying trusted computer products suitable for use in processing information that requires protection.

I invite your suggestions for revising this technical guide. We will review this document as the need arises.

Patrick R. Gallagher, Jr.

Patrick R. Gallagher, Jr.
Director
National Computer Security Center

November 1993

ACKNOWLEDGMENTS

The National Computer Security Center (NCSC) extends special recognition and acknowledgment to Virgil D. Gligor as primary author and preparer of this document, to Jonathan K. Millen for providing significant technical input for the covert channel identification and bandwidth estimation sections, and to the first covert channel working group of the NCSC (which met from 1989 to 1991) for providing most of the material presented in Appendices A and 1B. Capt. James K. Goldston (USAF) and Capt. James A. Muysenberg (USAF) are recognized for the development, editing, and publication of this guide.

We wish to thank the many members of the computer security community who enthusiastically gave their time and technical expertise in reviewing this guide and providing valuable comments and suggestions.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 BACKGROUND

The principal goal of the National Computer Security Center (NCSC) is to encourage the widespread availability of trusted computer systems. In support of this goal, the NCSC created a metric, the *Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)* [NCSC TCSEC], against which computer systems could be evaluated.

The *TCSEC* was originally published on 15 August 1983 as CSC-STD-001-83. In December 1985, the Department of Defense adopted it, with a few changes, as a Department of Defense Standard, DoD 5200.28-STD. DoD Directive 5200.28, *Security Requirements for Automated Information Systems (AISs)* [DoD Directive], requires the *TCSEC* be used throughout the Department of Defense. The *TCSEC* is the standard used for evaluating the effectiveness of security controls built into DoD AISs.

The *TCSEC* is divided into four divisions: D, C, B, and A. These divisions are ordered in a hierarchical manner, with the highest division (A) being reserved for systems providing the best available level of assurance and security. Within divisions C and B are subdivisions known as classes, which are also ordered in a hierarchical manner to represent different levels of security in these divisions.

1.2 PURPOSE

An important set of *TCSEC* requirements, which appears in classes B2 to A1,is that of covert channel analysis (CCA). The objectives of CCA are:

- Identification of covert channels;
- Determination of covert channels' maximum attainable bandwidth;
- Handling covert channels using a well-defined policy consistent with the *TCSEC* objectives; and
- Generation of assurance evidence to show that all channels are handled according to the policy in force. To help accomplish these objectives, this guide (1) presents the relative merits of covert channel identification methods and of the covert channel information sources, (2) recommends sound bandwidth determination and handling policies and methods based on the *TCSEC* requirements, and (3) defines the types of evidence that should be provided for handling assurance.

This document provides guidance to vendors on what types of analyses they should carry out for identifying and handling covert channels in their systems, and to system evaluators and accreditors on how to evaluate the manufacturer's analysis evidence. Note, however, that the only measure of *TCSEC* compliance is the *TCSEC*. This guide contains suggestions and recommendations derived from *TCSEC* objectives but which are not required by the *TCSEC*.

This guide is not a tutorial introduction to any topic of CCA. Instead, it is a summary of analysis issues that should be addressed by operating systems designers, evaluators, and accreditors to satisfy the requirements of the B2-A1 classes. Thus, we assume the reader is an operating system designer or evaluator already familiar with the notion of covert channels in operating systems. For this reader, the guide defines a set of baseline requirements and recommendations for the analysis and evaluation of covert channels. For the reader unfamiliar with CCA techniques used to date, the following areas of further documentation and study may be useful:

- Mandatory security models and their interpretation in operating systems [Bell and La Padula76, Biba77, Denning83, Gasser88, Honeywell85a, Honeywell85b, Luckenbaugh86, Rushby85, Walter74];
- Experience with covert channel identification reported in the literature to date [Benzel84, Haigh87, He and Gligor90, Karger and Wray91, Kemmerer83, Lipner75, Loepere85, Millen76, Millen8l, Millen89b, Schaefer77, Tsai90, Wray91];
- Bandwidth estimation techniques using standard information theory [Huskamp78, Millen89a, Shannon and Weaver64]; informal bandwidth estimation techniques [Tsai and Gligor88j;
- Covert channel handling techniques [Schaefer77, Shieh and Gligor90, Hu91]; and
- Other *TCSEC* guidelines relevant to covert channel handling [NCSC Audit, NCSC Testing].

The reader who is intimately familiar with CCA techniques may want to refer only to the sections on the "*TCSEC* Requirements and Recommendations" (i.e., Sections 3.4, 4.3, and 6.1) and on "Satisfying the *TCSEC* Requirements for Covert Channel Analysis" (Chapter 7).

1.3 SCOPE

This guide refers to covert channel identification and handling methods which help assure that existent covert channels do not compromise a system's secure operation. Although the guide addresses the requirements of systems supporting the *TCSEC* mandatory policy, the analysis and handling methods discussed apply equally well to systems supporting any nondiscretionary (e.g., mandatory) security policy [Saltzer and Schroeder75]. We make additional recommendations which we derive from the stated objectives of the *TCSEC*. Not addressed are covert channels that only security administrators or operators can exploit by using privileged (i.e., trusted) software. We consider use of these channels an irrelevant threat because these administrators, who must be trusted anyway, can usually disclose classified and sensitive information using a variety of other more effective methods.

This guide applies to computer systems and products built with the intention of satisfying *TCSEC* requirements at the B2-A1 levels. Although we do not explicitly address covert channels in networks or distributed database management systems, the issues we discuss in this guide are similar to the ones for those channels.

1.4 CONTROL OBJECTIVE

Covert channel analysis is one of the areas of operational assurance. As such, its control objective is that of assurance. The assurance objective provided in [NCSC TCSEC] is the following:

> Systems that are used to process or handle classified or other sensitive information must be designed to guarantee correct and accurate interpretation of the security policy and must not distort the intent of that policy. Assurance must be provided that correct implementation and operation of the policy exists throughout the system's life-cycle.

This objective affects CCA in two important ways. First, covert channels are the result of an implementation of a nondiscretionary security policy at the operating system level; therefore, depending on how this policy is implemented within a given system, the resulting system will have fewer or more covert channels. Second, the existence of covert channels poses a potential threat to the use of the mandatory policy throughout the system's life cycle. Thus, the identification and handling of covert channels represents an important tenet of mandatory policy support in B2-A1 systems.

1.5 DOCUMENT ORGANIZATION

This guide contains seven chapters, a glossary, a bibliography, and two appendices. Chapter 2 reviews various definitions of covert channels, presents the policy implications of those definitions, and classifies channels. Chapter 3 presents various sources of covert channel information and identification methods, and discusses their relative practical advantages. Chapter 4 describes bandwidth estimation and illustrates a technique based on standard information theory that can be applied effectively in practice. Chapter 5 reviews various covert channel handling methods and policies that are consistent with the *TCSEC* requirements. Chapter 6 discusses covert channel testing and test documentation. Chapter 7 presents *TCSEC* requirements for CCA, and includes additional recommendations corresponding to B2-A1 evaluation classes. The glossary contains the definitions of the significant terms used herein. The bibliography lists the references cited in the text. Appendix A cites some examples of storage and timing channels. Appendix B describes the capabilities of several tools for covert channel identification.

2.0 COVERT CHANNEL DEFINITION AND CLASSIFICATION

In this chapter we provide several definitions of covert channels and discuss the dependency of these channels on implementations of nondiscretionary access control policies (i.e., of policy models). Also, we classify channels using various aspects of their scenarios of use.

2.1 DEFINITION AND IMPLICATIONS

The notion of covert communication was introduced in [Lampson73] and analyzed in [Lipner75, Schaefer77, Huskamp78, Denning83, Kemmerer83], among others. Several definitions for covert channels have been proposed, such as the following:

- *Definition 1* - A communication channel is covert if it is neither designed nor intended to transfer information at all. [Lampson73] (Note: Lampson's definition of covert channels is also presented in [Huskamp78].)
- *Definition 2* - A communication channel is covert (e.g., indirect) if it is based on "transmission by storage into variables that describe resource states." [Schaefer77]
- *Definition 3* - Covert channels "will be defined as those channels that are a result of resource allocation policies and resource management implementation." [Huskamp78] (Note: The computing environment usually carries out resource allocation policies and implementation.)
- *Definition 4* - Covert channels are those that "use entities not normally viewed as data objects to transfer information from one subject to another." [Kemmerer83]

The last three of the above definitions have been used successfully in various security designs for new and retrofitted operating systems and in general covert channel analyses. However, none of the above definitions brings out explicitly the notion that covert channels depend on the type of nondiscretionary access control (e.g., mandatory) policy being used and on the policy's implementation within a system design. A new definition using these concepts can be provided that is consistent with the *TCSEC* definition of covert channels, which states that a covert channel is "a communication channel that allows a process to transfer information in a manner that violates the system's security policy."

- *Definition 5* - Given a nondiscretionary (e.g., mandatory) security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects $I(S_h)$ and $I(S_i)$ of I(M) is covert if and only if any communication between the corresponding subjects S_h and S_i of the model M is illegal in M. [Tsai90]

The above definition has several consequences that help explain the relevance (or lack thereof) of covert channels to different access control policies, as listed below:

(1) Irrelevance of Discretionary Policy Models

The above definition implies that covert channels depend only on the interpretation of nondiscretionary security models. This means the notion of covert channels is irrelevant to discretionary security models.

Discretionary policy models exhibit a vulnerability to Trojan Horse attacks regardless of their interpretation in an operating system [NCSC DAC, Gasser88]. That is, implementations of these models within operating systems cannot determine whether a program acting on behalf of a user may release information on behalf of that user in a legitimate manner. Information release may take place via shared memory objects such as files, directories, messages, and so on. Thus, a Trojan Horse acting on behalf of a user could release user-private information using legitimate operating system requests. Although developers can build various mechanisms within an operating system to restrict the activity of programs (and Trojan Horses) operating on behalf of a user [Karger87], there is no general way, short of implementing nondiscretionary policy models, to restrict the activity of such programs. Thus, given that discretionary models cannot prevent the release of sensitive information through legitimate program activity, it is not meaningful to consider how these programs might release information illicitly by using covert channels.

The vulnerability of discretionary policies to Trojan Horse and virus attacks does not render these policies useless. Discretionary policies provide users a means to protect their data objects from unauthorized access by other users in a relatively benign environment (e.g., an environment free from software containing Trojan Horses and viruses). The role of nondiscretionary policies is to confine the activity of programs containing Trojan Horses and viruses. In this context, the implementation of mandatory policies suggested by the *TCSEC*, which forms an important subclass of nondiscretionary security policies, must address the problem of unauthorized release of information through covert channels.

(2) Dependency on Nondiscretionary Security Policy Models

A simple example illustrates the dependency of covert channels on the security policy model used. Consider a (nondiscretionary) separation model M that prohibits any flow of information between two subjects S_h and S_i Communication in either direction, from S_h to S_i and vice versa, is prohibited. In contrast, consider a multilevel security model, M', where messages from S_h to S_i are allowed only if the security level of S_i dominates that of S_h. Here, some communication between 5h and Si may be authorized in M'.

The set of covert channels that appears when the operating system implements model M' may be a subset of those that appear when the same operating system implements model M. The covert channels allowing information to flow from S_h to S_i in interpretations of model M could become authorized communication channels in an interpretation of model M'.

The dependency of covert channels on the (nondiscretionary) security policy models does not imply one can eliminate covert channels merely by changing the policy model. Certain covert channels will exist regardless of the type of nondiscretionary access control policy used. However, this dependency becomes important in the identification of covert channels in specifications or code by automated tools. This is the case because exclusive reliance on syntactic analysis that ignores the semantics of the security model implementation cannot avoid false illegal flows. We discuss and illustrate this in sections 3.2.2 and 3.3.

(3) Relevance to Both Secrecy and Integrity Models

In general, the notion of covert channels is relevant to any secrecy or integrity model establishing boundaries meant to prevent information flow. Thus, analysis of covert channels is equally important to the implementation of both nondiscretionary secrecy (e.g., [Bell and La

Padula76, Denning76, Denning77, Denning83, NCSC TCSEC]) and integrity models (e.g., [Biba77, Clark and Wilson87]). In systems implementing nondiscretionary secrecy models, such as those implementing the mandatory security policies of the *TCSEC* at levels B2-A1, CCA assures the discovery of (hopefully all) illicit ways to output (leak) information originating from a specific secrecy level (e.g., "confidential/personnel files/") to a lower, or incomparable, secrecy level (e.g., "unclassified/telephone directory/"). Similarly, in systems implementing nondiscretionary integrity models, such analysis also assures the discovery of (hopefully all) illicit ways to input information originating from a specific integrity level (e.g., "valued/personnel registry/") to a higher, or incomparable, integrity level (e.g., "essential/accounts payable/"). Without such assurances, one cannot implement appropriate countermeasures and, therefore, nondiscretionary security claims become questionable at best. Figures 2-1(a) and 2-1(b) illustrate the notion of illegal flows in specific nondiscretionary secrecy and nondiscretionary integrity models.

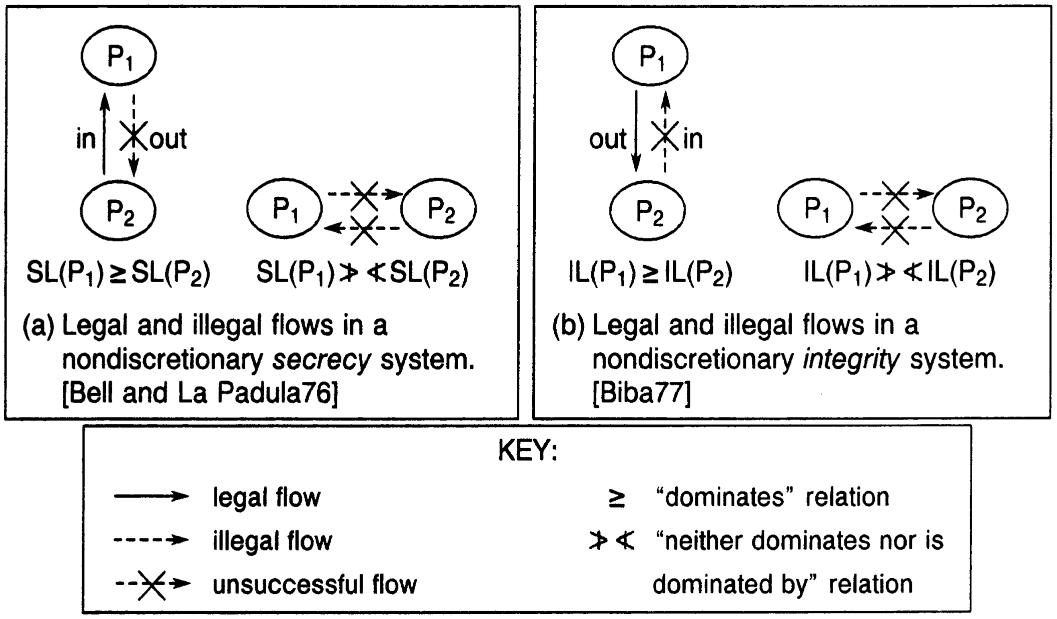

P1
in
out
P2
P1
P2
SL(P1) ≥ SL(P2)
SL(P1) ⊁⊀ SL(P2)
(a) Legal and illegal flows in a nondiscretionary *secrecy* system. [Bell and La Padula76]
P1
out
in
P2
P1
P2
IL(P1) ≥ IL(P2)
IL(P1) ⊁⊀ IL(P2)
(b) Legal and illegal flows in a nondiscretionary *integrity* system. [Biba77]
KEY:
legal flow
illegal flow
unsuccessful flow
≥ "dominates" relation
⊁⊀ "neither dominates nor is dominated by" relation

Figure 2-1. Legal and Illegal Flows

Example 0 - Relevance of Covert Channels to an Integrity Model

Figure 2-2 illustrates the relevance of covert channels to nondiscretionary integrity models. Although this figure assumes a specific nondiscretionary integrity model (i.e., Biba's [Biba77]), covert channels are equally relevant to all nondiscretionary integrity models. In Figure 2-2, a user logged in at the integrity level IL_1 invokes, through a command processor (i.e., the shell), an accounts payable application that prints payees, names on signed-check papers on a printer. The user is trusted to operate at integrity level IL_1 and, by virtue of this trust, his input to the accounts payable application is also classified at integrity level IL_1. For similar reasons, both the accounts payable application and the printer are activated at the current integrity level IL_1. However, the accounts payable application (and, possibly, the shell) consists of an untrusted set of programs.

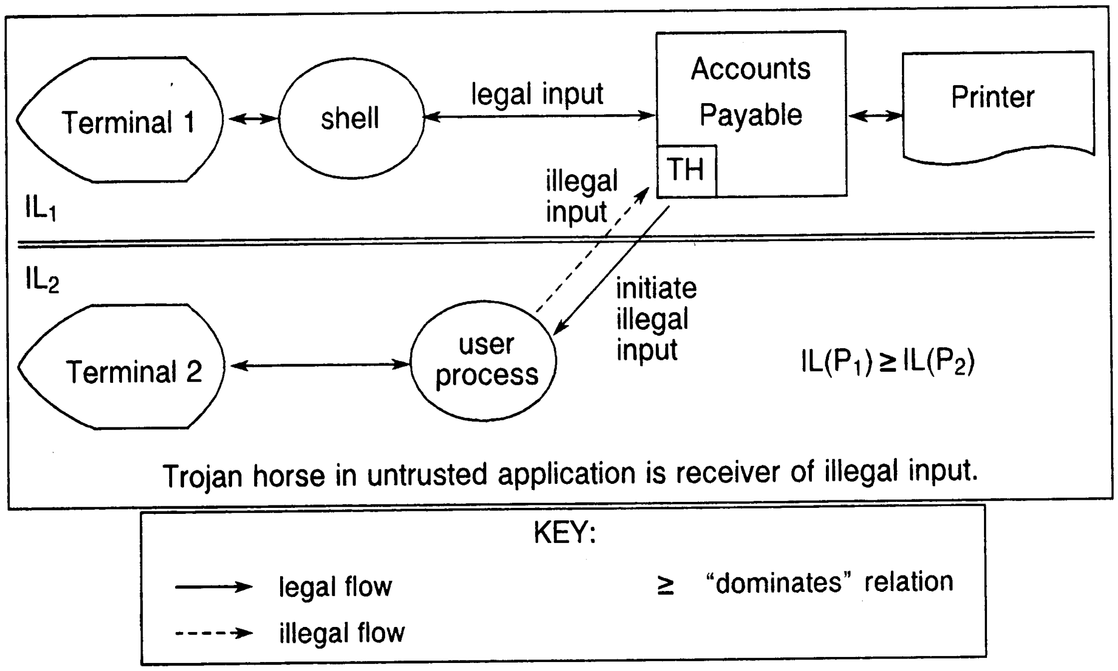


Figure 2-2. Relevance of Covert Channels to an Integrity Model

The presence of untrusted software in the above example should not be surprising. Most application programs running on trusted computing bases (TCBs) supporting nondiscretionary secrecy consist of untrusted code. Recall that the ability to run untrusted applications on top of TCBs without undue loss of security is one of the major tenets of trusted computer systems. Insisting that all applications that might contain a Trojan Horse, which could use covert channels affecting integrity, be included within an integrity TCB is analogous to insisting that all applications that might contain a Trojan Horse, which could use covert channels affecting secrecy, be included within a secrecy TCB, and would be equally impractical.

If the untrusted accounts payable application contains a Trojan Horse, the Trojan Horse program could send a (legal) message to a user process running at a lower integrity level IL_2, thereby initiating the use of a covert channel. In this covert channel, the Trojan Horse is the receiver of (illegal) lower integrity-level input and the user process is the sender of this input.

The negative effect of exploiting this covert channel is that an untrusted user logged in at a lower integrity level could control the accounts payable application through illegal input, thereby producing checks for questionable reasons. One can find similar examples where covert channels help violate *any* nondiscretionary integrity boundary, not just those provided by lattice-based integrity models (e.g., [Biba77]). Similar examples exist because, just as in the case of TCBs protecting sensitive information classified for secrecy reasons, not all applications running on trusted bases protecting sensitive information for integrity reasons can be verified and proved to be free of miscreant code.

(4) Dependency on TCB Specifications

To illustrate the dependency of covert channels on a system's TCB specifications (Descriptive or Formal Top-Level), we show that changes to the TCB specifications may eliminate existent, or introduce new, covert channels. The specifications of a system's TCB include the specifications of primitives which operate on system subjects, objects, access privileges, and security levels, and of access authorization, object/subject creation/destruction rules, for example. Different interpretations of a security model are illustrated in [Honeywell85a, Honeywell85b, Luckenbaugh86]. Changes to a TCB's specifications may not necessarily require a change of security model or a change of the security model interpretation.

Example 1 - Object Allocation and Deallocation

As an example of the effect of TCB specification changes on covert channel existence (and vice versa), consider the case of an allocator of user-visible objects, such as memory segments. The specifications of the allocator must contain explicit "allocate/deallocate" (TCB) operations that can be invoked dynamically and that subjects can share. A covert channel between the subjects using these user-visible objects exists here [Schaefer77]. However, if the dynamic allocator and, consequently, its specifications are changed to disallow the dynamic allocation/deallocation of objects in a shared memory area, the covert channel disappears. Static object allocation in a shared memory area, or dynamic object allocation in a memory area partitioned on a security level basis, need not change the interpretation of the system's subjects and objects; it only needs to change the specification of the rules for the creation and destruction of a type of object. Although eliminating dynamic sharing of resources and either preallocating objects or partitioning resources on a per-security-level basis represent effective ways to remove some covert channels, they are neither necessary nor possible in all cases because they may cause performance losses.

Though this example illustrates the dependency of covert channels on TCB specifications, it is not a general solution for eliminating covert channels. In fact, we can find other examples to show that changing a TCB's specifications may actually increase the number of covert channels.

Example 2 - Upgraded Directories

As a second example of the strong dependency between the covert channel definition and TCB specifications, consider the creation and destruction of upgraded directories in a system supporting mandatory security and using specifications of interfaces similar to those of UNIX*. The notion of an upgraded directory [Whitmore73, Schroeder77, Gligor87], its creation and removal, is illustrated in Figures 2-3(a)-(d).

In such a system, whenever a user attempts to remove an upgraded directory from level $L_h > L_i$ where he is authorized to read and write it (as in Figure 2-3(c)), the remove operation fails because it violates the mandatory authorization check (the level of the removing process, L_h, must equal that of the parent directory, L_i). In contrast, the same remove operation invoked by a process at level $L_i < L_h$ succeeds (Figure 2-3(d)).

* UNIX is a registered trademark of the UNIX Systems Laboratories.

However, a covert channel appears because of the specification semantics of the remove operation in UNIX "rmdir." This specification says a nonempty directory cannot be removed. Therefore, if the above user logs in at level L_i and tries to remove the upgraded directory from the higher level L_h, the user process can discover whether any files or directories at level $L_h > L_i$ are linked to the upgraded directory. Thus, another process at level L_h can transmit a bit of information to the user process at level $L_i < L_h$ by creating and removing (e.g., unlinking) files in the upgraded directory. Figure 2-4 illustrates this concept.

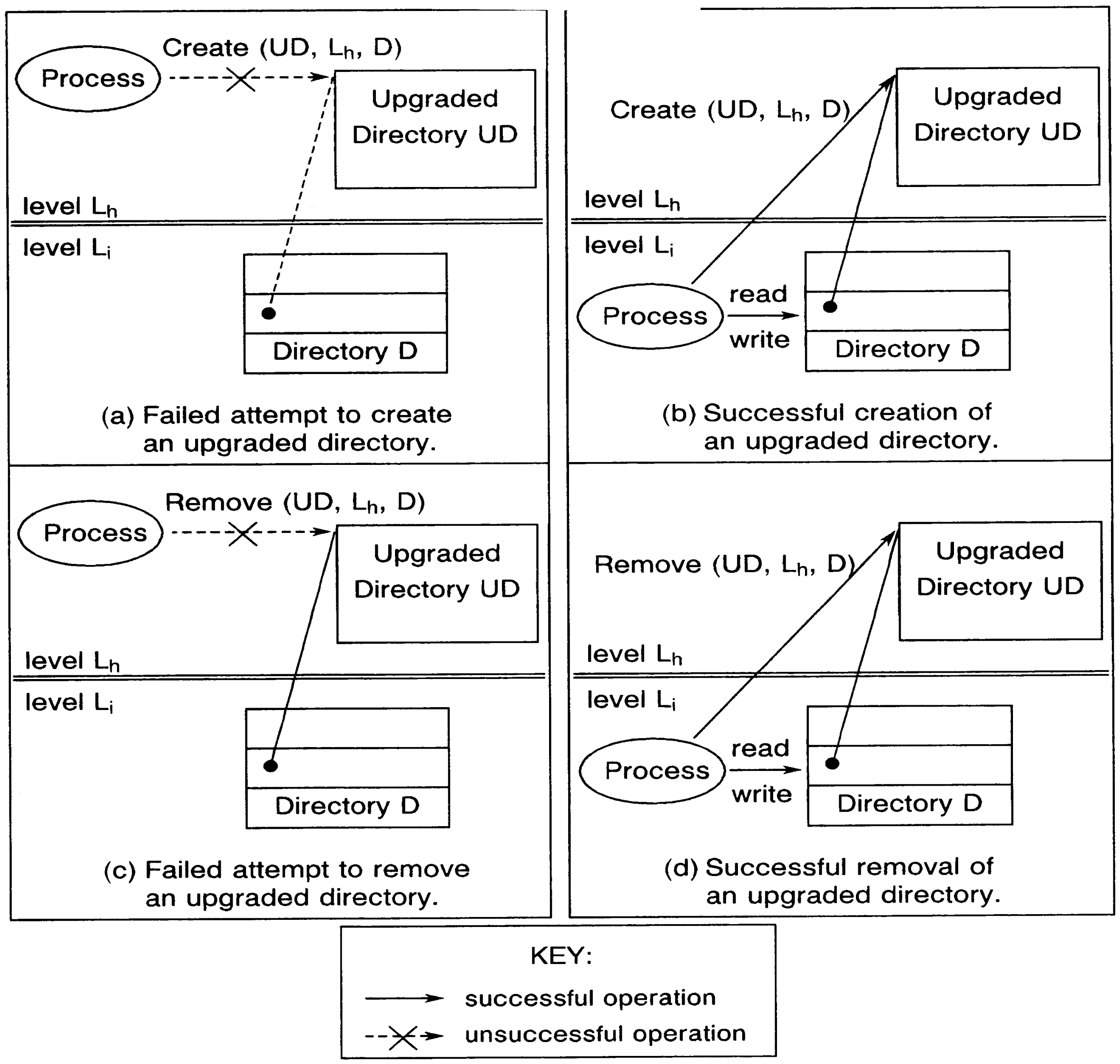


Figure 2-3. Creation and Destruction of an Upgraded Directory at Level $L_h > L_i$

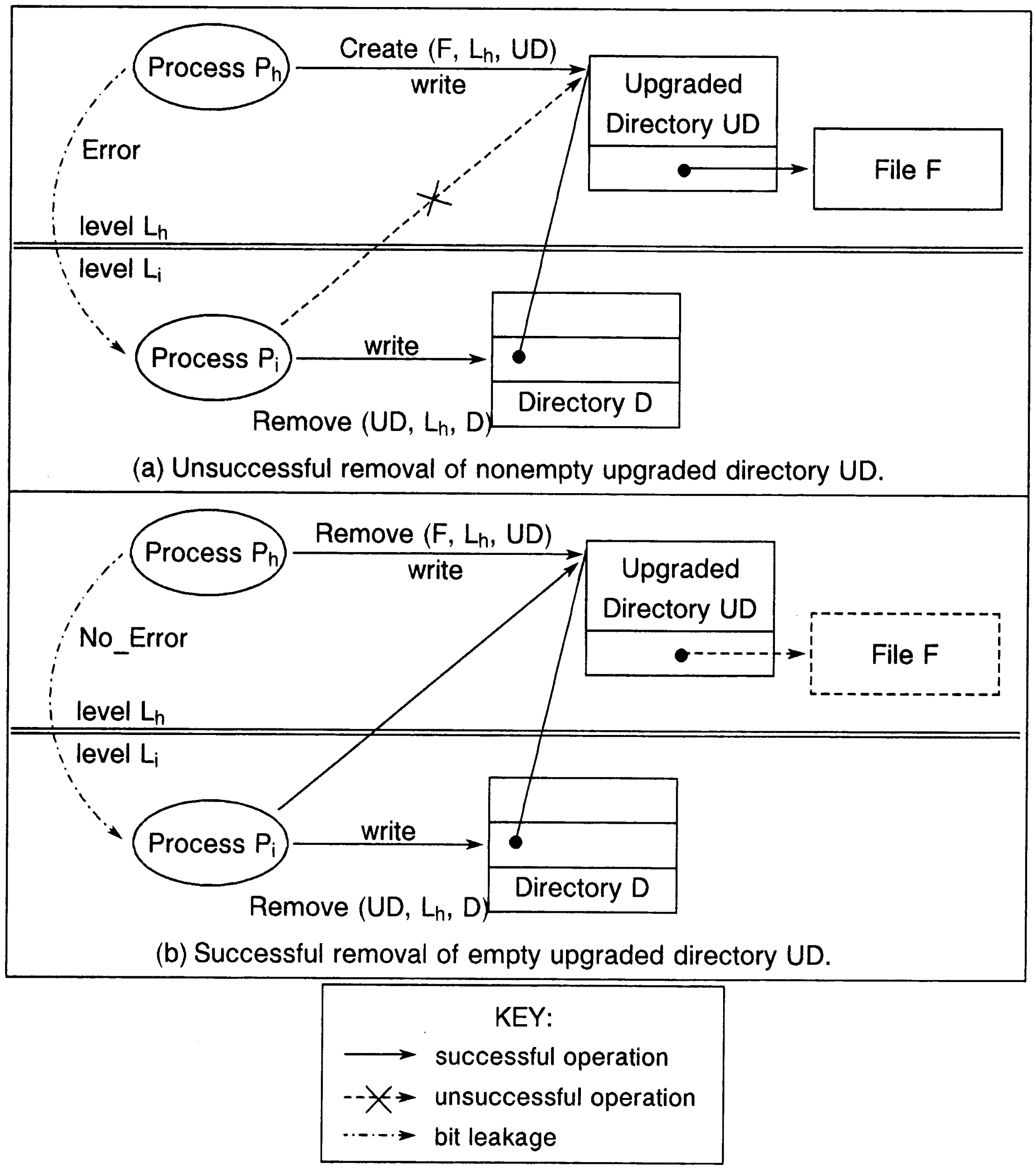


Figure 2-4. Covert Channel Caused by (UNIX) TCB Interface Conventions (where $L_h > L_i$)

This covert channel would not appear if nonempty directories, and the directory subtree started from them, could be removed (e.g., as in Multics [Whitmore73, Bell and La Padula76]). However, if the specification of directory removal is changed, disallowing removal of nonempty directories (as in UNIX), the covert channel appears. One cannot eliminate the channel without modifying the UNIX user-visible interface. This is an undesirable alternative given that user programs may depend on the interface convention that nonempty UNIX directories cannot be removed. One cannot invent a new TCB specification under which either directories are not user-visible objects or in which the notion of upgraded directories disappears for similar reasons; that is, the UNIX semantics must be modified.

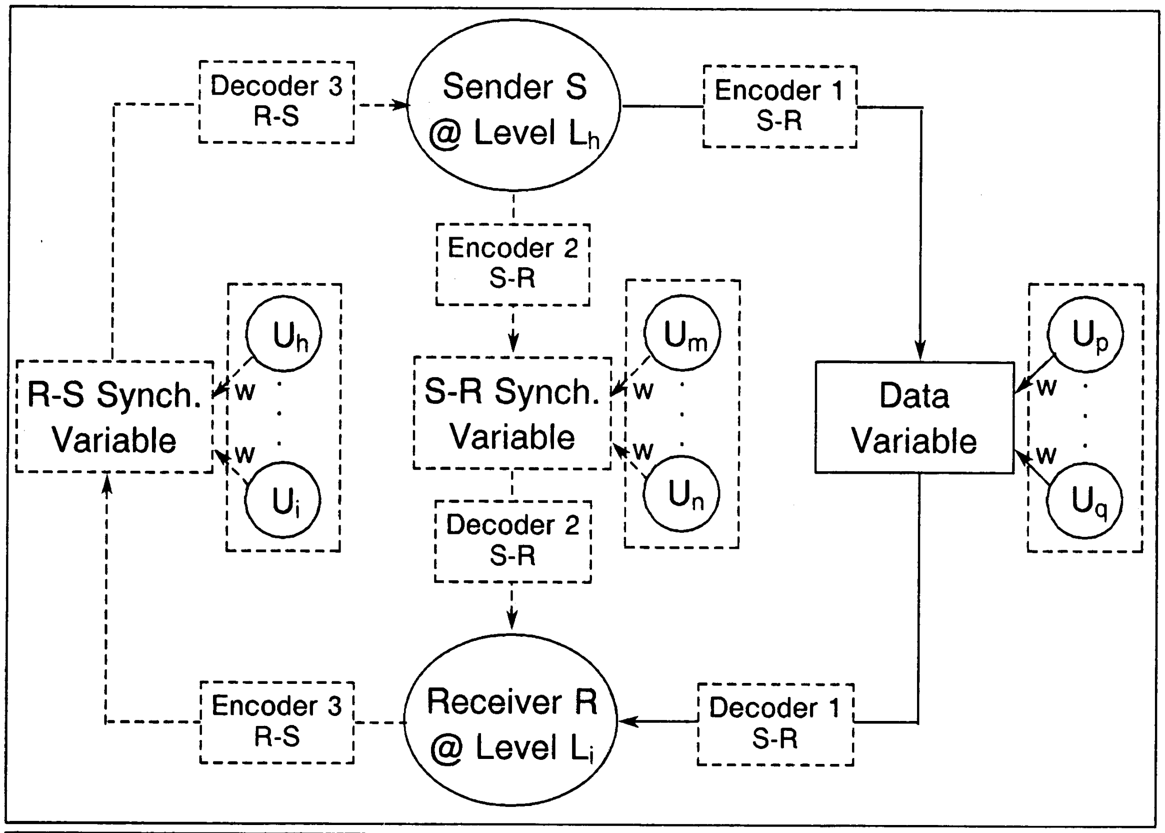
2.2 CLASSIFICATION

2.2.1 Storage and Timing Channels

In practice, when covert channel scenarios of use are constructed, a distinction between covert storage and timing channels [Lipner75, Schaefer77, NCSC TCSEC, Hu91, Wray91] is made even though theoretically no fundamental distinction exists between them. A potential covert channel is a storage *channel* if its scenario of use "involves the direct or indirect writing of a storage location by one process [i.e., a subject of I(M)] and the direct or indirect reading of the storage location by another process." [NCSC TCSEC] A potential covert channel is a *timing channel* if its scenario of use involves a process that "signals information to another by modulating its own use of system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process." [NCSC TCSEC] In this guide, we retain the distinction between storage and timing channels exclusively for consistency with the *TCSEC*.

In any scenario of covert channel exploitation, one must define the synchronization relationship between the sender and the receiver of information. Thus, covert channels can also be characterized by the synchronization relationship between the sender and the receiver. In Figure 2-5, the sender and the receiver are asynchronous processes that need to synchronize with each other to send and decode the data. The purpose of synchronization is for one process to notify the other process it has completed reading or writing a data variable. Therefore, a covert channel may include not only a covert data variable but also two synchronization variables, one for sender-receiver synchronization and the other for the receiver-sender synchronization. Any form of synchronous communication requires both the sender-receiver and receiver-sender synchronization either implicitly or explicitly [Haberman72]. Note that synchronization operations transfer information *in both directions*, namely from sender to receiver and vice versa and, therefore, these operations may be indistinguishable from data transfers. Thus, the synchronization and data variables of Figure 2-5 may be indistinguishable.

Some security models, and some of their interpretations, allow receiver-sender communication for subsets of all senders and receivers supported in the system. For example, all mandatory security models implemented in commercial systems to date allow information to flow from a low security level to a higher one. However, sender-receiver synchronization may still need a synchronization variable to inform the receiver of a bit transfer. A channel that does not include sender-receiver synchronization variables in a system allowing the receiver-sender transfer of messages is called a *quasi-synchronous channel*. The idea of quasi-synchronous channels was introduced by Schaefer in 1974 [Reed and Kanodia78].



Note: Processes U_h, . . . , U_i, . . . , U_m, . . . , U_n, . . . , U_p, . . . , U_q may introduce noise whenever they write on the variables, and may introduce delay whenever they do not write on variables and run between the sender and the receiver.

Figure 2-5. Representation of a Covert Channel between Sender S and Receiver R (where $L_h > L_i$ or $L^* * L_i$)

In all patterns of sender-receiver synchronization, synchronization data may be included in the data variable itself at the expense of some bandwidth degradation. Packet-formatting bits in ring and Ethernet local area networks are examples of synchronization data sent along with the information being transmitted. Thus, explicit sender-receiver synchronization through a separate variable may be unnecessary. Systems implementing mandatory security models allow messages to be sent from the receiver to the sender whenever the security level of the sender dominates that of the receiver. In these cases, explicit receiver-sender synchronization through a separate variable may also be unnecessary.

The representation of a covert channel illustrated in Figure 2-5 can also be used to distinguish between scenarios of storage and timing channels. For example, a channel is a storage channel when the synchronization or data transfers between senders and receivers use storage variables, whereas a channel is a timing channel when the synchronization or data transfers between senders and receivers include the use of a common time reference (e.g., a clock). Both storage and timing channels use at least one storage variable for the transmission/sending of the information being transferred. (Note that storage variables used for timing channels may be ephemeral in the sense that the information transferred through them may be lost after it is sensed by a receiver. We discuss this in more detail in Appendix A.) Also, a timing channel may be converted into a storage channel by introducing explicit storage variables for synchronization; and vice versa, a storage channel whose synchronization variables are replaced by observations of a time reference becomes a timing channel.

Based on the above definitions of storage and timing channels, the channels of Examples 1 and 2 are storage channels. Examples 3 and 4 below illustrate scenarios of timing channels. Appendix A presents additional examples of both storage and timing channels.

Example 3 - Two Timing Channels Caused by CPU Scheduling

Quantum-based central processing unit (CPU) scheduling provides two typical examples of timing channels (Figure 2-6). In the first example, the sender of information varies the nonzero CPU time, which it uses during each quantum allocated to it, to send different symbols. For 0 and 1 transmissions, the sender picks two nonzero values for the CPU time used during a quantum, one representing a 0 and the other a 1. This channel is called the "quantum-time channel" in [Huskamp78]. The receiver of the transmitted information decodes the transmitted information by measuring its waiting time for the CPU. If only the receiver and the sender are in the system, the receiver can decode each transmitted bit correctly with probability one for some quantum sizes. A condition of this channel is that the sender be able to block itself before the end of some quantum and reactivate itself before the beginning of the next quantum. The sender can meet this condition in a variety of ways depending upon the size of the quantum (e.g., a typical range for quanta is 50-1000 milliseconds). For example, the sender may use an "alarm clock" to put itself to sleep for a fraction of the quantum time, or it may generate a page fault (whose handling may take only a fraction of a quantum time also). A quantum of 100-200 milliseconds is sufficiently large for either case.

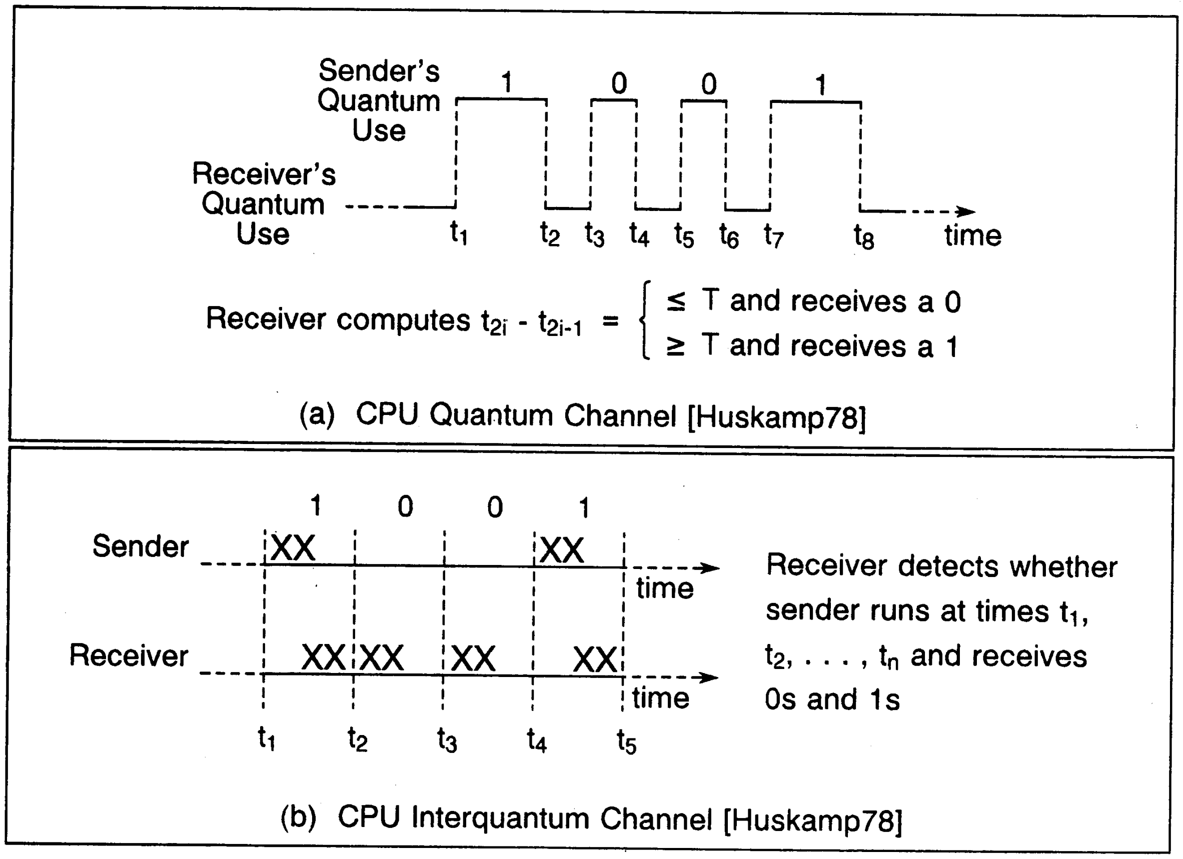

Sender's Quantum Use
Receiver's Quantum Use
1
0
0
1
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
time
Receiver computes $t_{2i} - t_{2i-1} = \begin{cases} \leq T \text{ and receives a 0} \\ \geq T \text{ and receives a 1} \end{cases}$
(a) CPU Quantum Channel [Huskamp78]
Sender
Receiver
XX
time
Receiver detects whether sender runs at times t_1, t_2, . . . , t_n and receives 0s and 1s
(b) CPU Interquantum Channel [Huskamp78]

Figure 2-6. Two CPU Timing Channels

In the second example of Figure 2-6, the sender transmits information to the receiver by encoding symbols, say 0s and 1s, in the time between two successive CPU quanta. This channel is called the "interquantum-time channel" [Huskamp78], and is shown in Figure 2-6(b) for the case where only the sender and the receiver appear in the system. To send information, the sender and the receiver agree on set times for sending the information. The transmission strategy is for the sender to execute at time "t_i" if the i-th bit is 1, and to block itself if the i-th bit is 0. The receiver can tell whether the sender executes at time t_i because the receiver cannot execute at the same time.

Example 4 - Other Timing Channels Caused by Shared Hardware Resources

The CPU scheduling channels of Example 3 appear because processes at different secrecy or integrity levels share a hardware resource, namely the CPU. Other sharable hardware resources provide similar timing channels. For example, in any multiprocessor design, hardware resources are shared. Multiple processors share the same bus in shared-bus architectures, share the same memory ports in bus-per-processor architectures, and share multiple busses and memory ports in crossbarswitch architectures, as shown in Figure 2-7. In all multiprocessor architectures, each instruction referencing the memory must lock the shared resource along the CPU-memory interconnection path for at least one memory cycle. (The number of cycles during which the shared resource must be locked depends on the instruction semantics.) Hardware controllers of the shared resource mediate lock conflicts. When the shared resource is no longer needed during the execution of the instructon, the resource is unlocked.

Whenever two processes at two different levels execute concurrently on two separate processors, a covert channel appears that is similar to the CPU interquantum channel presented in Example 3. That is, the sender and the receiver processes establish by prior agreement that the sender process executes at time"t_i"if the i-th bit is a 1 and does not execute (or at least does not execute memoryreferencing instructions) at time "t_i" if the i-th bit is a 0. The receiver can execute a standard set of memory-referencing instructions and time their execution. Thus, the receiver can discover whether the sender executes at time "t_i" by checking whether the duration of the standard set of timed instructions was the expected 1 or longer. As with the CPU channels of Example 3, these channels appear in any multiprocessor system regardless of the nondiscretionary model interpretation. Note that adding per-processor caches, which helps decrease interprocessor contention to shared hardware resources, cannot eliminate these channels. The sender and receiver processes can fill up their caches and continue to exploit interprocessor contention to transmit information.

Appendix A provides other examples of timing channels, which also appear due to the sharing of other hardware resources.

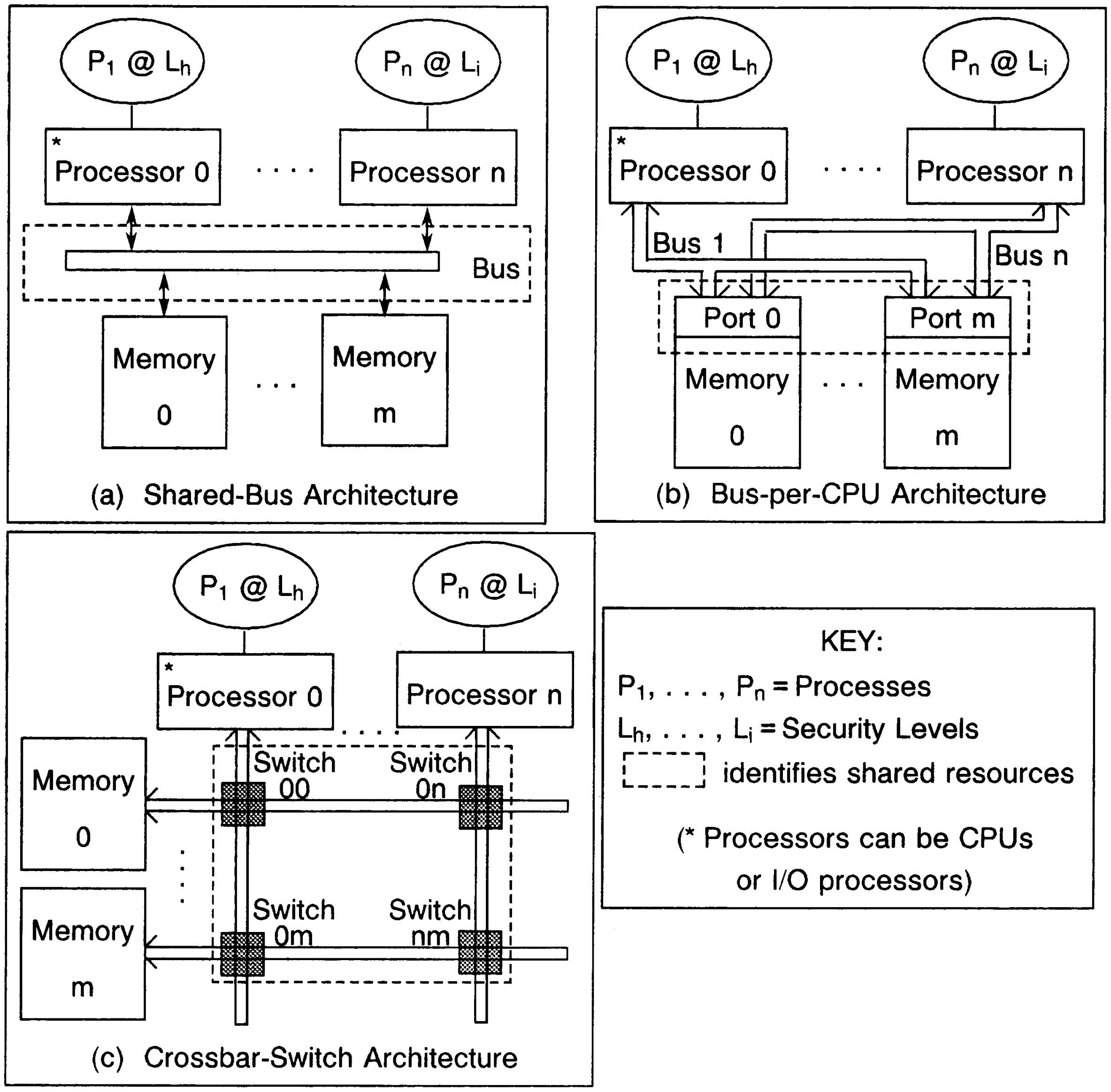

P1 @ Lh
Pn @ Li
* Processor 0
Processor n
Bus
Memory 0
Memory m
(a) Shared-Bus Architecture
Bus 1
Bus n
Port 0
Port m
(b) Bus-per-CPU Architecture
Switch 00
Switch 0n
Switch 0m
Switch nm
(c) Crossbar-Switch Architecture
KEY:
P1, . . . , Pn = Processes
Lh, . . . , Li = Security Levels
identifies shared resources
(* Processors can be CPUs or I/O processors)

Figure 2-7. Examples of Shared Hardware Resources in Multiprocessor Architectures

2.2.2 Noisy and Noiseless Channels

As with any communication channel, covert channels can be noisy or noiseless. A channel is said to be noiseless if the symbols transmitted by the sender are the same as those received by the receiver with probability 1. With covert channels, each symbol is usually represented by one bit and, therefore, a covert channel is noiseless if any bit transmitted by a sender is decoded correctly by the receiver with probability 1. That is, regardless of the behavior of other user processes in the system, the receiver is guaranteed to receive each bit transmitted by the sender.

The covert channel of Example 2 is a noiseless covert channel. The sender and receiver can create and remove private upgraded directories, and no other user can affect in any way whether the receiver receives the error/no_error signal. Thus, with probability 1, the receiver can decode the bit value sent by the sender. In contrast, the covert channels of Examples 3 and 4 are noisy channels because, whenever extraneous processes—not just the sender and receiver-use the shared resource, the bits transmitted by the sender may not be received correctly with probability 1 unless appropriate error-correcting codes are used. The error-correcting codes used depend on the frequency of errors produced by the noise introduced by extraneous processes (shown in Figure 2-5) and decrease the maximum channel bandwidth. Thus, although error-correcting codes help change a noisy channel into a noiseless one, the resulting channel will have a lower bandwidth than the similar noise-free channel.

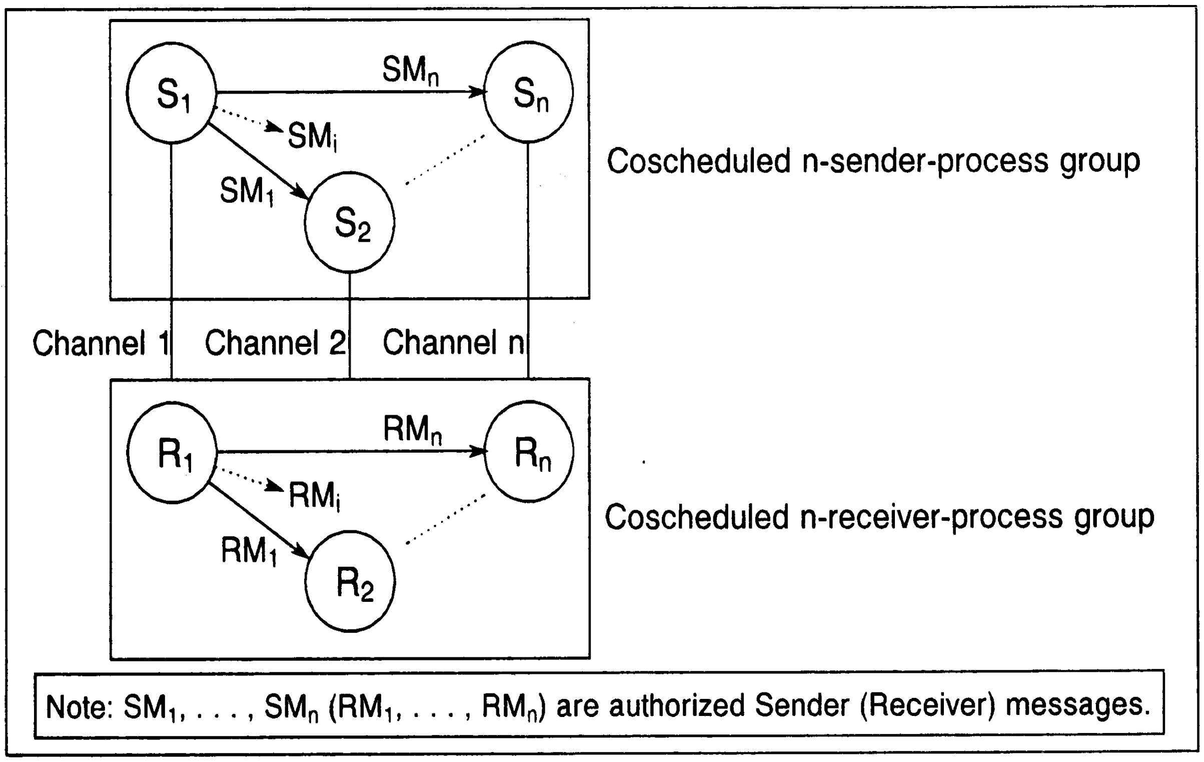
We introduce the term "bandwidth" here to denote the rate at which information is transmitted through a channel. Bandwidth is originally a term used in analog communication, measured in hertz, and related to information rate by the "sampling theorem" (generally attributed to H. Nyquist although the theorem was in fact known before Nyquist used it in communication theory [Haykin83]). Nyquist's sampling theorem says that the information rate in bits (samples) per second is at most twice the bandwidth in hertz of an analog signal created from a square wave. In a covert channel context, bandwidth is given in bits/second rather than hertz, and is commonly used, in an abuse of terminology, as a synonym for information rate. This use of the term "bandwidth" is also related to the notion of "capacity." The capacity of a channel is its maximum possible error-free information rate in bits per second. By using error-correcting codes, one can substantially reduce the error rates of noisy channels. Error-correcting codes decrease the effective (i.e., error-free) information rate relative to the noisy bit rate because they create redundancy in the transmitted bit stream. Note that one may use error-detecting, rather than error-correcting, codes in scenarios where the receiver can signal the sender for retransmissions. All of these notions are standard in information theory [Gallager68].

2.2.3 Aggregated versus Nonaggregated Channels

Synchronization variables or information used by a sender and a receiver may be used for operations on multiple data variables. Multiple data variables, which could be independently used for covert channels, may be used as a group to amortize the cost of synchronization (and, possibly, decoding) information. We say the resulting channels are aggregated. Depending on how the sender and receiver set, read, and reset the data variables, channels can be aggregated serially, in parallel, or in combinations of serial and parallel aggregation to yield optimal (maximum) bandwidth.

If all data variables are set, reset, and read serially, then the channel is serially aggregated. For example, if process P_h of Example 2 (Figure 2-4) uses multiple upgraded directories designated "empty/nonempty" before transferring control to process P_i, the signaling channel will be serially aggregated. Similarly, if all data variables are set, reset, and read in parallel by multiple senders and receivers, then the channel is aggregated in parallel. Note that combinations of serial/parallel aggregaton are also possible. For example, the data variables may be set in parallel but read serially and vice versa. However, such combinations do not maximize bandwidth and are, therefore, of limited interest.

Parallel aggregation of covert channel variables requires, for bandwidth maximization reasons, that the sender and receiver pairs be scheduled on different processors at the same time as a group, as illustrated in Figure 2-8 and in [Gligor86]. Otherwise, the bandwidth of the parallel aggregation degrades to that of a serially aggregated channel. The application programmer can strictly control group scheduling of senders and receivers in multiprocessor operating systems such as Medusa or StarOS [Osterhout80, Jones79], which use "coscheduling" [Osterhout82]. Also group scheduling may be possible in multiple workstation systems such as those used in LOCUS [Walker83] or Apollo [Leach83] whenever multiple workstatons are available to a single application. In such systems, the analysis of individual covert channels is insufficient to determine the maximum covert channel bandwidth.



S_1
SM_n
S_n
SM_i
SM_1
S_2
Coscheduled n-sender-process group
Channel 1
Channel 2
Channel n
R_1
RM_n
R_n
RM_i
RM_1
R_2
Coscheduled n-receiver-process group
Note: $SM_1, \ldots, SM_n$ ($RM_1, \ldots, RM_n$) are authorized Sender (Receiver) messages.

Figure 2-8. Example of n Channels Aggregated in Parallel

Parallel aggregation of covert channels also requires, for bandwidth maximizaton reasons, that the synchronization messages between all senders, and those between all receivers, be transmitted at a much higher speed than those between senders and receivers. In practice, messages sent among senders, and those sent among receivers, have negligible transmission delays compared to those used by covert channels between senders and receivers. (Also, note that all messages among senders and those among receivers are authorized messages.)

2.3 COVERT CHANNELS AND FLAWED TCB SPECIFICATIONS

An unresolved issue of covert channel definition is whether one can make a distinction between a covert channel and a flaw introduced by the implementation of the security models. In other words, one would like to differentiate between implementation flaws and covert channels, if possible, for practical reasons. For example, both implementors and evaluators of systems supporting mandatory access controls in class B1 could then differentiate between flaws and covert channels. They could determine whether instances of leakage of classified information must be eliminated or otherwise handled or ignored until the B2 level and above.

The covert communication Definition 5 does not differentiate between covert channels and interpretation or TCB specification flaws. This definition implies that, in a fundamental sense, covert channels are in fact flaws of nondiscretionary access control policy implementations, which are sometimes unavoidable in practice regardless of the implementors' design (e.g., Example 3). However, the focus of that definition on the notion of model implementation may help provide a criterion for distinguishing between different types of covert channels or implementation flaws.

To define a distinguishing criterion, let us review Examples 1-4. Examples 1 and 2 show that a change of the TCB specification can, in principle, eliminate the existent covert channels in the specific systems under consideration. In contrast, Examples 3 and 4 show that as long as any system allows the sharing of the CPUs, busses, memory, input/output (I/O) and other hardware resources, covert channels will appear for *any TCB specification*. Furthermore, Example 2 illustrates that, in many systems, a change of TCB specification that would eliminate a covert channel may sometimes be impractical. That is, evidence may exist showing that contemplated changes of the TCB specification would cause a significant loss of compatibility with existing interfaces of a given system. Similar examples can be found to illustrate that changes of TCB specifications may help eliminate other covert channels (or flaws) at the expense of loss of functionality or performance in a given system (e.g., Example 1).

The following criterion may help distinguish between different types of covert channels (or flaws) in practice, thereby providing the necessary input for covert channel, or flaw, handling at levels B1 versus levels B2-A1:

- *Fundamental Channels* - A flaw of a TCB specification that causes covert communication represents a fundamental channel if and only if that flaw appears under any interpretation of the nondiscretionary security model in any operating system.

- *Specific TCB Channels* - A flaw of a TCB specification that causes covert communication represents a specific TCB channel if and only if that flaw appears only under a specific interpretation of the nondiscretionary security model in a given operating system.

- *Unjustifiable Channels* - A flaw of a TCB specification that causes covert communication represents an unjustifiable channel if and only if that flaw appears *only under a specific but unjustifiable interpretation* of a nondiscretionary security model in a given operating system. (The primary difference between specific TCB and unjustifiable channels is in whether any evidence exists to justify the existence of the respective channels.)

Using this criterion, the covert channels of Examples 3 and 4 are fundamental channels, whereas those of Examples 1 and 2 are specific TCB channels.

The above criterion for distinguishing different types of covert channels (or flaws) suggests the following differentiation policy for B1 and B2A1 systems. For B1 systems, there should be no handling obligation of fundamental covert channels; specific TCB channels should be handled under the policies in force for classes B2Al (as recommended in Chapter 5 of this guide); unjustifiable channels should be eliminated by a change of TCB specification or model implementation for any B-rated systems.

3.0 COVERT CHANNEL IDENTIFICATION

We discuss in this chapter the representation of a covert channel within a system, the sources of information for covert channel identification, and various identification methods that have been used to date and their practical advantages and disadvantages. We also discuss the TCSEC requirements for covert channel identification and make additional recommendations.

A covert channel can be represented by a TCB internal variable and two sets of TCB primitives, one for altering (PA_h) and the other for viewing (PV_i) the values of the variable in a way that circumvents the system's mandatory policy. Multiple primitives may be necessary for viewing or altering a variable because, after viewing/altering a variable, the sender and/or the receiver may have to set up the environment for sending/reading the next bit. Therefore, the primary goal of covert channel identification is to discover all TCB internal variables and TCB primitives that can be used to alter or view these variables (i.e., all triples <variable; PA_h, PV_i>). A secondary, related goal is to determine the TCB locations within the primitives of a channel where time delays, noise (e.g., randomized table indices and object identifiers, spurious load), and audit code may be placed for decreasing the channel bandwidth and monitoring its use. In addition to TCB primitives and variables implemented by kernel and trusted processes, covert channels may use hardware-processor instructions and user-visible registers. Thus, complete covert channel analysis should take into account a system's underlying hardware architecture, not just kernels and trusted processes.

3.1 SOURCES OF INFORMATION FOR COVERT CHANNEL IDENTIFICATION

The primary sources of information for covert channel identification are:

- System reference manuals containing descriptions of TCB primitives, CPU and I/O processor instructions, their effects on system objects and registers, TCB parameters or instruction fields, and so on;
- The detailed top-level specification (DTLS) for B2-A1 systems, and the Formal top-level specification (FTLS) for A1 systems; and
- TCB source code and processor-instruction (micro) code.

The advantage of using system reference manuals for both TCB-primitive and processor-instruction descriptions is the widespread availability of this information. Every implemented system includes this information for normal everyday use and, thus, no added effort is needed to generate it. However, there are disadvantages to relying on these manuals for covert channel identification. First, whenever system reference manuals are used, one can view the TCB and the processors only as essentially "black boxes." System implementation details are conspicuous by their absence. Thus, using system reference manuals, one may not attain the goal of discovering all, or nearly all, channels. Whenever these manuals are the only sources of information, the channel identification may only rely on guesses and possibly on analogy with specifications of other systems known to contain covert channels. Second, and equally important, is the drawback that analysis based on system reference information takes place too late to be of much help in covert channel handling. Once a system is implemented and the manuals written, the option of eliminating a discovered covert channel by removing a TCB interface convention may no longer be available. Third, few identification methods exist that exhibit any degree of precision and that can rely exclusively on information from system reference manuals. The inadequacy of using only system reference manuals for CCA is illustrated in Example 6 of Section 3.2.3.

Most identification methods developed to date have used formal top-level TCB specifications as the primary source of covert channel identification. The use of top-level specifications has significant advantages. First, these specifications usually contain more detailed, pertinent information than system reference manuals. Second, use of top-level specifications helps detect design flaws that may lead to covert channels in the final implementation. Early detection of design flaws is a useful prerequisite for correct design because one can minimize efforts expended to correct design flaws. Third, tools aiding the identification process exist for the FTLS and thus one gains additional assurance that all channels appearing within the top-level specifications are found (see Appendix B).

However, total reliance on analysis of top-level specifications for the identificaton of covert channels has two significant disadvantages. First, it cannot lead to the identification of all covert channels that may appear in implementation code. Formal methods for demonstrating the correspondence between information flows of top-level specifications and those of implementation code do not exist to date. Without such methods, guarantees that all covert storage channels in implementation code have been found are questionable at best. The only significant work on specification-to-code correspondence on an implemented system (i.e., the Honeywell SCOMP [Benzel84]) reported in the literature to date has been thorough but informal. This work shows that, in practice, a significant amount of implementation code has no correspondent formal specifications. Such code includes performance monitoring, audit, debugging, and other code, which is considered security-policy irrelevant but which, nevertheless, may contain variables providing potential storage channels.

Second, formal/descriptive top-level specifications of a TCB may not include sufficient specification detail of data structures and code to detect indirect information flows within TCB code that are caused by the semantics of the implementation language (e.g., control statements, such as alternation statements, loops, and so on; pointer assignments, variable aliasing in structures [Schaefer89, Tsai90]). Insufficient detail of specifications used for information flow and storage channel analysis may also cause inadequate implementation of nondiscretionary access controls and channel-handling mechanisms. This is the case because, using the results of top- level specification analysis, one cannot determine with certainty the placement of code for access checks, channel use audits, and time delays to decrease channel bandwidth within TCB code.

In contrast with the significant efforts for the analysis of design specifications, little practical work has been done in applying CCA to implementation code or to hardware. Identifying covert storage channels in source code has the advantages that (1) potentially all covert storage channels can be found (except those caused by hardware), (2) locations within TCB primitives for placement of audit code, de-lays, and noise can be found, and (3) adequacy of access-check placement within TCB primitives could be assessed [Tsai90]. However, analysis of TCB source code is very labor-intensive, and few tools exist to date to help alleviate the dearth of highly skilled personnel to perform such labor-intensive activity.

3.2 IDENTIFICATION METHODS

All of the widely used methods for covert channel identification are based on the identification of illegal information flows in top-level design specifications and source code, as first defined by [Denning76, 77, 83] and [Millen76]. Subsequent work by [Andrews and Reitman80] on information-flow analysis of programming language statements extended Denning's work to concurrent-program specifications.

3.2.1 Syntactic Information-Flow Analysis

In all flow-analysis methods, one attaches information-flow semantics to each statement of a specification (or implementation) language. For example, a statement such as "a: = b" causes information to flow from b to a (denoted by b $\rightarrow$ a) whenever b is not a constant. Similarly, a statement such as "if v = k then w: = b else w: = c" causes information to flow from v to w. (Other examples of flows in programming-language statements are found in [Denning83, Andrews and Reitman80, Gasser88]). Furthermore, one defines a flow policy, such as "if information flows from variable x to variable y, the security level of y must dominate that of x." When applied to specification statements or code, the flow policy helps generate flow formulas. For exampIe, the flow formula of "a: = b" is security level(a) $\geq$ security_level(b). Flow formulas are generated for complete program and TCB-primitive specifications or code based on conjunctions of all flow formulas of individual language statements on a flow path. (Formula simplifications are also possible and useful but not required.) These flow formulas must be proven correct, usually with the help of a theorem prover. If a pro-gram flow formula cannot be proven, the particular flow can lead to a covert channel flow and further analysis is necessary. That is, one must perform semantic analysis to determine (1) whether the unproven flow is real or is a false illegal flow, and (2) whether the unproven flow has a scenario of use (i.e., leads to a real—not just a potential—channel). Example 5 of this section and Examples 7 and 8 of Section 3.3 illustrate the notion of false illegal flow and the distinction between real and potential channels.

Various tools have been built to apply syntactic flow analysis to formal specifications. For example, the SRI Hierarchical Development Methodology (HDM) and Enhanced HDM (EHDM) tools [Feiertag80, Rushby84] apply syntactic analysis to the SPECIAL language. Similarly, the Ina Flo tool of the Formal Development Methodology (FDM) [Eckmann87] and the Gypsy tools [McHugh and Good85, McHugh and Ackers87] have been used for syntactic information-flow analyses. Appendix B reviews these tools. Experience with information-flow analysis in practice is also reported in references [Millen78, MiIlen81].

Syntactic information-flow analysis has the following advantages when used for covert channel identification:

- It can be automated in a fairly straightforward way;
- It can be applied both to formal top-level specifications and source code;
- It can be applied incrementally to individual functions and TCB primitives; and
- It does not miss any flow that leads to covert channels in the particular specification (or code).

All syntactic information-flow analysis methods share the following three drawbacks:

- Vulnerability to discovery of false illegal flows (and corresponding additional effort to eliminate such flows by manual semantic analysis);
- Inadequacy of use with informal specifications; and
- Inadequacy in providing help with identifying TCB locations for placing covert channel handling code.

All syntactic flow-analysis methods assume each variable or object is either explicitly or implicitly labeled with a specific security level or access class. However, as pointed out in [Kemmerer83], covert channels use variables not normally viewed as data objects. Consequently, these variables cannot necessarily be labeled with a specific security level and, therefore, cannot be part of the interpretation of a given nondiscretionary security model in an operating system. Instead, these variables are internal to kernels or trusted processes and their security levels may vary dynamically depending upon flows between labeled objects. Therefore, the labeling of these variables with specific security levels to discover all illegal flows also renders these code-analysis methods vulnerable to discovery of false flow violations. These false flow violations are called "formal flow violations" in references [MilIen78, Schaefer89, Tsai90].

Example 5 - A False Illegal Flow

An example of a false flow violation in the fragment of code shown in Figure 3-1(a) is illustrated in Figures 3-1(b, c). Here, both the alterer and the viewer of the "msgque → mode" variable is the TCB primitive "msgget" of Secure Xenix. The flow formula sl(u.u_rval1) ≥ sl(qp) ≥ sl(msgque → mode) ≥ sl(flag) ≥ sl(uap→msgflg), where sl stands for the security level, cannot be proven because the security levels of the variables vary dynamically, depending on the security levels of the processes invoking the "msgget" primitive. Thus, syntactic flow analysis would identify this flow as il legal. However, an examination of the program conditions under which this flow can actually occur (shown in Figure 3-1 (b)) quickly reveals this flow is legal. This flow can occur because the conditions enabling the flow at run time include security checks of the nondiscretionary model interpretations for both viewing and altering InterProcess Communication (IPC) objects. These checks prevent all illegal flows through the "msgque → mode" variable.

Practical examples of false illegal flows appear in all covert channel analyses relying exclusively on syntactic flow analysis. For example, sixty-eight formulas that could not be proven have been found in the SCOMP analysis using the Feiertag Flow tool [Benzel84, Millen89b]. Only fourteen of these flows caused covert channels; the balance were all false illegal flows. Similar examples can be given based on experience with other flow tools. For instance, even in a small (twenty-line) program written in Ina Jo, the lna Flow tool discovered one hundred-seventeen illegal flows of which all but one were false [Cipher90].

Information-flow analysis does not lend itself to use on informal (e.g., English language) specifications. This means that, if one uses information-flow analysis for B2-B3 class systems, one should apply it to source code. Furthermore, discovery of illegal flows in formal top-level specifications (for class A1 systems) offers little help for identifying TCB locations where covert channel handling code may be necessary. The identification of such locations requires semantic analysis of specifications and code.

```
SYSTEM_CALL
msgget()
{
    struct a  {
            key_t       key;
            int         msgflg;
    }  *uap;
    register struct msqid_ds *qp  /* ptr to associated q */
    uap = (struct a *)u.u_ap;
    if ((qp = ipcget(uap→msgflg, msgque, &s)) = = NULL)
        return;
    if (obj_access(OBJ_IPCGET, msgque, ASK_READ))
        return;
    u.u_rval1 = qp→msg_perm.seq * v.v_msgmni + (qp - msgque);
}

struct ipc_perm *
ipcget(flag, base, status)
int flag, *status;
register struct ipc_perm *base;
{
    if (base→mode & IPC_ALLOC) {
        u.u_error = ENOSPC;
        return(NULL);
    }
    if (obj_access(OBJ_IPCGET, base, ASK_WRITE))
        return;
    *status = 1;
    base→mode = IPC_ALLOC | (flag & 0777);
    return(base);
}
```

(a) A fictitious fragment of code in a "msgget()" system call.

Figure 3-1. An Example of a False Illegal Flow Caused by Syntactic Flow Analysis

3.2.2 Addition of Semantic Components to Information-Flow Analysis

Reference [Tsai90] presents a method for identification of potential storage channels based on (1) the analysis of programming language semantics, code, and data structures used within the kernel, to discover variable alterability/visibility; (2) resolution of aliasing of kernel variables to determine their indirect alterability; and (3) information-flow analysis to determine indirect visibility of kernel variables (e.g., the "msgque → mode" variable in Figure 3-1). These steps precede the application of the nondiscretionary (secrecy or integrity semantic) rules specified in the interpretation of the security model, and implemented in code, to the shared variables and kernel primitives. This last step helps distinguish the real storage channels from the legal or inconsequential ones. The delay in the application of these rules until the security levels of shared variables can be determined with certainty (i.e., from the levels of the objects included in the flows between variables) helps avoid additional (manual) analysis of false illegal flows. Furthermore, discovery of all locations in kernel code where shared variables are altered/viewed allows the correct placement of audit code and time-delay variables for channel-handling mechanisms, and of access checks for nondiscretionary policy implementation.

A disadvantage of this method is that its manual application to real TCBs requires extensive use of highly skilled personnel. For example, its application to the Secure Xenix system required two programmer-years of effort. Thus, using this method in real systems requires extensive use of automated tools. Although the method is applicable to any implementation language and any TCB, its automation requires that different parser and flow generators be built for different languages.

The addition of an automated tool for semantic information-flow analysis to syntactic analysis is reported in [He and Gligor90]. The semantic component of this tool examines all flows visible through a TCB interface and separates the legal from the illegal ones. Since this analysis uses the interpretation of a system's mandatory security model in source code, false illegal flows are not detected. Although one can apply this method to any system, the tool component for semantic analysis may differ from system to system because the interpretation of the mandatory security model in a system's code may differ from system to system. The separation of real covert channels from the potential ones, which requires real scenarios of covert channel use, must still be done manually. Compared to the separation of all potential channels from flows allowing a variable to be viewed/altered through a TCB interface, the separation of real channels from potential channels is not a labor-intensive activity since the number of potential channels is typically several orders of magnitude smaller than the number of flows through a TCB interface.

VARIABLE: **msgque→mode**

KEY:	
$\Rightarrow$	is an explicit flow [Denning77]
$\cdot\cdot>$	is an implicit flow

ALTERER TCB PRIMITIVE: **msgget**

PATH: msgget: (msgflg) $\Rightarrow$ ipcget: (flag)
msgget: (msgque) $\Rightarrow$ ipcget: (base)
ipcget: (flag) $\Rightarrow$ ipcget: (base→mode)

COND: ipcget: !(base→mode & IPC_ALLOC) &&!(obj_access(OBJ_IPCGET, base, ASK_WRITE))

RESULTING FLOW: msgflg $\Rightarrow$ flag $\Rightarrow$ msgque→mode

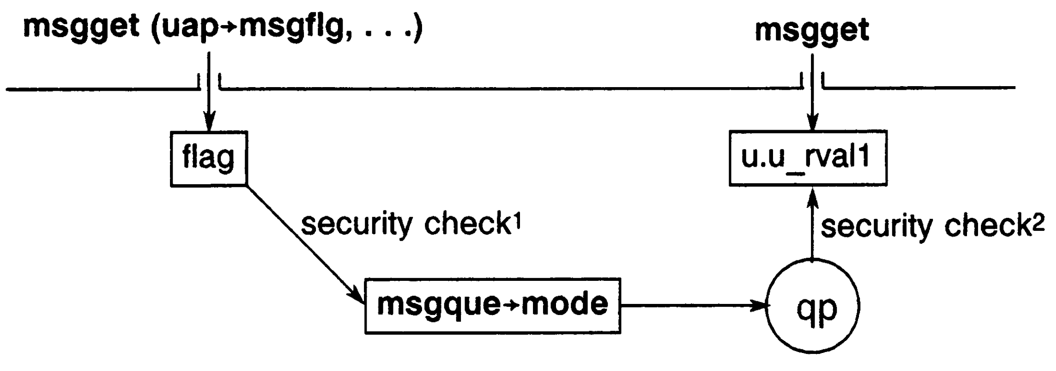
VIEWER TCB PRIMITIVE: **msgget**

PATH: msgget: (msgque) $\Rightarrow$ ipcget: (base)
ipcget: (base→mode) $\cdot\cdot>$ msgget: (qp)
msgget: (qp) $\Rightarrow$ msgget: (u.u_rval1)

COND: msgget: !(qp = NULL) &&!(obj_access(OBJ_IPCGET, msgque, ASK_READ))

RESULTING FLOW: msgque→mode $\Rightarrow$ qp $\Rightarrow$ u.u_rval1

(b) A flow path and flow condition of code fragment in Figure 3-1(a).



security check^1 = !obj_access(OBJ_IPCGET, base, ASK_WRITE)
security check^2 = !obj_access(OBJ_IPCGET, msgque, ASK_READ)

(c) Graphic representation of the flow path of Figure 3-1(b).

Figure 3-1. An Example of a Flase Illegal Flow Caused by Syntactic Flow Analysis

3.2.3 Shared Resource Matrix (SRM) Method

The SRM method for identifying covert channels was proposed by [Kemmerer83], and used in several projects [Haigh87]. When applied to TCB specifications or code, this method requires the following four steps:

(1) Analyze all TCB primitive operations specified formally or informally, or in source code;

(2) Build a shared resource matrix consisting of user-visible TCB primitives as rows and visible/alterable TCB variables representing attributes of a shared resource as columns; mark each <TCB primitive, variable> entry by R or M depending on whether the attribute is read or modified. (This step assumes one has already determined variable visibility/alterability through the TCB interface.) Variables that can neither be viewed nor altered independently are lumped together and analyzed as a single variable. We show a typical shared-resource matrix in Figure 3-2 and discuss it in Example 6.

(3) Perform a transitive closure on the entries of the shared resource matrix. This step identifies all indirect reading of a variable and adds the corresponding entries to the matrix. A TCB primitive indirectly reads a variable y whenever a variable x, which the TCB primitive can read, can be modified by TCB functions based on a reading of the value of variable y. (Note that whenever the SRM method is applied to informal specifications of a TCB interface as defined in system reference manuals—and not to internal TCB specifications of each primitive, which may be unavailable—performing this step can only identify how processes *outside* the TCB can use information covertly obtained through the TCB interface. Therefore, whenever people using the SRM method treat the TCB as a black box, they can eliminate the transitive closure step since it provides no additional information about flows within the TCB specifications or code.)

(4) Analyze each matrix column containing row entries with either an ‘R’ or an ‘M’; the variable of these columns may support covert communication whenever a process may read a variable which another process can write and the security level of the former process does not dominate that of the latter. Analysis of the matrix entry leads to four possible conclusions [Kemmerer83]:

  (4.1) If a legal channel exists between the two communicating processes (i.e., an authorized channel), this channel is of no consequence; label it “L”.

  (4.2) If one cannot gain useful information from a channel, label it “N”.

  (4.3) If the sending and receiving processes are the same, label the channel “S”.

  (4.4) If a potential channel exists, label it “P”.

The labeling of each channel is a useful means of summarizing the results of the analysis.

(5) Discover scenarios of use for potential covert channels by analyzing all entries of the matrix. Examples 7 and 8 of Section 3.2.5 illustrate potential covert channels that cannot be exploited because real scenarios of use cannot be found.

The SRM method has been used successfully on several design specifications [Kemmerer83, Haigh87]. This method has the following advantages:

- It can be applied to both formal and informal specifications of both TCB software and hardware; it can also be applied to TCB source code.
- It does not differentiate between storage and timing channels and, in principle, applies to both types of channels. (However, it offers no specific help for timing channel identification.)
- It does not require that security levels be assigned to internal TCB variables represented in the matrix and, therefore, it eliminates a major source of false illegal flows.

However, lack of security-level assignment to variables has the following negative consequences:

- Individual TCB primitives (or primitive pairs) cannot be proven secure (i.e., free of illegal flows) in isolation. This shortfall adds to the complexity of incremental analysis of new TCB functions.
- The SRM analysis may identify potential channels that could otherwise be eliminated automatically by information-flow analysis.

Although the SRM method is applicable to source code, tools to automate the construction of the shared resource matrix for TCB source code, which is by far the most time-consuming, labor-intensive step, do not exist to date. The manual use of this method on source code—as with other methods applied manually—is susceptible to error.

Example 6 - Inadequacy of Exclusive Reliance on Informal TCB Specifications

The major advantage of the SRM method over syntactic information flow analysis, namely its applicability to informal TCB top-level specifications, is diminished to some extent by the fact that informal top-level specifications lack sufficient detail. We illustrate this observation (1) by showing the results of applying the SRM method to a UNIX TCB specification as found in the Xenix reference manuals [IBM87] using three internal variables of the file subsystem (i.e., “mode,” “lock,” and “file_table”) as the target of CCA, and (2) by comparing this analysis with the automated analysis performed for the same three variables and the Secure Xenix TCB source code with the tool presented in [He and Gligor90].

Figure 3-2 illustrates the results of this comparison. In this figure, the bold-faced matrix entries denote the information added to the original SRM matrix as a result of the automated analysis. This figure shows that about half of the relevant primitives were missed for one of the variables (i.e., “mode”) and a third were missed for another variable (i.e., “file_table”).

Furthermore, more than half of the R/M entries constructed for the primitives found to reference the three variables in system manuals were added R/M designators by the automated analysis of source code. Although different results can be obtained by applying the SRM method to different informal specifications, this example illustrates that the application of SRM (and of any other method) to informal specification can be only as good as the specifications.

PRIMITIVES	SHARED GLOBAL VARIABLES		
	mode	lock	file table
access	*R*		
chmod	**R** *M*		
chsize	**R**	**M**	
close	**R**	**M**	**R** *M*
creat	*R M*	**R M**	*R M*
dup	**R**	**M**	**R**
exec	**R M**	**M**	**R**
fcntl		*R M*	**R M**
fstat	**R M**		
link	**R M**		
locking	**R**	*R M*	
open	*R M*	**R M**	*R M*
read	**R M**		
stat	**R M**		
unlink	**R** *M*		
utime	**R M**		
write	**R M**	**M**	

(1) R = read and M = modify.
(2) Primitives in **Boldface**: primitives that are found to be able to read/modify a shared global variable in the source code but not in the specification (DTLS).
(3) Primitives in *Italics*: primitives that are found to be able to read/modify a shared global variable both in the source code and in the specification (DTLS).
(4) Rules (2) and (3) also apply to the readability/modifiability of primitives to a shared global variable.

PRIMITIVES	SHARED GLOBAL VARIABLES		
	mode	lock	file table
aclcreat			**R M**
aclopen			**R M**
brk	**R M**		
brkctl	**R M**		
chdir	**R M**		
chown	**R M**		
creatsem	**R M**		**R M**
exit	**R M**		**R**
fork	**R M**		**R**
ioctl	**R**		
mknod	**R M**		
nbwaitsem	**R**		
opensem	**R M**		**R M**
pipe	**R M**		**R M**
rdchk	**R**		
sdfree	**R M**		
sdget	**R M**		
seek	**R**		
shmat	**R M**		
shmctl	**R M**		
shmget	**R M**		
sigsem	**R**		
sync	**R M**		
vhangup			**R**
waitsem	**R**		

Figure 3-2. Shared Resource Matrix for Three Variables

3.2.4 Noninterference Analysis

Noninterference analysis of a TCB requires one to view the TCB as an abstract machine. From the point of view of a user process, a TCB provides certain services when requested. A process' requests represent the abstract machine's inputs, the TCB responses (e.g., data values, error messages, or positive acknowledgements) are its outputs, and the contents of the TCB internal variables constitute its current state. Each input results in a (TCB) state change (if necessary) and an output. Each input comes from some particular process running at a particular security level, and each output is delivered only to the process that entered the input that prompted it.

[Goguen and Meseguer82] formulated the first general definition of information transmission in the state-machine view of a TCB, generalizing on an earlier but more restricted definition by [Feiertag80]. They defined the concept of noninterference between two user processes. The definition was phrased in terms of an assumed initial or start-up state for the machine. It stated, in effect, that one user process was noninterfering with another when the output observed by the second user process would be unchanged if all inputs from the first user process, ever since the initial state, were eliminated as though they had never been entered. Goguen and Meseguer reasoned that if inputs from one user process could not affect the outputs of another, then no information could be transmitted from the first to the second. (One can verify this property using Shannon's definition of information transmission [Millen 89b].)

To define noninterference precisely, let X and Y be two user processes of a certain abstract-machine TCB. If w is a sequence of inputs to the machine, ending with an input from Y, let Y(w) be the output Y receives from that last input (assuming the machine was in its initial state when w was entered). To express noninterference, w/X is the subsequence that remains of w when all X-inputs are deleted, or "purged," from it. Then X is noninterfering with Y if, for all possible input sequences w ending with a Y-input, Y(w) = Y(w/X).

It is somewhat unintuitive that noninterference relates a whole sequence of inputs, including, perhaps, many X-inputs, to a single Y-output. In CCA, the traditional view is that whenever a covert channel exists between X and Y, each individual X-input has an effect on the next Y-output. Noninterference analysis suggests another view may be appropriate, however. Note that user process Y might enter an input to request an output at any time. Suppose, in fact, that Y enters an input every time X did. Ignoring other inputs, the overall input sequences looks like: $x_1y_1x_2y_2 \ldots x_ny_n$. The definition of noninterference applies not only to the whole sequence, but to all the initial segments of it ending in a Y-input, namely: (x_1y_1), $(x_1y_1x_2y_2)$, . . . ` $(x_1y_1\ x_ny_n)$. Noninterference requires that *every* Y output is unaffected by *all* previous X inputs. Thus, it seems necessary to analyze all past X inputs because of the following: Suppose each X input is reported as a Y output after some delay; a covert channel arises just as it would if the X input came out immediately in the next Y output.

In practice, it is cumbersome to analyze the entire history of inputs to the machine since its initial state. However, this analysis is unnecessary because the current state has all the information needed to determine the next Y-output. Thus, noninterference of X with Y can be expressed in terms of the current state instead of the whole prior input history.

Clearly, if X is noninterfering with Y, an X input should have no effect on the next Y output. Noninterference is actually stronger than this, however, since it requires that an X input has no effect on *any subsequent* Y output. To avoid analyzing unbounded input sequences, it is useful to partition TCB states into equivalence classes that are not distinguishable using present or subsequent Y outputs. That is, two states are Y-equivalent if (1) they have the same Y output in response to the same Y input, and (2) the corresponding next states after any input are also Y-equivalent. (This definition is recursive rather than circular; this is computer science!) [Goguen and Meseguer84] proved a theorem, called the "Unwinding Theorem," which states that X is noninterfering with Y if and only if each X input takes each state to a Y-equivalent state; a simpler version of this theorem was given by [Rushby85].

Unwinding is important because it leads to practical ways of checking noninterference, especially when given a formal specification of a TCB that shows its states and state transitions. The multilevel security policy requires that each process X at a given security level should interfere only with a process Y of an equal or higher security level. To apply this requirement in practice, the TCB states must be defined, and the Y-equivalent states must be determined. A straightforward way of identifying Y-equivalent states in a multilevel secure TCB is to label state variables with security levels. If Y is cleared for a Security level s, then the two states are Y-equivalent if they have the same values in those state variables having a security level dominated by s. A less formal way of expressing this statement is that Y has (or should have) a blind spot when it tries to observe the current state. Y can observe state variables at or below its own level, but state variables at a higher level are in the blind spot and are invisible. So two states are Y-equivalent if they look the same under Y's "blind spot" handicap.

The state-variable level assignment must have the property that the effect of any input turns equivalent states into equivalent states. This means that invisible variables cannot affect the visible part of the state. This property is one of three that must be proved in a noninterference analysis. The other two properties are that (1) any return values reported back to Y depend only on variables visible to Y, and (2) an input from a higher level user process X cannot affect the variables visible to user process Y.

Noninterference analysis has the following important advantages:

- It can be applied both to formal TCB specifications and to source code;
- It avoids discovery of false illegal flows; and
- It can be applied incrementally to individual TCB functions and primitives.

However, it has three practical disadvantages. First, one can only apply it to formal TCB top-level specifications and, possibly, to source code. Therefore, its application to systems in classes where analyses of formal specifications or source code is not required (i.e., class B2-B3 systems) can only be recommended but not mandated. Only the Al system design, which requires specification-to-code correspondence, can be construed to require covert channel identification on source code (during the specification-to-code correspondence). Second, manual application of noninterference to significant-size TCBs may be impractical, and automated tools are currently unavailable for use in significant-size systems. Third, noninterference analysis is "optimistic." That is, it tries to prove that interference does not appear in TCB specifications or code. Thus, its best application is TCB specifications of trusted-process isolation rather than to TCB components containing large numbers of shared variables (i.e., kernels). Noninterference analysis was used to discover covert channels of the Logical Co-processing Kernel (LOCK)—a successor of the Secure Ada Target (SAT) [Boebert85]. The process of using the Gypsy system to verify noninterference objectives, and the consequences of discovering that a real operating system does not quite attain them, was discussed in reference [Haigh87].

3.3 POTENTIAL VERSUS REAL COVERT CHANNELS

Covert channel identification methods applied statically to top-level specifications or to code produce a list of *potential* covert channels. Some of the potential covert channels do not have scenarios of real use. These potential channels are artifacts of the identification methods. However, false illegal flows do not necessarily cause these potential channels. As illustrated in Figure 3-1(b), all flows have a condition that enables the flow to take place as the system runs (e.g., dynamically). A general reason why a potential covert channel may not necessarily be a real covert channel is that, at run time, some flow conditions may never become true and, thus, may never enable the illegal flow that could create a covert channel. Another reason is that the alteration (viewing) of a covert channel variable may not be consistent with the required alteration (viewing) scenario. For example, a field of the variable may be altered but it could not be used in the scenario of the covert channel. Similarly, not all TCB primitives of a channel can be used in real covert channel scenarios. The ability to use some TCB primitives of a channel to transfer information may depend on the choice of the primitive's parameters and the TCB state. Examples 7, 8, and 9 illustrate these cases. To determine whether a potential covert channel is a real covert channel, one must find a real-time scenario enabling an illegal flow.

Example 7 - An Example of a Potential Covert Channel

Figure 3-3(a) illustrates the difference between potential and real covert channels. Two UNIX TCB primitives "read" and "write" share the same internal function "rdwr" but pass different values to the parameter of this function. CCA on the internal function "rdwr" reveals all possible information flows within "rdwr" (i.e., both flows that lead to real channels and flows that only lead to potential channels). Among the latter are flows with the condition "mode = FWRlTE." These flows cannot be exploited by TCB primitive "read" because it can never enable this condition. Similarly, TCB primitive "write" cannot exploit those flows with the condition "mode = FREAD."

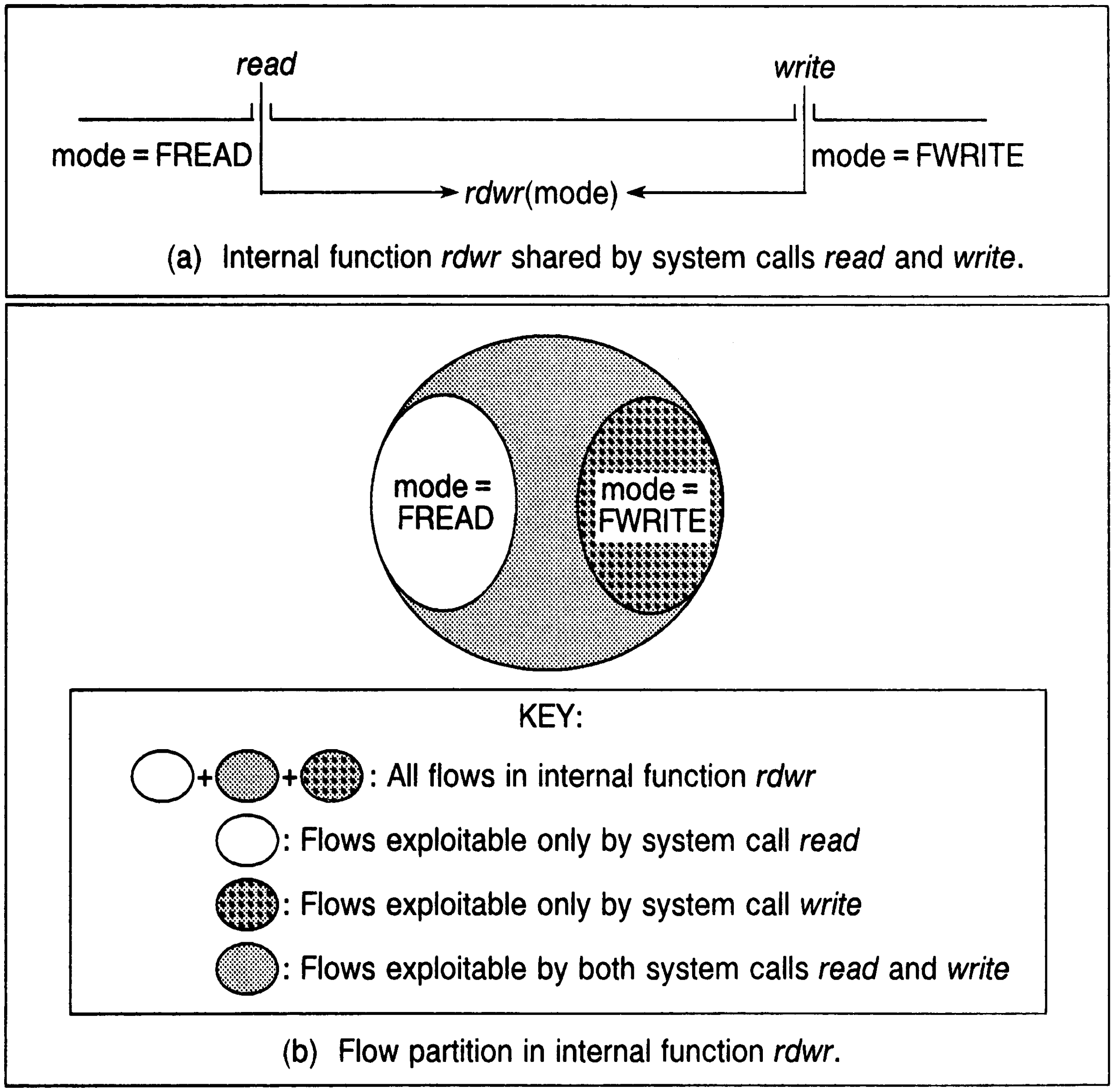

read
write
mode = FREAD
mode = FWRITE
rdwr(mode)
(a) Internal function rdwr shared by system calls read and write.
mode = FREAD
mode = FWRITE
KEY:
: All flows in internal function rdwr
: Flows exploitable only by system call read
: Flows exploitable only by system call write
: Flows exploitable by both system calls read and write
(b) Flow partition in internal function rdwr.

Figure 3-3. Potential and Real Covert Channels Corresponding to Different Flow Partitions

Thus, among the potential covert channels arising from the invocation of the internal function "rdwr," those with condition "mode = FWRITE" cannot be real covert channels for the "read" primitive, and those with the condition "mode = FREAD" cannot be real covert channels for the "write" primitive. Real-time scenarios do not exist for those potential covert channels. Figure 3-3 (b) shows the partitioning of flows in internal function "rdwr" based on whether a flow can be exploited by the "read" and "write" primitives.

TCB Primitive \ Channel Var	File Table	Inode Table	Disk Space	Message ID Table	Process ID Table	Text Table
creat	AV	AV	AV			
exec			AV			AV
fork	a	a	AV		AV	
msgget				AV		
msgctl				A		
open	AV	AV	AV			
wait					A	

(a) Examples of resource-exhaustion channels in Secure Xenix™

TCB Primitive \ Channel Var	Number of Free Blocks	Number of Free Inodes	Message Identifier	Process Identifier
creat	A	A		
chsize	A	a		
fork	A	a		AV
msgget			V	
msgctl			A	
ustat	V	V		

(b) Examples of event-count channels in Secure Xenix™

Figure 3-4. Examples of Potential and Real Channels

Example 8 - Real and Potential Covert Channels in Secure Xenix

Figure 3-4 illustrates examples of real and potential covert channels of Secure Xenix. The two tables shown in Figure 3-4 contain two basic types of covert storage channels: resource-exhaustion and event-count channels. Resource-exhaustion channels arise wherever system resources are shared among users at more than one security level. To use a resource-exhaustion channel, the sending process chooses whether or not to exhaust a system resource to encode a signal of a 0 or 1. The receiving process detects the signal by trying to allocate the same system resource. Depending on whether the resource can be allocated, the receiving process can determine the value of the signal from the sending process.

In event-count channels, the sending process encodes a signal by modifying the status of a shared system resource (but not exhausting the resource). By querying the status of the resource, either through TCB primitives returning the resource status explicitly or by observing the return result of some TCB primitives that allocate the resource, the receiving process can detect the signal from the sending process.

In the tables of Figure 3-4, each row is marked with a TCB primitive and each column with a shared global variable. Entries in the tables indicate whether a TCB primitive can alter (A or a) and/or view (V or v) the corresponding global variable. An upper-case A in an entry indicates that the TCB primitive can alter the global variable as the means of encoding and transferring information through the variable. Similarly, an upper-case V in an entry indicates that the TCB primitive can view the value of the global variable and detect the signal transmitted by sending user processes. Thus, for any shared global variable, the set of TCB primitives that have a capital A and those that have a capital V constitute a real covert channel. For example, TCB primitives "creat" and "open" can be used by the sending and the receiving processes to transfer information through the shared global variable representing the "file_table" in the system. On the other hand, a lower-case a in an entry means that, although the TCB primitive can alter the global variable, the alteration cannot be used for encoding information in the global variable. For example, the execution of the TCB primitive "fork" alters the "file_table" because it increments the file reference count for the child process it creates. This alteration, however, is different from that of allocating a file-table entry and, thus, it does not provide a real scenario for sending a signal. Similar reasoning explains why the entries marked with a lower-case v in Figure 3-4 cannot be used in real scenarios of covert channels.

The distinction between an alteration of a global variable that can be used to encode information (i.e., entry denoted by A) and one that cannot (i.e., entry denoted by a) can be eliminated if a finer partitioning of the "file table" structure is performed. That is, if the file reference count of the "file_table" is viewed as a separate variable within the "file_table" structure, then the TCB primitive "fork" would not appear to alter the "file_table" structure. Instead, "fork" would alter only the new variable, namely, the file reference count. In either case, however, the covert channel analysis should yield the same results.

Example 9 - Dependencies on TCB State and Primitive Parameters

The covert channel examples of Figure 3-5 illustrate both system-state and parameter dependencies found in UNIX systems. For example, the primitive "creat" can alter (i.e., decrement) the total number of free inodes (nfi) only if the object to be created does not exist. If the object exists, "creat" has no effect on nfi. In addition, the primitive "creat" can be used to alter (i.e., increment) the total number of free blocks (nfb) in the system if the file being created currently exists. That is, if the file exists, "creat" truncates the file, and as a result increments nfb. Otherwise, "creat" has no effect on nfb. (The disk-block-space channel is also affected by this condition.) Furthermore, the alteration of the disk-block-space channel, and of the nfi and nfb channels by the primitive "creat," is determined by the file system specified in the parameter of the "creat" invocation.

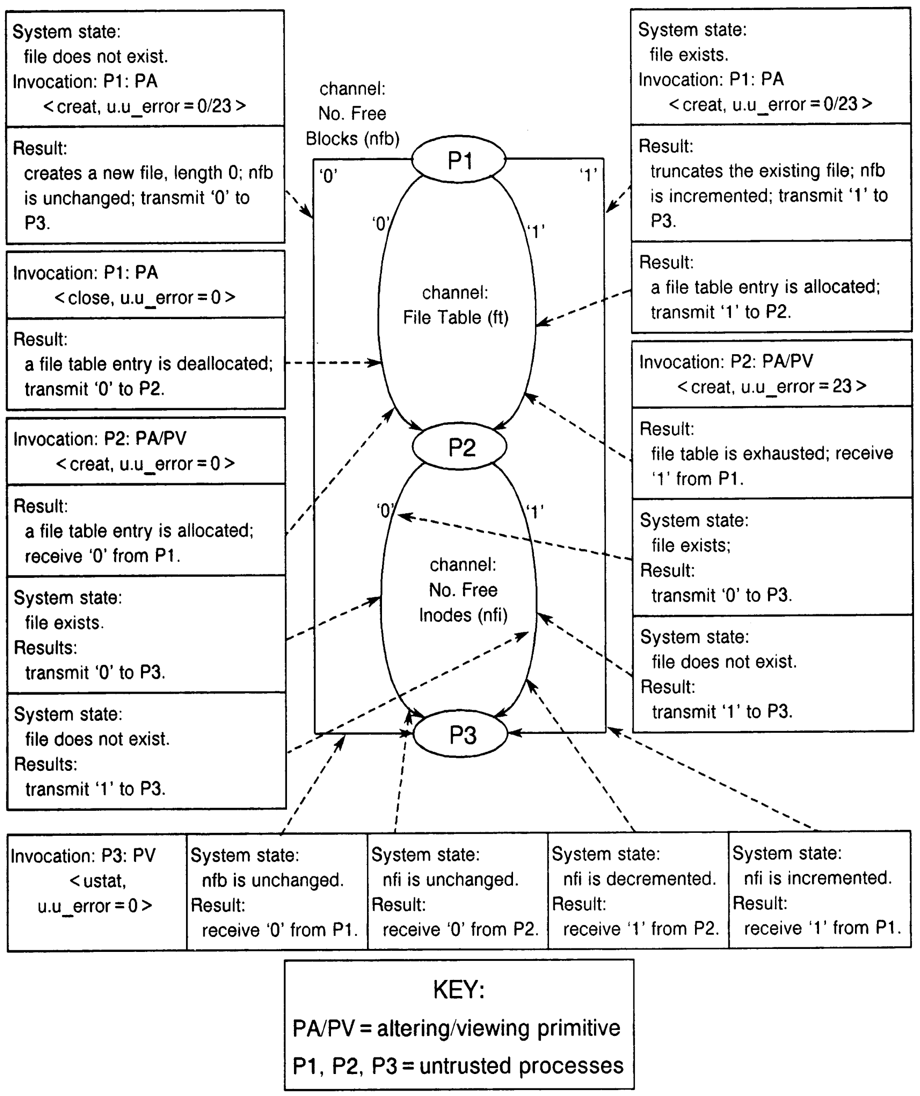

System state: file does not exist. Invocation: P1: PA < creat, u.u_error = 0/23 >
Result: creates a new file, length 0; nfb is unchanged; transmit '0' to P3.
Invocation: P1: PA < close, u.u_error = 0 >
Result: a file table entry is deallocated; transmit '0' to P2.
Invocation: P2: PA/PV < creat, u.u_error = 0 >
Result: a file table entry is allocated; receive '0' from P1.
System state: file exists. Results: transmit '0' to P3.
System state: file does not exist. Results: transmit '1' to P3.
channel: No. Free Blocks (nfb)
P1
P2
P3
'0'
'1'
channel: File Table (ft)
channel: No. Free Inodes (nfi)
System state: file exists. Invocation: P1: PA < creat, u.u_error = 0/23 >
Result: truncates the existing file; nfb is incremented; transmit '1' to P3.
Result: a file table entry is allocated; transmit '1' to P2.
Invocation: P2: PA/PV < creat, u.u_error = 23 >
Result: file table is exhausted; receive '1' from P1.
System state: file exists; Result: transmit '0' to P3.
System state: file does not exist. Result: transmit '1' to P3.
Invocation: P3: PV < ustat, u.u_error = 0 >
System state: nfb is unchanged. Result: receive '0' from P1.
System state: nfi is unchanged. Result: receive '0' from P2.
System state: nfi is decremented. Result: receive '1' from P2.
System state: nfi is incremented. Result: receive '1' from P1.
KEY:
PA/PV = altering/viewing primitive
P1, P2, P3 = untrusted processes

Figure 3-5. An Example of the Use of Multiple Channels by Three Processes

The example of Figure 3-5 also illustrates the combined state and parameter dependencies. Consider again the channel that modulates the nfb and the disk-block-space channel. Primitive “chsize” can be used to alter these channel variables (i.e., deallocate memory and increase the total number of free blocks) only if the file on which it is applied exists, and only if its parameter indicates file shrinkage. When used to expand the size of an existing file, primitive “chsize” does not alter the channel variables but merely changes the ip-i_size field of the inode.

Other examples of parameter dependency and combined state and parameter dependencies, unrelated to those of Figure 3-5, can be found. For example, the primitive “semget(key, nsems, semflg)” can affect the semaphore-identifier channel and the semaphore-map exhaustion channel. Within this primitive, if parameter “key” is equal to IPC_CREAT, thereby denoting the creation of a semaphore, a semaphore identifier, its associated semaphore data structure, and a set containing “nsems” semaphores are created for key. In contrast, if parameter key is not equal to IPC_CREAT, nothing is created.

Furthermore, if parameter key does not already have a semaphore identifier associated with it, and if the Boolean expression (semflg and IPC_CREAT) is true, a “semget” call creates for parameter key a semaphore identifier, its associated data structure, and the set containing “nsems” semaphores. If parameter key already has a semaphore identifier associated with it, a new semaphore structure is not created.

3.4 *TCSEC* REQUIREMENTS AND RECOMMENDATIONS

Covert channel identification requirements appear for the classes B2-A1 of the [NCSC TCSEC]. The B2 requirements of CCA state that the “system developer shall conduct a thorough search for storage channels....”

For class B2, the search for covert storage channels should be conducted on system reference manuals and on the DTLS of the TCB. Although the *TCSEC* does not require storage channel identification in the TCB source code and in hardware (microcode) specifications, such a search would ensure the completeness of the identification results. Although no specific identification method is required, arbitrary, ad hoc, undocumented methods, which system evaluators cannot repeat on independently selected test cases, are unacceptable. This nonacceptance is justified by the notion that system developers must conduct a thorough search for covert channels and an evaluation team must evaluate the search.

Use of any identification method on informal top-level specifications may yield incomplete results, as illustrated in Example 6 of this section in the context of the SRM method. For this reason, it seems important to apply the storage channel identification method to the TCB source code. Otherwise, the thoroughness of the covert channel identification search may be in doubt. Furthermore, source-code analysis may be useful in the definition of covert channel scenarios that help distinguish real and potential channels. Therefore, we recommend analyzing both the DTLS and the source code for covert channel identification at class B2.

The B3-class requirement of the *TCSEC* extends the above B2-class requirement to all covert channels (i.e., to timing channels). Although this extension imposes no added requirement in terms of the identification method used, timing channel scenarios should be developed. These scenarios should include all system sources of independent timing such as the CPU and the I/O processors.

Inclusion of all these sources will provide additional assurance that classes of timing channels are not overlooked. [Huskamp78] provides an example of complete timing channel analysis for the CPU scheduling channels.

The A1-class requirement of CCA includes all the B2-B3-class requirements and extends them by stating, "Formal methods shall be used in analysis."

One may apply CCA methods to both formal specifications and source code of the TCB. Examples of such methods include syntactic information-flow analysis (with or without the use of semantic analysis), SRM, and noninterference analysis. Other formal methods for covert channel identification may exist and may be equally suitable at level A1. The identification method chosen by the developer should be applied to the FTLS. Unless the identification of covert channels is made a part of the specification-to-code correspondence, in which case source-code analysis is included, we recommend complementing FTLS analysis with formal or informal source-code analysis. Otherwise, covert channels may remain undetected.

4.0 COVERT CHANNEL BANDWIDTH ESTIMATION

In this chapter we discuss various factors that affect the covert channel bandwidth computation, including TCB primitive selection, parameter and state dependencies, and channel aggregation. We also present both information-theory-based and informal methods for maximum bandwidth estimation, and discuss various factors that degrade the covert channel bandwidth. The TCSEC requirements and recommendations are also discussed.

4.1 FACTORS AFFECTING THE BANDWIDTH COMPUTATION

The computation of covert channel bandwidths is one of the key aspects of covert channel analysis. This is the case because most decisions about how to handle identified channels rely on the determination of channel bandwidth. Therefore, it is important to examine briefly the factors that primarily affect the computation of covert channel bandwidth.

4.1.1 Noise and Delay

Two of the most important factors affecting the bandwidth of a covert channel in any operating system or hardware platform are the presence of noise and the delays experienced by senders and receivers using the channel. The primary sources of noise and delay are the processes $U_p, \ldots, U_q$ shown in Figure 2-5, which can interpose themselves due to scheduling of various hardware resources between senders and receivers. Although these processes can degrade the maximum attainable bandwidth of a channel significantly (e.g., up to about 75% [Tsai and Gligor88]), the degradation is not certain in all architectures and at all times since it depends on the nature of the multiprogrammed system (e.g., single user, multiprocess workstation, multiuser time sharing system) and on the system load. Thus, while the noise and delay factors are significant, the computation of the maximum attainable bandwidth of any channel must discount both noise and delays, and must assume that only the senders and receivers are present in the system [Millen89a].

4.1.2 Coding and Symbol Distribution

In general, the attainable maximum bandwidth (i.e., capacity) depends on the choice of symbol encoding scheme agreed upon by a sender and a receiver. Coding schemes exist that allow the exploitation of the maximum attainable bandwidth of a channel on the distribution of symbols in the space of transmitted messages [Millen89a]. However, informal covert channel analysis usually assumes a 0 or a 1 represents each symbol transmitted. Thus, the distribution of 0s and 1s becomes an important factor of bandwidth computation whenever using informal methods. Where the time required to transmit a 0 is close (on the average) to the time required to transmit a 1, one can assume that 0s and 1s are used with approximately equal frequency. This assumption is based on the fact that the bandwidth (i.e., capacity) of discrete memoryless channels is maximized under such distributions. (Section 4.2 below illustrates both an informal bandwidth-computation method, where such distributions are assumed, and an information-theory-based method, where such distributions are not assumed.)

Informal bandwidth computation methods do not achieve, in general, the maximum bandwidth of a channel because they do not use appropriate coding techniques. Formal bandwidth-computation methods not only allow the precise determination of attainable maximum bandwidth but also help define coding schemes that can be used to attain those bandwidths [Millen89a].

4.1.3 TCB Primitive Selection

In most systems, covert channel identification associates multiple TCB primitives with a covert channel variable. For example, most UNIX covert channel variables can be altered or viewed by a number of primitives that varies between about ten and forty. Among the primitives of each variable, one must select those having the highest speed for the bandwidth computation. Although one should measure each primitive's speed with only senders and receivers using the system, one should not conduct these measurements independently of the covert channel scenario of use (i.e., without using parameters and TCB state conditions that would be present when a channel is in use). Otherwise, the bandwidth computation could lead to unrealistically high or low values. Low values may cause security exposures whereas high values may cause performance degradation whenever delays are used based on bandwidth values. We can illustrate the latter case by the "chsize primitive of UNIX. The speed of this primitive depends on whether a file is shrunk (low speed) or expanded (higher speed). However, the use of "chsize" with the expand option cannot be made in covert channels requiring disk free block alteration because this primitive does not alter the disk free block variable. We discuss this in more detail in the next section.

4.1.4 Measurements and Scenarios of Use

The performance measurements of the TCB primitives of covert channels require one to include not only the altering and viewing primitives but also the performance of the primitives that initialize the environment for the altering and viewing of a variable. The environment initialization primitives may differ for the altering and viewing primitives. For example, the environment initialization primitive for altering a variable to transmit a 1 may differ from that necessary to transmit a 0. Similarly, the environment initialization primitives for viewing a 1 may differ from those necessary for viewing a 0. Furthermore, the same primitive may use different amounts of time depending upon whether it is used to set (read) a zero or a one (e.g., whether it returns an error). Scenarios of covert channel use are needed to determine which environment initialization primitives must be taken into account. Section 4.2 provides examples of different environment initialization primitives and their use for two real covert channels of UNIX.

Also included in the measurements is the process- or context-switching time. The measurement of this time is needed because, during the transmission of every bit of information through a covert channel, control is transferred between the sender and receiver at least twice. In most operating systems, the measurement of the minimum process switching time is a fairly complex task. The reason for this complexity is that with every process switch the measurement environment changes and, therefore, the measurement of each switch may yield different values. Sometimes it is also difficult to measure individual process-switching times because process switching may be possible only as a side-effect of some primitives. Other processes may be scheduled to run during a switch from one process to another, thereby adding unwarranted delay to switching time. To eliminate the difference between measured process-switching times within the same system, one must ensure that only a few processes are present in the system when taking measurements and repeat the measurements a large number of times (e.g., a hundred thousand times) to ensure choosing the minimum value.

Real scenarios of covert channel use include sender-receiver synchronization. This synchronization delays the covert channel and, therefore, decreases the channel's bandwidth. However, since one cannot predict the synchronization scenario (because it is privately agreed upon by the sender and receiver), we generally assume the bandwidth decrease caused by synchronization is negligible. This assumption helps ensure computing the maximum bandwidth.

All primitive measurements and process-switching time measurements must be repeatable. Otherwise, independent evaluators cannot verify the bandwidth computations.

4.1.5 System Configuration and lnitialization Dependencies

TCB primitive measurements and process switching times depend very heavily on a number of system architecture parameters. These parameters include:

- System-component speed (e.g., disk, memory, and CPU);
- System configuration (e.g., configurations using or not using caches);
- Configuration-component sizes (e.g., memory sizes, cache sizes); and
- Configuration initialization.

The least obvious dependency is memory size. The same measurement on two systems configured identically but using different memory sizes may yield different results. For example, in systems with smaller memory the primitives will appear to be slower due to the additional swapping and buffer management necessary to accommodate the measurement environment. Similarly, to ensure repeatable results, one must properly initialize the measurement environment.

4.1.6 Aggregation of Covert Channels

In general, both serial and parallel aggregation of distinct covert channels can increase the effective bandwidth available to senders and receivers for covert transmission of information. The easiest way to approximate the effect of aggregation on the maximum channel bandwidths is to (1) set the context-switching time to zero for both serial and parallel aggregation, and (2) to sum the bandwidths of the individual channels for parallel aggregation. However, the *TCSEC* requirements and guidelines neither require nor recommend that one consider aggregation in covert channel analysis. However, one needs to consider the notion of channel aggregation in the area of threat analysis, whenever such analysis is performed in the environment of system use (see Section 5.4 below).

4.1.7 Transient Covert Channels

Transient covert channels are those which transfer a fixed amount of data and then cease to exist. Normally, bandwidth and capacity calculations apply only to channels that are sustainable indefinitely. Thus, it would seem transient channels are an irrelevant threat. However, if a large volume of data can be leaked through a transient channel, one must consider channel bandwidth analysis and handling, for the threat of channel use becomes real.

4.2 BANDWIDTH ESTIMATION METHODS

4.2.1 Information-Theory-Based Method for Channel-Bandwidth Estimation

Millen presents in [1989a] a method based on Shannon’s information theory [Shannon and Weaver64]. In this method, one assumes the covert channels are noiseless, no processes other than the sender and receiver are present in the system during channel operation, and the sender-receiver synchronization takes a negligible amount of time. These assumptions are justified if the goal is the computation of the maximum attainable bandwidth. With these assumptions, one can model most covert channels that arise in practice as finite-state machines (graphs). Furthermore, these graphs are deterministic in that for any state transition corresponding to a given channel symbol (e.g., 0 or 1), only one next state appears in the graph. Figure 4-1 illustrates a state graph for a two-state channel. Most covert channels of interest in practice can be represented with two-state graphs. This is because, for most channels, the current state of the channel depends on the last signal sent and, thus, only two states are necessary to capture the scenario of information transfer.

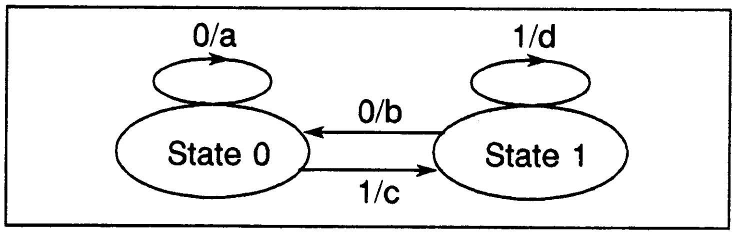


Figure 4-1. Two-State Graph for a Covert Channel

Example 10 - Scenario for a Two-State Covert Channel

A scenario for the transfer of 0s and 1s using a two-state graph can be defined by associating each transition of the graph with the transfer of a 0 or a 1. Each transition covers a sender's action followed by a receiver's action.

For example, to send 0 in state 0:

- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 0;
- receiver resets the environment (if necessary);
- receiver transfers control to sender.

To send 0 in state 1:

- sender sets the channel variable to 0;
- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 0;
- receiver resets the channel environment (if necessary);
- receiver transfers control to sender

To send 1 in state 0:

- sender sets the channel variable to 1;