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 $I(Sh)$ and $I(Si)$ of $I(M)$ is covert if and only if any communication between the corresponding subjects $Sh$ and $Si$ 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 Sh and Si. Communication in either direction, from Sh to Si and vice versa, is prohibited. In contrast, consider a multilevel security model, M', where messages from Sh to Si are allowed only if the security level of Si dominates that of Sh. Here, some communication between Sh 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 Sh to Si 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, covert channel analysis 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.
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 IL2, 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.

In such a system, whenever a user attempts to remove an upgraded directory from level Lh > Li where he is authorized to read and write it, the remove operation fails because it violates the mandatory authorization check (the level of the removing process, Lh, must equal that of the parent directory, Li). In contrast, the same remove operation invoked by a process at level Li < Lh succeeds.

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 Li and tries to remove the upgraded directory from the higher level Lh, the user process can discover whether any files or directories at level Lh > Li are linked to the upgraded directory. Thus, another process at level Lh can transmit a bit of information to the user process at level Li < Lh by creating and removing (e.g., unlinking) files in the upgraded directory.

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.

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. 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.

**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 crossbar-switch architectures. 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 instruction, 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 memory-referencing 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.

### 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 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". 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].

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 (PAh) and the other for viewing (PVi) 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; PAh, PVi>). 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 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.

However, total reliance on analysis of top-level specifications for the identification 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.

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 -> 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 example, the flow formula of "a := b" is security_level(a) > 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 program 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).

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 [Milen78, Schaefer89, Tsai90].

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. 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.

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 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.

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
(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.

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.

### 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's 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).

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 AI 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).

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. 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.

As said above, this is just a small part of the real document. Whole sections as well as single paragraphs and sentences have been removed from the original. The parts chosen are those explaining and illustrating the concepts relevant for the course.