Covert Channel Analysis: A Chapter of the *Handbook for the Computer Security Certification of Trusted Systems*[1]

John McHugh, Principal Investigator

Tektronix Professor
Department of Computer Science
Portland State University
POB 751
Portland, Oregon, 97207-0751

Revised Final Report – 16 December 1995

[1]Prepared by the University of North Carolina for the Naval Research Laboratory under contract N00014–91–K–2032

Contents

Section 1

Introduction

Covert channel analysis can be viewed either as an arcane aspect of computer security having little to do with "real" security issues or as the key to protecting nominally secure systems against a wide variety of both internal and external threats. Which view should be adopted is a function of a number of factors including the nature of the material to be protected: its sensitivity, size, and timeliness, and the threat environment to which the system will be exposed. Under the TCSEC [28], covert channel analysis is required starting at the B2 level of assurance with increasingly rigorous analysis required for B3 and A1 systems. This is consistent with the view that high assurance security systems are primarily required to protect highly sensitive information, to counter serious threats, or both.

In the introduction, we will provide an overview of covert channel analysis, beginning with a definition of covert channels and a discussion of the nature of the issues concerning them that affect the various users of this handbook. Since covert channels involve (often complex) coding and signaling mechanisms, these will be discussed. The introduction concludes with a characterization of covert channels.

Covert channel analysis can, and should, be performed on system descriptions ranging from abstract models to machine code. Section 2 discusses issues of system representation and the suitability of various representation paradigms for covert channel analysis. Since covert channels are built from information flows within a trusted computing system, one of the first steps in performing an analysis is the abstraction of potential information flows from a description of the system. A formalism for performing this task, called dependency analysis is presented in section 3. Dependency analysis provides a general framework in which any suitable system representation can be characterized, providing one of the inputs for a covert channel analysis.

Covert channels can be either innocuous or harmful. Innocuous channels are consistent with the intent of system's security policy. They may result in surprising system behaviors, but do not place the system or the information that it protects at risk. Harmful covert channels are information flows that are contrary to the intent of the system's security policy. In section 4 the problem of extending the system security policy to cover the information flows revealed by dependency analysis is considered.

Given an extended security policy applied to system dependencies, a search for covert channels can be performed. This search, which is discussed in section 5, can be done using

manual methods such as the shared resource matrix, or it can be done using mechanical tools based on information flow formulas (also known as security verification conditions), covert flow trees, or other techniques. It can also be done using "non-interference" formulations that arguably characterize both high level security and covert channels.

If the search for covert channels exposes a flaw in the system, the flaw must be analyzed to determine its potential significance. This includes the development of scenarios for exploiting the flaw, threat analysis, the determination of the channel capacity, etc. Countermeasures ranging from auditing the use of the channel to restructuring the system may apply. Section 6 covers this analysis.

By the end of section 6, the evaluator will have a good view of the techniques that are useful for performing a covert channel analysis on a system under construction and, we hope, some insight into techniques for designing the system so as to minimize the introduction of covert channels. Section 7 provides guidance for purchasers and evaluators in evaluating a covert channel analysis plan. This should be of interest to developers as well. The handbook proper concludes with an annotated bibliography that provides the interested reader with a means to gain further knowledge of the subject. Although the handbook is primarily intended for the evaluators of B2 or B3 systems where mechanical covert channel analysis tools are not required, readers should be aware of the tools that have been developed to support the verification systems used for A1 systems as well as some experimental tools discussed in the recent literature. These are discussed in Appendix A. A small example of a manual covert channel analysis using the shared resource matrix is presented in Appendix B.

1.1 What is a Covert Channel?

Marv Schaefer, former chief scientist of the National Computer Security Center, once characterized covert channels [35] as system behaviors that surprise the system's developers. We offer the following as a working definition of the term "Covert Channel."

> A covert channel is a mechanism that can be used to transfer information from one user of a system to another using means not intended for this purpose by the system developers.

This definition is deliberately vague (What is information? What is intent?) and omits any discussion of security. The definition results from the desire to have an intuitive notion of a secure computer system that is easily understandable to a user accustomed to manual procedures and to present unobvious flaws in the system as special cases. Whether or not this is the best approach to the problem is open to debate, but it is traditional and widely accepted.

In the pen and paper world, access to sensitive information is controlled by people who are trusted to give the information (usually in the form of documents) only to persons whose identity and whose authorization to obtain the information have been established by an appropriate authority. In a secure computer system, the information is typically stored in files or associated with devices such as communication lines. Once the user's identity and authorization are established, the user's access (actually access by programs the user runs) to the information is mediated by the trusted portion of the system. This view

leads to relatively simple abstractions of security such as the Bell and La Padula model in which the system's security state is characterized by a matrix that shows the nature of the access permission that active, information free, entities called "subjects" have to passive, information containing, entities called objects. In this model, the system is secure if all subjects have clearances that are higher than (dominate) the classifications of the objects to which they have read or observe access (the simple security property) and if the classification of each object to which a subject has write or modify access dominates the classification of all objects to which the subject has read access.

While this model is intuitive and captures the essence of the pen and paper world, it is not adequate. Let us suppose that we have given a reader a classified document and locked both in a room. Now, suppose that the reader sends the content of the document to an uncleared confederate in the next room by tapping Morse code on his room's radiator. This is, in effect, a covert channel since the radiator and its pipes were never intended for communication by the building's designers. While this scenario is somewhat far fetched (Why not just tell the confederate later? The cleared user is violating trust, etc.), its analog in the computer system is not. Although the user is cleared and trusted, there is no assurance that the programs the user uses to access and process classified data are trustworthy. As we will see, most systems have features such as file locks, device status bits, a memory pool, etc. that are shared between users at different levels. Each such feature or resource has attributes that can often be manipulated to form the basis for a covert channel. If we posit malicious, untrusted software, used by trusted users, covert channels that signal information in violation of the intent of the system security policy using mechanisms not intended for explicit information transfer are a distinct possibility.

1.2 Why Consider Covert Channels?

For a system to be evaluated at the B2 level of the TCSEC or above, a covert channel analysis is required. For some, this alone may be a sufficient reason to be concerned. The more curious will probably want to probe deeper. The following discussions will indicate the concerns that apply to the various communities of interest that use this handbook.

1.2.1 Evaluators

Evaluators require a thorough understanding of covert channels. They also need the ability to judge the job that the developer has done in dealing with covert channels in both the design and analysis phases of the effort. In addition to evaluating the developer's efforts in analyzing a system for covert channels, the evaluator should also consider the appropriateness of design decisions made by the developer and their impact on system security. Since current practice under the TCSEC divides security into two areas, access control, which is mediated by the system's MAC and DAC policies and covert channels, which are unmediated, an inappropriate choice of "objects" can result in serious system insecurities being labeled as "covert channels." In a system for which covert channel analysis is required, this may be not be serious but it can lead to a false sense of assurance in a B1 system.

1.2.2 The Designated Accreditation Authority (DAA)

In an accreditation process, it is necessary to consider both threats against the system and the nature of the information to be protected as well as the vulnerabilities of the system being examined. The DAA must weigh the operational risks associated with the deployment of a less than perfect system. In particular, the DAA must be able to evaluate the mission risks that arise due to the deployment of a system that contains exploitable insecurities in the form of covert channels.

1.2.3 Developers

Developers of high assurance systems for which covert channel analysis is required should have a thorough understanding of both the mechanisms that make covert channels possible, particularly the nature of resource sharing, and of the analysis techniques that are available to identify them. While it is difficult to eliminate all covert channels in a complex multilevel secure system, careful design supported by early analysis can minimize the impact of covert channels. Developers should learn to think in terms of potential covert channels in making design decisions. Any design decision that involves the sharing of some system resource among users at different security levels has the potential for introducing a covert channel into the resulting system. The designer should be aware of this potential and should take steps to control it in ways that can reduce its impact on overall system security.

1.2.4 Purchasers

Procurement officers and other specifiers of purchases of multilevel secure systems need to be aware of the nature of covert channels and the mechanisms that permit them in order to match the system procured with its mission. This analysis needs to take into account the nature of the threats to which the system will be exposed as well as the nature of the material that it will protect and the range of security levels over which it must operate. In considering covert channel issues, it is important to consider the quantity of information that must be compromised to cause a serious breach of security. Systems in which a compromise of a few hundred bits (say a master encryption key) is serious are much less tolerant of covert channels than systems in which megabytes (say a high resolution image) must be compromised for a serious problem to exist.

1.2.5 End users

End users need to have an understanding of both the security mechanisms and the possible vulnerabilities of the systems that they use. This is just as important for high assurance systems where users should be alert for anomalous behaviors that may indicate security compromises as it is for low assurance systems where users should be aware of the system's limitations. Users should be aware that in the case of B1 systems, for which covert channel analysis is not required, developers may try to categorize serious security flaws as covert channels to avoid dealing with them. Users of low assurance systems such as compartmented mode workstations should be aware of their limitations[6].

1.3 Coding and Signaling

"One if by land, two if by sea." Covert channels involve messages transmitted through mechanisms not normally intended for communication. This typically requires some kind of encoding. If, as was the case with Paul Revere, the number of messages is limited, a very simple encoding may be used. In the general case, the message vocabulary may be open ended, and a general encoding must be developed. The signaling mechanism that is used to transmit the encoded message will involve the manipulation of some attribute of a resource that is shared between the sender and receiver.

Typically, the real users of a covert channel are computer programs that have been subverted for this purpose. The human user who invokes the program is unaware that sensitive information is being compromised by the program. Because the compromise is done by the program rather than a human, it can proceed at electronic, rather than human speeds. This means that, even if an elaborate sequence of actions is required to transmit a single bit of information, it is possible to transmit hundreds or thousands of bits per second with high reliability. In a shared, uniprocessor environment, context switching time is often the most important factor limiting covert signaling rates. In multiprocessor systems, this may not be a factor, and mechanisms that support covert signaling rates on the order of hundreds of megabits per second have been posited[38].

1.4 Characterizing Covert Channels

Covert channels can be characterized in a variety of ways, based on the mechanisms that they use, the level of abstraction at which they appear, or on the nature of the sender and receiver.

1.4.1 Storage vs. Timing

Traditionally, covert channels are characterized as storage or timing channels. As we will see, this characterization is somewhat artificial, but it it useful because it describes the focus of the analysis used to identify the channels. Given that a covert channel involves the manipulation of some system resource or resource attribute (such as whether or not a file is in use), we can characterize a covert channel as a storage channel if the recipient of the signaled information perceives the signal as a change in the value of the shared resource or attribute. Similarly, if the information is perceived via a change in the time required for the recipient to perform some action, the channel may be characterized as a timing channel.

Kemmerer [13] gives the following definitions of storage and timing channels:

> "In order to have a storage channel, the following minimum criteria must be satisfied:
>
> 1. The sending and receiving processes must have access to the same attribute of a shared resource.
> 2. There must be some means by which the sending process can force the shared attribute to change.

3. There must be some means by which the receiving process can detect the attribute change.
4. There must be some mechanism for initiating communication between the sending and receiving processes and for sequencing the events correctly. This could be another channel with a smaller bandwidth.

. . .

The minimum criteria necessary in order for a timing channel to exist are as follows:

1. The sending and receiving processes must have access to the same attribute of the shared resource.
2. The sending and receiving processes have access to a time reference, such as a real–time clock.
3. The sender must be capable of modulating the receiver's response time for detecting a change in the shared attribute.
4. There is some mechanism for initiating the process and for sequencing the events."

Implicit in the timing channel criteria is the ability of either the sender or the receiver to change the value of the shared attribute used to implement the timing channel.

Since high level specifications typically do not contain timing information, the covert channel analysis typically performed on the DTLS or FTLS of a system is a storage channel analysis. Timing channel analyses are typically much more ad hoc in nature since timing behavior is most often manifest in code execution timing or in device response times, factors that are usually not considered at the specification level and may be difficult to determine even at the code level.

Note that the same mechanism can manifest itself as either a timing or storage channel. The disk channel discussed by Wray [39] is a good example. In many cases, the value of some storage resource, such as the disk arm position register, is correlated with some temporal behavior, such as seek time. Thus a channel involving disk arm manipulation can manifest itself as either a timing or as a storage channel. Note also that the term "clock" can be used rather loosely. All that is really necessary to form a clock for the purposes of exercising a timing channel is for the receiver to be able to observe the order in which events occur and for the sender to be able to influence that order.

1.4.2 Senders and Receivers

A pedantic definition of covert channels requires that the sender and receiver be subjects (in the Bell and La Padula sense) under the control of the TCB. In a traditional multiuser system, this is a reasonable restriction. In distributed systems, especially widely distributed systems, the possibility of signaling mechanisms in which the recipient is not a subject exists. In these cases, the recipient is typically an intruder or wiretapper on the transmission medium used for the distribution. We prefer not to characterize such channels as covert channels. The reasons for this are discussed in [4], and have to do with the fact that the receiver and possibly resource attributes used in the exercising the channel may not be

mentioned, much less adequately described in the specification forms typically analyzed for covert channels. Nonetheless, we note that some of the techniques that can identify covert signaling mechanisms can identify wiretapping channels, as we call them, given a suitable system representation.

1.4.3 Level of Abstraction

Covert channel analysis can be performed on any level of system representation from abstract models to machine code (and hardware). By analyzing high level abstractions, security flaws can be identified at a stage of development where they are easier to fix. A flaw introduced early in the design process is, in effect, a requirement for the system to be insecure since it becomes part of the specification for refinement of the system design.

By performing covert channel analyses repeatedly as the system is refined, the introduction of security flaws can be minimized and appropriate countermeasures introduced to deal with flaws that must be present for reasons of functionality or performance.

At the hardware level, it is often the case that little flexibility is available. Hardware is often chosen before system design is complete, or the system must be hosted on existing hardware. The hardware base is likely to contain mechanisms that can be used to construct covert channels. Typical features include memory management units, shared memory or I/O busses, device controllers, etc. If a high capacity signaling mechanism is discovered in the hardware, there seems to be little choice other than to avoid its use altogether or to design the system so that the mechanism does not penetrate a security boundary. As shared memory multiprocessors become more common, this is likely to prove increasingly difficult.

1.4.4 When is a channel harmful?

Many covert channels are benign. Benign channels usually possess one of the following characteristics:

- The sender and receiver are the same subject
- The sender is allowed to communicate directly with the receiver under the system's security policy.
- There is no effective way to utilize the signaling mechanism.

For a covert channel to be harmful, the sender must be forbidden to communicate with the receiver under the system's security policy and there must be an effective procedure for exploiting a security flaw to form a channel for transmitting a useful quantity of information from sender to receiver in a timely fashion. Note that part of this definition is a function of the system in question and part of it is a function of the environment in which the system is used. For this reason, the decision to deploy a less than perfect system must ultimately lie with the accrediting authority, who must rely on information supplied by the evaluators, with possible inputs from the users and developers.

Section 2

System Characterizations

Covert channel analysis can be conducted on any level of system description from highly abstract specifications to implementation code. This section will discuss the suitability of various levels and forms of system representations for performing covert channel analysis. The primary system description for performing a covert channel analysis for a system to be evaluated at the B2 or B3 levels of the TCSEC is the descriptive top level specification (DTLS). This characterizes the TCB of the system in terms of its input, outputs, exceptions, and error messages. The preparation of a DTLS is covered in a related portion of this handbook [29] and will not be discussed in detail here. However, there are a number of issues that must be addressed if the DTLS is to be suitable for covert channel analysis purposes. In addition, it is possible, and usually desirable to perform a covert channel analysis on system characterizations other than the DTLS. The benefits and difficulties of such an approach will also be discussed here.

2.1 Levels of System Specification

Although the DTLS is, under the TCSEC, the primary description and specification for the system under evaluation, other forms of specification will be created during system development. These may range from higher level abstract models to lower level coding specifications and the code itself. Since the entire family of descriptions should be consistent, it is likely that a security flaw that is introduced in a high level abstraction will persist in a low level design and in the implementation. For this reason it is desirable to perform a preliminary covert channel analysis on the high level designs, refining the design only when its security characteristics and their consequences are understood. Current techniques for covert channel analysis require examination of the system as a whole, ignoring the information hiding and careful structuring that should be the result of good software engineering practices. This means that the effort required to analyze a system for covert channels is a function of the size of the system being analyzed and of its complexity. The ideal situation would be to analyze the machine code using the semantics of its hardware platform as a base. In practice, this is not feasible. Analysis at the source code level has been carried out experimentally [37], but with inconclusive results. The DTLS or its more formal counterpart, the FTLS, are the traditional vehicles for covert channel analysis. If they are sufficiently detailed and complete, as required by the TCSEC, it would seem to be

unlikely that their implementation would introduce substantial additional covert storage channels. Unfortunately, this is not true. The TCSEC requires the DTLS to capture the security related aspects of the system's behavior. This does not necessarily include a wide variety of functionality that may be used to build covert channels. Any system resource that is shared between users operating at different levels is a possible vehicle for building a covert channel. In recent years, channels have been reported that use process scheduling [12], disk seeks [39], and shared memory busses in multiprocessors [38]. As a result of this, a thorough covert channel analysis must consider implementation details that introduce or use resource sharing that is not apparent in the higher level specifications and that may not appear to be security relevant. This may involve an analysis of hardware as well as software.

2.2 System Description Issues

At the implementation level, the security analyst does not have much control over the way in which the system is described. The programming language to be used is usually prescribed. The hardware description is given (if at all) by the vendor. At higher levels, the analyst may have some influence. The following issues should be considered.

2.2.1 The system model

A DTLS can take on a number of forms. The state machine model is the most common. In a state machine model, the TCB is viewed as an abstract state machine that makes changes to or returns information about an internal collection of resources in response to requests made by users. The state machine appears to be the only specification form that lends itself easily to covert channel analysis. This is because it is relatively easy to demonstrate that a state machine specification is definitional and complete.

The suitability of other specification forms, such as trace specifications [26, 11] is open to question. Trace specifications allow a system to be characterized in terms of sequences of calls on system modules. The trace specification defines which sequences are legal and the results of sequences that end in value returning operations. A trace specification is said to be complete if it supports a model. This notion of completeness is weaker than the notion discussed in the following section as it allows multiple models with possibly differing behaviors to satisfy a single trace specification. For a trace specification to completely define the behavior of a system in a way that supports a covert channel analysis, it appears that the specification must be both complete and total. All the models of a total trace specification are equivalent in terms of input and output behavior. Given a total and complete trace specification for a system, it should be possible to produce a model of the system that is suitable for performing a covert channel analysis. There are no examples of this process in the literature, and some research is needed before the technique is proposed for developing a real system.

2.2.2 Completeness

For an adequate covert channel analysis it is essential that the analyzed system description be complete. This generally requires a definitional or cause and effect style of specification.

For example, it is not sufficient to say only that two output variables of a routine have equal values. This does not tell us why or how they become equal. From a covert channel standpoint, it may be crucial to know whether they are equal because they are both set to some constant value or because they are both set to the value of some input variable. Similarly, leaving functions pending or "To be determined" can inhibit a covert channel analysis since the analyst must assume that the function could obtain information from anywhere.

2.2.3 The Notation

System specifications can be presented in a variety of forms ranging from natural language to mathematical notation. In general, the more precise and unambiguous the notation, the better the covert channel analysis. There are substantial benefits to using mechanical tools to aid in covert channel analysis. The three analysis tools discussed in Appendix A each require the specification to be written in a particular formal specification language. If developers who understand such languages are available, they may provide advantages for B2 or B3 systems. The use of an approved formal specification language is required for an A1 system.

2.2.4 Formality

In the next section, we introduce dependency analysis. This provides a way of extracting information flows from a system description. Informal system descriptions typically result in informal information flow semantics and lead to imprecise dependency analyses. As the descriptive language becomes more formally defined, it is usually possible to define its information flow semantics more precisely, as well. As a bare minimum, a well defined pseudo code notation should be used for the DTLS. More formal notations such as 'Z' or VDM are preferable.

Machine processable specification languages such as those supported by the FDM, Gypsy, and EHDM systems are even more preferable since they have well defined information flow semantics as well as tools to support covert channel analysis based on these semantics. While the use of a formal specification language is mandated by the TCSEC for A1 systems, such a language can be beneficial for a lower assurance system and some consideration should be given to using these languages for B3 systems, even if machine checked security proofs will not be performed.

Section 3

Dependency Analysis

Dependency analysis characterizes the system under consideration in terms of its information flows. This section discusses the semantics of information flow and develops techniques for analyzing the flows in a variety of system representations ranging from English language specifications and pseudocode to formal specification languages and code. The importance of a complete, definitional specification cannot be over emphasized. Unless we can assume that the specification that we are analyzing explains all the changes in the system resources and resource attributes that can be observed when the system is in operation, the analysis will be incomplete. In the discussion that follows, it is assumed that the specification is complete and definitional. In addition, we will assume that, when we discuss the changes observed as a result of invoking a given operation, the changes seen are, in fact, a result of invoking that operation and not another one.

Unlike other analyses that we might perform on a specification (or on some other representation), dependency analysis is not concerned with what the system does but rather with how it does it. Since covert channels typically involve signaling coded information through variations in the values of storage items or through the variation of response time, we are only interested in how these variations can be effected or detected. We introduce the notion of a "dependency" as an abstraction to represent these variations. We say that a dependency from one system resource attribute, say A, to another, say B, under an operation, O, exists if it is possible to infer something about the value of A by first observing B, then invoking the operation O and then observing B again. Note that it is not necessary to determine the value of A in this manner. It is sufficient to be able to conform or refute some conjecture about the value of A. B may be a storage resource attribute capable of direct or indirect (via another dependency) observation. It may also be a timing resource attribute, such as the time required to execute O. In the latter case the initial observation may be another invocation of the operation and the conjecture may be that the value of A has changed.

The resource attributes involved in a given dependency are characterized as Targets, Sources, and Guards. Targets and Sources are reasonably intuitive, being the resource attributes modified by the operation in question and the resource attributes from which the modified value(s) were derived. Guards appear when the modification is conditional, that is, the modification may or may not occur depending on the values of one or more resource attributes. If an operation is conditional, the resources and resource attributes

used in deciding whether or not the target will be modified are called guard resources because they guard or control the modification decision. Note that it is possible to infer something about the values of the guard resources from observing a change or lack of change in a target when an operation involving a guarded modification of the target is invoked.

Formally, we define a dependency as a triple, $\{T; \{S\}; G\}$ where T is a system resource or resource attribute that is the Target of the dependency, $\{S\}$ is a set of system resources and resource attributes that are the Sources of the dependency. G is a boolean expression called the Guard of the dependency. Information flows from each $s \in \{S\}$ to T if and only if G evaluates to True. The objective of dependency analysis is to identify system resources and resource attributes whose values change when system operations are invoked and to determine both the sources of the modified values and the circumstances under which the modifications occur. This is done in two parts: the identification of system resources and resource attributes and the decomposition of the system specification to identify individual target modifications.

3.1 Resource Identification

The first step in performing a dependency analysis is to identify the resources and resource attributes under the control of the Trusted Computing Base (TCB) of the system being analyzed. This is relatively straightforward if the TCB specification is described in terms of a formal specification or pseudocode (assuming that the specification or pseudocode is relatively close to the implementation in level of detail), but is more difficult for both code and informal natural language specifications. Storage resources and attributes are by far the easiest to identify. Timing resources and attributes are more difficult.

The general approach is to analyze the system representation to identify all persistent resources, i.e. those that retain values from one system operation to the next. In specification styles that require an explicit declaration of storage entities, this is straightforward. For other styles, it may be necessary to infer a resource attribute from the active language of the specification. For example, the specification may call for testing to determine whether a file is "locked." From this, we can infer a "file locked" attribute associated with each file. Elsewhere in the specification, we would expect to find mention of an operation that "locks" the file and one that "unlocks" it. These operations also affect the "file locked" attribute.

Static structures, such as arrays, and dynamic structures, such as lists, have attributes that are modified or referenced as side effects of operations on them or their elements. These are discussed further in section 3.4.2

Timing resources are more difficult to identify, especially in high level descriptions. In the absence of explicit timing specifications, the usual case, the analyst must be aware of operations that seem to permit varying execution times and identify a timing resource to associate with each such operation. In performing the dependency analysis, the factors that lead to this variability must be identified.

It is useful to identify two "pseudo resources" to serve as place holders for information introduced into the system by the user and information returned to the user by the system. Any covert channel must start with a "user input" and end with a "user output."

Resource identification in code presents additional problems. These are discussed in

more detail in section 3.3 below.

3.2 Specification Decomposition

The objective of performing a dependency analysis is to capture the information flows through the system in such a way as to permit the identification of any security flaws that the system may have. In doing this, it is useful to examine the system, operation by operation and reduce its information flows to a uniform representation that is suitable for subsequent analysis. This is done in a series of steps.

Many system operations modify a number of different system resources. For example, opening a file for writing may set the file's "locked" attribute, set the use count of the file to 1, and enter the identification of the locker into the users list of the file. By splitting this larger operation into three smaller operations that are viewed as occurring in parallel, we add precision to the analysis since the resources that contribute to each of the individual modifications are considered separately.

The objective of the decomposition process is to separate the information flows in the system under analysis to an extent that ensures that security flaws can be properly identified and adequately scoped. This is often a trial and error process. if the decomposition is carried out in excessive detail, the same or similar analyses will be repeated over and over, showing that each of a group of similar flows is secure (or insecure). On the other hand, if the decomposition is not carried far enough, flows that are secure may appear to be insecure. A reasonable strategy for a large, complex system is to perform an initial analysis on a relatively coarse decomposition and to do further decomposition only on those components or operations that appear to contain security flaws, stopping when either a flaw is identified or the decomposition shows that a flaw is not actually present. This strategy will successfully identify the security flaws from which covert channels are constructed, but further decomposition may be required to construct scenarios for exercising the flaw as a part of a covert channel.

3.2.1 Guarded assignments

Guarded assignment statements are a useful form for representing the operations of a system. In the introduction to this section, we introduced the notion of a dependency as the fundamental abstraction of information flow. in converting a specification into dependencies, it is useful to introduce an intermediate form, the *guarded assignment.* The guarded assignment was introduced by Dijkstra [2, 3] as an abstraction for program design and specification. In decomposing a specification for dependency analysis, guarded assignments are useful because they succinctly represent the information flows that occur in connection with both conditional and unconditional operations.

Guarded assignments have the form: `If (G) Then T := S` where

`T` is the target of the operation. This is a resource or resource attribute that is affected by the operation. It may be an isolated entity, a timing resource, part of a structure such as an array or list, or an entire structure.

`S` is the source of the information that is directly transferred by the operation. It is an expression that is compatible with the target. It is unconditional, that is, it does not contain any "if expressions". The reasons for this are discussed in section 3.4.1 below.

`G` is the guard for the assignment. It is a boolean or logical expression involving resources, resource attributes, constants, and literal values. If it evaluates to true, the assignment takes place. If it is false, the assignment does not take place.

If the system specification is such that some target always changes, but the nature of the change depends on whether or not a given guard is `true`, two guarded assignments result, `If G Then T := S1`, and `If not G Then T := S2`. Unconditional changes can be represented by a guard of `true`.

In some cases, it is useful to nest guarded assignments. This is often the case when a security related test is made to determine whether the functional operation should be allowed. For example, if we only allow subjects to open files for writing at their own security level, we might represent part of the operation as

```
If  Level(U)  =  Level(F)  Then
     If  not  Locked(F)  Then
          Locker(F)  :  =  U;
```

where Level is a function that returns the classification or clearance of objects (files `F` or subjects (users `U`), Locked is a file attribute that indicates whether or not the file is open for writing, and Locker is a file attribute that contains the identity of the subject that locked the file.

This formulation allows us to assume the outer guard in reasoning about the security of the operation, something that may be necessary to demonstrate the absence of a security flaw. Note that operationally, the nested `If A then If B then ...` has a distinctly different meaning in terms of information flow from the conjunction `A & B` in that the former implies an order of examination with `A` being evaluated first and `B` being evaluated only if `A` evaluates to `true`. On the other hand, `A & B` imposes no such order and results in the evaluation of both `A` and `B`. The effect is that there is an information flow from B in the nested structure only if `A` is `true` while there is always a flow assumed from `B` in the conjunction.

In decomposing an informal specification, it is usually clear which form is intended. Designers should be aware that nesting tests with appropriate error responses at each level of the nesting may avoid security flaws that would be present if a conjunctive form were used.

3.2.2 Sources and Targets

Once a system has been decomposed into guarded assignments, it is necessary to identify the resources and resource attributes from which information flows as well as those into which it flows. The sources include all resources that are mentioned in the `S` portion of the guarded assignment. They also include all those mentioned in the guard `G`. This is because we can infer something about the resources of `G` if we assume that we know the value of `T`

and can observe T before and after the operation. For example, if we observe a change in T when the operation in question is invoked, we know that G was `true`.

In addition, T may contribute to the list of sources. If T is a component of a structure, the resources referenced in its indexing expression will be sources. To see why this is so, consider the case in which the assignment `T(I) := S` is made. If we know that no element of T had a value equal to that of S before the assignment, we can determine I by examining T for changes.

At first glance, target identification appears to be more straight forward. Were it not for structures, this would be the case. If the target T of the guarded assignment is a simple, isolated resource or resource attribute, it alone is the target of the information transfer. As we have seen above, modifying an element of an array allows determination of the elements index under some circumstances. To account for this, we introduce the concept of "name resources." The information transfer into the array as a whole that results from an assignment into one of its elements is said to be a modification of its "static name resource." The security implications of this were first discussed by Gasser [8] in 1979 and later considered in detail by Singer [36]. A similar situation exists for dynamic structures such as lists, sets, mappings, etc. that are often used for specification purposes. In addition to a static name resource, these structures have dynamic name resources. For example, an operation that appends a new entry to the end of a queue changes both the contents of the queue and its size. The size is considered to be a dynamic name resource. It may be observed directly or indirectly. Indirect observations include full or empty checks for such structures as well as membership tests for sets and the like. Thus a single guarded assignment may represent multiple targets.

A guarded assignment statement, `If G Then T := S` gives rise to one or more dependency triples $\{T;\{S\};G\}$ depending on the number of resources or resource attributes represented by the target T of the guarded assignment. The source set $\{S\}$ for each dependency consists of all the resources and resource attributes explicitly and implicitly referenced in the source and guard expressions of the guarded assignment statement from which the dependency is derived. If the target of the guarded assignment involves a selection expression, this may also contribute sources to the dependency.

Thus, the source set $\{S\}$ for a dependency $\{T;\{S\};G\}$ is $\{S\} = \{S_s\} \cup \{S_g\} \cup \{S_t\}$ where

$\{S_s\}$ is the set of sources obtained from the source expression S of the guarded assignment statement,

$\{S_g\}$ is the set of sources obtained from the guard expression G of the guarded assignment statement,

$\{S_t\}$ is the set of sources (if any) obtained from the target expression T of the guarded assignment statement,

The objective of specification decomposition is to reduce the specification, no matter what its original form to a list of dependencies, each consisting of a single target, a list of sources derived from the guard and target as well as the source of a guarded assignment statement, and a guard expression. Once the decomposition has been performed and the system security policy considered (see section 4) the search for covert channels can begin.

3.3 Code analysis issues

Most specifications are effectively functional in nature. While they may explicitly describe effects on the global state of the system, they do not usually introduce temporary local resources (variables) and do not usually have hidden side effects that alter the global state in unexpected ways. Code also often makes use of procedures that can affect multiple resources independently. For these reasons, it is necessary to analyze code in a different manner from specifications.

3.3.1 Local variables

Local variables can be a source of confusion in performing a code analysis. For example, in the following code fragment `tmp` is local.

```
tmp  :=  A  +  B;
D  :=  tmp;
tmp  :=  F  +  G;
H  :=  tmp;
```

Some analytical techniques would (wrongly) indicate a flow from `A` to `H` while it is obvious that this is not the case. The best approach is to transform the code in a way that eliminates local variables, in effect converting imperative code into its functional equivalent. In the case of the example, this transformation would yield:

```
D  :=  A  +  B;
H  :=  F  +  G;
```

In the case of loops, this may require a transformation to a recursive form and/or a transitive closure of the information flows introduced in the loop. Techniques for doing this are discussed in detail in [17].

3.3.2 Procedures and Functions

In most programming languages, procedures and functions serve two purposes. The first is to provide an encapsulating abstraction for an operation or a related group of operations. The second is to extend the operational vocabulary of the language by providing an efficient way to apply or reuse the encapsulated abstraction in a variety of contexts. A procedure or function is usually defined in terms of its use of and its effects on a list of dummy variables called its formal parameters. These are placeholders for the variables to which it will be applied when it is called in a program that uses it. When the procedure or function is called, real variables or expressions from the calling environment are substituted for the dummy variables used in the definition of the function or procedure. These are the actual parameters of the call site under analysis.

Procedures, especially those that modify more than one of their formal parameters, pose an analysis problem. In general, the best approach is to treat each output formal parameter as though it were computed independently and the procedure call as though it were a parallel invocation of a set of assignments, one for each formal output parameter. The

key to performing this analysis manually is to adopt a notation that distinguishes between the initial (time of call) and final (time of return) values of each output parameter. (In languages such as Ada that have output only parameters there may not be an input value.) The net effect of the analysis and transformation is to create a set of functional definitions, one for each formal output parameter, for the procedure that can be substituted for the procedure call. Functions without side effects can be treated in a similar fashion, reducing their imperative form to a functional one for their result or return value. A simple example will illustrate the process:

```
Procedure  OpenW (      F  :  File,
                  Var   L  :  Lock,
                  Var   W  :  Writer,
                        R  :  Reader,
                        U  :  Users)  =
                  If R = Null  and  not L
                    then
                      L  :=  true;
                      W  :=  U;
                    end;
```

Reduced to functional definitions for the formal output parameters, L and W, this becomes:

```
L(L', R)  :=  If R = Null  and  not L' then  true  else  L';
W(L', W', R, U)  :=  If R = Null  and  not L' then  U  else  W';
```

where L' and W' denote the original (or calling time) values of L and W respectively.

In tracing dependencies through a procedure or function call, the actual calling form is replaced by the functional form(s). This is expanded to replace the functional call with its defining expression. A substitution is then made, using the actual parameters of the call to replace the formals of the called routine. The result, at the calling site is as though the called routine had been replaced by inline code having the same effect.

Continuing the example, if a call of this procedure were made, say

```
OpenW  (TheFile,
        Locked(TheFile),
        Writers(TheFile),
        Readers(TheFile),
        TheUser);
```

The effect would be the same as if the code

```
Writers(TheFile) :=  If      Readers(TheFile) = Null
                        and  not Locked(TheFile)
                     then    TheUser
                     else    Writers(TheFile);
Locked(TheFile)  :=  If      Readers(TheFile)  =  Null
```

```
 and  not Locked(TheFile)
then  true
else  Locked(TheFile);
```

had been inserted in place of the call.

Note that the order of the operations was reversed to avoid problems with the simultaneous redefinition and use of Locked(TheFile). In general, it may be necessary to add a distinguishing notation such as the prime used above each time a variable is redefined during the transformations. In the end, substitution of actual parameters for formal parameters will remove the intermediate renamed instances leaving only initial and final versions.

3.3.3 Side Effects

Most programming languages allow global variables, that is, variables that may be directly referenced by a number of different functions or procedures. Many languages also allow functions to modify global variables and parameters in addition to returning a result. These actions are called side effects. When a procedure or function having side effects is analyzed for dependencies, additional assignments are added to the effect set of the procedure or function. These are introduced at each calling site. Interactions between the primary effects and the side effects are possible (though they represent poor software engineering practice) and care must be taken to ensure that the proper order of dependency is captured through either ordering of the substituted code or through variable instance naming.

3.4 Expression Decomposition

In general, the source `S` of a guarded assignment is the same whether a specification or code is being analyzed. From a practical standpoint, the complexity of the expression and the number of unknowns that it contains affects the strength of the dependencies that it represents. In the case of dependencies that are involved in a covert channel, this will affect the signal to noise ratio of the channel. This will be discussed further in section 6.

All variables, constants, and literals that appear in an expression are sources for the expression. Thus if we have a simple expression such as `a + b * c` the sources are `a`, `b`, and `c`. There are several cases that deserve closer attention.

3.4.1 If Expressions

In general, it is preferable to move all conditionality to the guards of guarded assignment statements. If the source expression being analyzed contains a conditional, the conditional can be added to the guard of the guarded assignment and the "true" branch of the conditional substituted for the conditional. In addition, a new guarded assignment is created with the negation of the condition added to its guard and the "false" branch of the conditional substituted for the conditional. For example:

```
If  G  Then  X  :=  If  Z>0  Then  Y  Else  W  fI;
```

would become

```
If  G  &  Z  >  0  Then  X  :=  Y;
If  G  &  Z  <=  0  Then  X  :=  W;
```

or, if a nested form is indicated

```
If  G  Then  If  Z  >  0  Then  X  :=  Y;
If  G  Then  If  Z  <=  0  Then  X  :=  W;
```

3.4.2 Arrays and other structures

We have noted earlier that the use of indexable structures introduces additional resource attributes. These attributes may be referenced directly or indirectly. Singer [36] calls these resource attributes name resources. Indexable structures of constant size have a single name resource that reflects the fact that operations on the elements of the array can be detected by observations of properties of the array as a whole. In particular, a direct reference to a component of such a structure introduces not only the element as a source but also the static name resource of the structure. Dynamic structures have additional name resources. Size is a resource attribute that can be changed by adding or removing an element to a structure. Structures such as mappings are often used as abstractions in specifications. They introduce additional dynamic name resources such as the domain and range of the mapping. These, in turn, have name resources of their own. Operations such as push and pop on stacks also reference the size resource of the stack. Enqueue and dequeue operations have similar effects. Operations on dynamic structures should be examined with care to determine all the dynamic name resources associated with them. Operations that fail under some circumstances may return information about one or more of the dynamic name resources associated with the structure. Note that operations on dynamic structures are often defined so that exceptions are raised if the operation results in an error such as underflow or overflow. Raising the exception may allow a stronger inference about the structure than not raising it, but some inference is possible in either case.

3.5 Approximations and Conservative Replacement

A complete dependency analysis, especially of code or of a highly detailed specification requires a large amount of work. If good software engineering practices have been followed, much of this work will involve the expansion and substitution of definitions for uses, especially at function or procedure call sites.

Is it possible to avoid some of the effort altogether or at least to postpone it until it is clearly needed? Sometimes this is possible. In the absence of side effects or references to global variables, the worst that a procedure or function can do is introduce dependencies between all of its inputs and each of it outputs. Side effects and references to globals complicate the situation somewhat, but knowing which globals are referenced and/or modified is equivalent to adding them to the parameter list, and a similar worst case applies.

If the system can be shown not to have security flaws under this worst case assumption, further refinement of the dependencies is not needed. On the other hand, if a suspected flaw appears to be due to a worst case assumption, a more detailed analysis may eliminate the flaw or may confirm it and pinpoint the exact mechanism.

Complex expressions involving arbitrary indexing may be treated in the same way. If the system would be secure in the face of dependencies on all structure elements, it is not necessary to determine which elements are actually involved. The practice of substituting a more comprehensive set of dependencies for a specific one is called conservative replacement. This is discussed in more detail in [19], but the general principles can be seen in the following examples.

Suppose that we want to suppress the detail in the procedure OpenW discussed in section 3.3.2 above.

```
Procedure  OpenW  (       F  :  File,
                             Var  L  :  Lock,
                             Var  W  :  Writer,
                                    R  :  Reader,
                                    U  :  Users)  =
                    UNDEFINED;
```

In this formulation, the body of the procedure is not known and we have no choice but to assume the worst case which is an information flow from each of the procedures inputs into each of its outputs. We assume that we know that the implementation of the procedure will not reference any system resources not explicitly or implicitly mentioned in its parameter list. Reduced to functional definitions for the formal output parameters, L and W, this becomes:

```
L(F,  L',  W',  R,  U)  :=  UNDEFINED;
W(F,  L',  W',  R,  U)  :=  UNDEFINED;
```

Substituting the actual parameters from the call shown in section 3.3.2 and reduced to dependency triples, $\{T;\{S\};G\}$ the information flows for the undefined version of OpenW becomes:

```
{Locked(TheFile);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Writers'(TheFile);  Name(Writers);  Size(Writers'(TheFile));
   Readers(TheFile);  Name(Readers);  Size(Readers(TheFile));
   TheUser};
  true}

{Name(Locked);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Writers'(TheFile);  Name(Writers);  Size(Writers'(TheFile));
   Readers(TheFile);  Name(Readers);  Size(Readers(TheFile));
   TheUser};
  true}

{Writers(TheFile);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Writers'(TheFile);  Name(Writers);  Size(Writers'(TheFile));
```

```
   Readers(TheFile);  Name(Readers);  Size(Readers(TheFile));
   TheUser};
  true}

{Name(Writers);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Writers'(TheFile);  Name(Writers);  Size(Writers'(TheFile));
   Readers(TheFile);  Name(Readers);  Size(Readers(TheFile);
   TheUser};
  true}

{Size(Writers(TheFile));
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Writers'(TheFile);  Name(Writers);  Size(Writers'(TheFile));
   Readers(TheFile);  Name(Readers);  Size(Readers(TheFile));
   TheUser};
  true}
```

The name resources appear because the `Locked`, `Writers`, and `Readers` attributes are assumed to be indexed by `TheFile`. The size attributes for `Readers(TheFile)` and `Writers(TheFile)` appear because these structures are assumed to be lists of users. In contrast, the dependencies for the version of `OpenW` given in section 3.3.2 would be:

```
{Locked(TheFile);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Size(Readers(TheFile)};
  true}

{Name(Locked);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Size(Readers(TheFile)};
  true}

{Writers(TheFile);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Size(Readers(TheFile);  TheUser};
  true}

{Name(Writers);
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Size(Readers(TheFile);  TheUser};
  true}

{Size(Writers(TheFile));
  {TheFile;  Locked'(TheFile);  Name(Locked);
   Size(Readers(TheFile);  TheUser};
```

```
true}
```

The difference in the number of attributes that need to be considered in the two cases is substantial, but if all the attributes involve similar security analyses, there may be little difference in the analysis effort required. In general, procedures or functions whose arguments do not differ in security level can be handled more efficiently using conservative replacement of this kind and making a general argument concerning the security of all the potential flows whether they occur or not. If this general argument cannot be made or level dependent flows are known to be present in the routine, expansion of the routine's definition is clearly indicated. For a second example of conservative replacement, suppose that the expression

```
(String1[i..j]  @  String2[k..l])[m]
```

appears in the specification being analyzed. In this case, the expression produces the `m`'th element of the sequence produced by concatenating the `i`'th through `j`'th elements of `String1` with the `k`'th through `l`'th elements of `String2`. Possible sources of information flow from this expression include `m`, `i`, `j`, `k`, `l`, `String1[i] ... String1[j]`, `String2[k] ... String2[l]`, `Name(String1)`, `Size(String1)`, `Name(String2)`, and `Size(String2)`. If a dependency on all these sources doesn't lead to the identification of an insecurity, it is easier to use the conservative formulation than it is to determine the exact circumstances under which the flows occur. The less conservative approach would be to transform the expression into a form conditioned on the index values. For this purpose, we will assume that the index values are within range as treatment of the out of range cases introduces yet another level of complexity. Assuming zero based indexing, the original expression can be rewritten as:

```
if  m  <  j-i+1  then  String1[m+i]
                 else  String2[m-j+i-1+k]  fi
```

This form would result in two guarded assignments, one for each branch of the if-expression. The information sources for each branch are a subset of those that appear for the original form and the relationship between `m`, `i`, and `j` can possibly be used in proving that the flows are secure. This formulation would only be useful if the security levels for elements of `String1` differ from those for `String2`, an unlikely situation. In general, the conservative replacement approach will prove adequate and will reduce the effort required to decompose a specification.

Section 4

Security Policy Issues

A covert channel represents a violation of the real security policy of the system under consideration. The real policy is usually an extension of the policy embodied in the Formal Model of the Security Policy (FMSP) of the system. Security policies are discussed in detail in a related portion of this handbook [30]. In performing a covert channel analysis, it is necessary to extend the security policy of the system so that security characteristics can be assigned to all resources and resource attributes of the system description being analyzed.

In general, we assume that the system has been characterized as a state machine and that the security policy model takes the form of either a Bell and La Padula [1] style access control model or one of its information flow analogs [18]. In either case, the model will be couched in terms of passive, information containing, objects, and active, information free, subjects (and possibly hybrids of various types) and will indicate the nature of the accesses or access permissions that a subject may obtain to an object as a function of the security attributes (clearances for subjects, classifications for objects) possessed by the subject and objects in question.

In applying such a model, a limited view of objects is usually taken. This restricts objects to entities such as files and devices that are intended to store substantial quantities of information. For the purpose of performing a covert channel analysis, any shared resource or shared resource attribute is an object. In the previous section, the dependency was introduced as an abstraction to represent arbitrary information flows. Applying a security policy to a dependency, $\{T; \{S\}; G\}$, we note that, since the target, T, of the dependency is modified, the subject invoking the operation that gives rise to the dependency must have modify access to the target and must have read access to each source in $\{S\}$ since the information that modifies the target comes from these sources. Because the target is modified only when the guard, G, is true, it must be the case that the invoking subject would be granted the necessary accesses only under circumstances that would result in the guard being satisfied. If the information flow represented by the dependency is to be considered "secure", this will be the case.

The TCSEC requires that the "subjects" and "objects" to which the FMSP applies be labeled with machine processable representations of the security attributes on which access control decisions are made. The process of associating security characteristics with system resources and resource attributes is an extension of this labeling requirement. It is not necessary to create physical labels or storage structures in the implementation to

contain this information, but it is necessary to describe the process for determining the security characteristics for each resource and resource attribute. The mechanisms may involve appealing to the contents of real system labels or it may involve the assignment of constant security attributes to some resources.

4.1 Linking policy to resources

Dependency analysis identifies the potentially shared system resources and attributes. Each resource and attribute must be given a classification if we are to reason about the security of dependencies. The set of values available is defined by the FMSP. DoD style policies typically define a lattice of values with each value consisting of a hierarchical classification and a (possibly empty) set of categories. Several distinguished values are of interest.

model low is the minimum classification value defined by the FMSP. It is dominated by all of the classification values defined by the FMSP.

model high is the maximum classification value defined by the FMSP. It dominates all of the classification values defined by the FMSP.

system low is the lowest classification value at which a given system is permitted to operate. It dominates model low and is dominated by all classifications that may be assigned to the subjects and objects of the system.

system high is the highest classification value at which a given system is permitted to operate. It dominates system low and all classifications that may be assigned to the subjects and objects of the system and is dominated by model high.

A substantial amount of skill is required in assigning security attributes. If inappropriate decisions are made, the system may appear to be far less secure than it actually is. On the other hand, it is not possible to mask insecurities by inappropriate labeling unless the labeling mechanism violates the tranquility principle by allowing the classification of a resource to change between (or during) operations. There are several easy cases:

1. The resource is an object in the terms of the FMSP. In this case, its security attributes are exactly the same as those it has in the FMSP.

2. The resource or attribute is logically associated with an object. In most cases, the security attributes should be the same as those of the object. This implies that file names have the same classification as files, etc. Exceptions are possible; for example, a system might have a predefined, fixed set of file or device names that can be read by all, in which case "system low" is an appropriate level.

3. The resource is closely associated with a subject. In this case, the security attributes of the subject are probably appropriate, although exceptions may exist.

4. The resource is never modified by an untrusted subject. "System Low" is appropriate.

5. The resource is never observed by an untrusted subject. "System High" is appropriate.

What remain are the hard cases. Resources that are shared but do not fall into one of the classes above are likely to be involved in a security flaw. It is probably the case that no security level can be found that will make all dependencies involving the resource secure. Arbitrary assignments of security attributes can cause the flaw to appear to be associated with various operations. If auditing is being considered as a countermeasure, associating the flaw with a less common operation will reduce the quantity of audit data collected.

Associating security attributes with timing resources seems problematic at first. Timing attributes are observed by the system being analyzed. If we associate the level of the subject invoking the operation with the timing attribute, the appropriate relationship is established since a secure timing dependency will only involve sources that are dominated by the invoking subject. Insecure timings will involve sources that the subject does not dominate.

Once the policy has been extended to cover all system resources and resource attributes and a dependency analysis has been performed, a search for security flaws can be made. This is covered in the next section.

Section 5

Looking for Channels

In the previous sections, we have developed a theory of information flow through dependencies and have discussed the issues surrounding the application of a security policy to a system representation. Covert Channel Analysis is a search for dependencies that can be exploited in violation of the system security policy. This section will develop a sequence of analytical approaches beginning with the basic Shared Resource Matrix and its extensions and proceeding to Covert Flow Trees and Information Flow Formulae. Non-interference formulations will also be discussed.

5.1 The Shared Resource Matrix

First introduced by Richard Kemmerer in the early 1980s, the Shared Resource Matrix [13] remains one of the best known analytical tools available. While the SRM has its limitations, it is still the most concise method for characterizing the information flows in a system. We spend a substantial amount of time developing the extended SRM as a tool for guiding a search for covert channels. Our development starts with the assumption that we have analyzed a system representation for dependencies and have the results available to us. To provide a concrete example, we will consider a fragment of code which, while meaningless by itself, serves well to illustrate the principles involved.

```
Global var a, b, c, d;

Procedure OP1 (Var u : uvar) =
begin
  a := if b then c else d;
  u := c;
  end;
```

We assume that the system state is completely characterized by the global variables `a`, `c`, `b`, and `d`. We also assume that the fragment is a complete characterization of the procedure `OP1`, that is, there are no hidden operations to change the values of the variables `u`, `b`, `c`, and `d`. The analysis and decomposition of OP1 results in the following guarded assignments:

```
If  b  then  a  :=  c;
If  not  b  then  a  :=  d;
If  true  then  u  :=  c;
```

which yield the following dependencies:

```
{a;  {b;  c};  b}
{a;  {b;  d};  not  b}
{u;  {c};  true}
```

5.1.1 The basic SRM formulation

As originally formulated by Kemmerer, the SRM has rows that correspond to the attributes of shared resources (variables `a`, `b`, `c`, `d`, and `u` in our simplified example) and columns that indicate the system operations (`OP1` in our case) on each shared resource. An **M** in an entry in the SRM means that a resource attribute is modified by the operation in question while an **R** indicates that the resource attribute is referenced or observed by the operation. The SRM for the fragment of code shown above would look like the following:

Resource Attribute	Operation Op1
a	M
b	R
c	R
d	R
u	M

Within a column, it is assumed that information about all referenced attributes is used to determine the new values of all modified attributes. This is fine as far as it goes. The primary problem is that the SRM is an overly pessimistic view of the system behavior. For example, the SRM above indicates that variable `a` is modified using information from variables `b`, `c`, and `d` and that variable `u` is also modified using the same information while inspection of the code shows more restrictive flows. If the system is secure, the conservative approach is harmless. If the system has flaws, the conservative approach indicates potential covert channels where none actually exist.

5.1.2 Adding detail to the SRM

In order to solve the potential problem of an excessively conservative system representation, several extensions can be made to the basic SRM. The first allows us to explicitly separate the information flows that occur between the user and the system from those that occur between the components of the system state. The second allows us to separate the flows into individual targets, while the third allows us to extract the conditions under which flows actually take place. When we reach the end of this development, we will have arrived at a basis for understanding the information flow formulas used by mechanical flow tools such as the GIFT and Ina Flow.

Characterizing inputs and outputs

The first transformation may seem to be trivial, but it is of considerable importance in providing a clean conceptual basis for understanding information flows through a system. Adding an explicit indication of flows to and from the user to the SRM representation forces us to realize that the system description must support analysis at a level of detail that makes it possible for us to identify such flows. In the previous formulation, we have indicated that the variable `u` which appears as a parameter to `OP1` is a resource attribute. Clearly, in a large system, we would not want to clutter the attribute list with all the parameters that appear in all the calls from users. In addition, the system can return information to users via signals, exceptions, shared memory objects, etc. From the viewpoint of the analyst, the exact route by which information is returned is irrelevant *until* it is determined that the return is involved in a potential covert channel. Analysis of such a channel takes us into the details of the system and beyond the SRM abstraction. Thus, we can lump all information returns into a single row of the SRM. In a similar fashion, we can lump together all inputs from the user into the system. Note that not all operations return information to the user. In trusted systems especially, many operations are designed not to provide output, or to provide output that is uniformly independent of the system state and the success or failure of the operation. On the other hand, all operations contain information of some sort from the user since the mere fact that the operation was invoked may alter an attribute of the system state.

The result of modifying the basic SRM to explicitly record user inputs and outputs is as follows:

Resource Attribute	**Operation Op1**
a	M
b	R
c	R
d	R
User In	R
User Out	M

The User In row will always contain an R indication. The User Out row will contain an M only if the user is able to perceive some information about an attribute of the state from the response to the call. If the SRM is being used to analyze a system for timing channels, the analyst should remember that response time is one way in which the user can perceive state information.

Distinguishing targets

The basic SRM gives the impression that information flows to the user and to state component `a` from each of the state components `b`, `c`, and `d` as well as from the user. Inspection of the code shows that this is not the case. The obvious solution is to split the matrix into additional columns so that each column contains only a single M entry. If we do this, the resulting SRM is:

Resource Attribute	Operation	
	Op1	Op1
a		M
b		R
c	R	R
d		R
User In	R	R
User Out	M	

Distinguishing cases

This is better, but it still leaves the impression that information flows to `a` from `b`, `c`, and `d` on every call to `OP1`. The real situation is that it flows from `c` or from `d` depending on the value of `b` but it never flows from both on the same call. Again, we can split the columns of the SRM to reflect the two cases. If we do this, we get an extended SRM of the form:

Resource Attribute	Operation		
	Op1		
	G1	G2	G3
a		M	M
b		R	R
c	R	R	
d			R
User In	R	R	R
User Out	M		

where G1 = true
G2 = b
G3 = $\neg$ b

5.1.3 Identifying Security Flaws

The extended SRM provides an accurate description of the system being analyzed from an information flow viewpoint, provided that the system description analyzed is complete, that the system state has been accurately characterized in terms of the resource attributes used and that the semantics applied to the system description in determining the dependencies reflected in the SRM are adequate. If these conditions are met, the SRM can be used to explore the system description for security flaws. In the next section, we will develop a mathematical analysis technique to support this exploration, but first we will try to motivate the process.

Consider that each column of the SRM represents the modification of some system resource attribute or the return of some information about system resource attributes to a user process as the result of a request made by that user process. The sources from which the information flows are indicated by "R" entries in the SRM, while the targets of the information flows are indicated by the "M"entries. If the system is secure, all these flows will represent information flows that conform to the system's security policy, that is, the

security level of the information source will dominate the security level of the flow's target. There are two cases to be considered:

1. The information source is a resource attribute at a security level that is not dominated by[1] that of the target, which is User Out. In this case, the system is seriously flawed, since the transfer is independent of the actions of any other user or subject of the system. Such cases should be discovered by other means, long before covert channel analysis is undertaken.

2. The other case is more subtle. The source of the flow may be a system resource attribute or a user input. The target is a system resource attribute and the security level of the source is not dominated by that of the target. In either case, we have a security flaw in which the actions of a user can cause information to flow within the system in a manner that is contrary to the system security policy.

In exploring the system specification, looking for security flaws of either type, the analyst should consider the circumstances under which information flows actually occur. The guards of the extended SRM capture these conditions explicitly. Guards that are couched in terms of appropriate tests of the relative security levels of the sources and target reflect efforts to enforce the system policy. If the guard can be seen to be consistent with the policy and to cover all the sources, the flow should be legitimate. If the guard fails to consider one or more sources and it can be seen that the sources can or do contain information at a level that shouldn't flow to the target, then a flaw has been identified. Often the flaw is subtle since neither the source nor the target is a resource explicitly intended as an information containing object and the appropriate guard formulation is not obvious. A case to look for is one in which a resource attribute is both modified and observed by users and in which the guard fails to consider either the user or resource level for at least one of the accesses. Such cases are likely to be found in several situations. The most common ones are:

1. Shared access to a system object, such as a file, by users at differing levels when the success or failure of a given operation depends on whether (or how) the object is being shared.

2. Operations that allocate system resources from a common pool so that the results of allocation operation depend on the history of past operations.

The techniques of the following section force us to assign security levels to all resource attributes, but some of the assignments are arbitrary and the resulting analysis is unconvincing without substantial justification. Whether performed by "the seat of the pants" or using more formal techniques, there is no substitute for experience, skill, and a devious mind in performing this analysis.

[1] The convoluted language is necessary to cover the case where the levels of the source and target are not comparable. This can occur, for example in the case of flows between compartments or categories of the same security level.

5.1.4 Finding Covert Channels

A security flaw does not a covert channel make. For the flaw to be exploited, it is necessary to find a scenario that allows the flaw to be exploited to transfer information between two user processes that would not be allowed to communicate directly under the system's security policy. To do this, the analyst must find a sequence of system operations that allows the sending process to modify a system resource attribute in violation of the system policy and allows the receiving user to detect the modification.

For this purpose, the transitive closure of the SRM is often useful. The basic and extended SRMs allow us to determine the direct sources of information that contribute to the modification of a system resource attribute or to a user output. The transitive closure of the SRM allows us to determine the indirect sources of information, that is information that requires a sequence of system operations to propagate to a given resource attribute or output.

The fragment of code that we have considered thus far only contains a single operation and is structured in such a way that performing the transitive closure adds no new information to the system. Consider, however, the following Basic SRM which comes from the example of Kemmerer [13] but has been modified to make the flows to and from the user explicit.

Resource Attribute		System Operation							
		Write File	**Read File**	**Lock File**	**Unlock File**	**Open File**	**Close File**	**File Loc'd**	**File Opn'd**
Process	**ID**								
	Access Rights			R		R		R	R
Files	**ID**								
	Security Classes			R		R		R	R
	Locked By	R		M	R				
	Locked	R		RM	RM	R		R	
	In–Use Set		R	R		RM	RM		R
	Value	M	R						
User In		R	R	R	R	R	R	R	R
User Out			M					M	M

The transitive closure is formed by noting that the read of a resource attribute in one operation when that resource attribute has been modified in a second operation is an indirect read of all the resource attributes that contributed to the modification. We can form the transitive closure, following Kemmerer's method, by looking at each entry that contains an "R" and checking to see if there is an "M" in the same row. If there is, we check the entries in the "M" column to see if it references (directly or indirectly) resource attributes not already referenced in the "R" column. For each such attribute, we add an "r" to the corresponding resource attributes in the "R" column. This process is repeated until it converges and no more indirect references can be added. The results of performing a transitive closure on the above SRM are as follows:

Resource Attribute		System Operation							
		Write File	**Read File**	**Lock File**	**Unlock File**	**Open File**	**Close File**	**File Loc'd**	**File Opn'd**
Process	**ID**								
	Access Rights	r	r	R	r	R	r	R	R
Files	**ID**								
	Security Classes	r	r	R	r	R	r	R	R
	Locked By	R	r	rM	R	r	r	r	r
	Locked	R	r	RM	RM	R	r	R	r
	In–Use Set	r	R	R	r	RM	RM	r	R
	Value	M	R						
User In		R	R	R	R	R	R	R	R
User Out			M					M	M

Note that most of the previously empty entries in the SRM contain indirect or "r" references. This indicates that, for this system, information flows rather freely among the operations. For larger system descriptions and fully expanded SRMs, this is less likely to be the case, and clusters of operations and resource attributes will appear. These clusters are appropriate places to start looking for scenarios that allow users to exploit a security flaw and create a covert channel.

At times, it may be useful to use the process of computing the transitive closure in the search for covert channel scenarios. This may be done by working from the flawed operation backwards to the initiating operation and forwards to the receiving operation.

In Kemmerer's example, the security policy forbids users with only read access to a file from communicating with those who have read/write access. A security flaw exists in the **Lock File** operation because a user with read/write access can modify the **Locked** attribute of the file based on whether the file's **In–Use Set** is empty or not. Since **Lock File** does not return any information to the user, we must look for other ways to observe the **Locked** attribute. We note that **File Loc'd** can return information about the **Locked** attribute. Performing a transitive closure of the **File Loc'd** operation with respect to the **Locked** attribute indicates a transitive read of the **In–Use Set** attribute via the **Lock File** operation, and we have identified the receiving portion of the scenario. We then look for operations that can modify the **In–Use Set**. Both **Open File** and **Close File** can do this and can be successfully invoked by users with only read access to the file. This provides the sending portion of the scenario.

When a potential covert channel is identified, it is necessary to see whether it can actually be exploited to transfer information in violation of the system's security policy. There are four possible cases to consider.

1. A legal or overt channel operates in parallel with the potential covert channel. In this case, the scenario does not include a real security flaw and something is wrong with the analysis.

2. No useful information can be gained from the channel. This will be the case if the

only information that can be signaled over the channel is information that the receiver already possesses.

3. The sending and receiving processes are the same. Processes are allowed to "mumble."

4. A covert channel exists and should be analyzed using the techniques of the next chapter.

5.2 Information Flow Formulas

Information flow formulas (also known as Security Verification Conditions (SVCs) are usually considered only in the context of a mechanical flow tool. SVCs form a basis for reasoning about the security of information flows and have a role in manual as well as automated analyses.

Several covert channel tools have been built based on the generation of information flow formulas. These tools are discussed in more detail in appendix A below. These tools require a restricted form of a state machine specification for the system being analyzed. A security level is assigned (sometimes arbitrarily) to each state component and to the invokers of the system operations. For each transfer of information within the system, a putative theorem is generated that, if true, guarantees that the information transfer proceeds according to the system's security policy.

This approach is overly conservative for several reasons:

1. It considers operations in isolation. It is often the case that there is no way to exploit the apparent insecurity of a given routine.

2. The required labeling of every state component with a classification results in inappropriate labels for some components. In many cases, the number of failed theorems is a function of the labeling strategy used, and substantial effort is required to choose a suitable labeling.

If it is possible to find a labeling that results in no failed theorems, then the model is free of covert storage channels. Failed theorems do not necessarily indicate exploitable channels, but substantial effort is required to ensure that this is the case. Most real systems exhibit a fair number of failed theorems when this approach is applied.

5.2.1 Deriving SVCs from a detailed SRM

The fragment of code used in connection with the above discussion of the SRM approach can be used to demonstrate the nature of the formulas that would be produced by the mechanical information flow tool. If we present the fragment as an equivalent set of guarded assignment statements, it becomes

```
if   b   then a := c;
if not b then a := d ;
if  true then u := c;
```

In addition to the guarded assignments that characterize the system in terms of conditional information flows, it is necessary to know the security level associated with each of the system entities, a, b, c, d, and u. Without a loss of generality, we will define a unique level function for each entity, leaving the arguments required unspecified. We will name these functions *Level_of_*a(...) through *Level_of_*d(...) and *Level_of_*u(...). As can be seen from the SRM and the guarded assignments, information flows into a from c when b=true, into a from d when b=false, and into u from c unconditionally. These flows are secure if and only if the targets of the flows have security levels that dominate those of the sources when the flow condition is satisfied. This gives rise to the following information flow formulas:

$$\begin{aligned}
\texttt{b} \rightarrow \quad & \textit{Level_of_}\texttt{a}(...) \succeq \textit{Level_of_}\texttt{c}(...) \\
\neg\texttt{b} \rightarrow \quad & \textit{Level_of_}\texttt{a}(...) \succeq \textit{Level_of_}\texttt{c}(...) \\
& \textit{Level_of_}\texttt{a}(...) \succeq \textit{Level_of_}\texttt{b}(...) \\
& \textit{Level_of_}\texttt{u}(...) \succeq \textit{Level_of_}\texttt{c}(...)
\end{aligned}$$

If each of these formulas can be proven, then the fragment is secure.

The general mechanism for generating formulas like the ones in the above example is as follows:

1. Define an appropriate security level for each system resource attribute. Levels may be defined as constants or as functions of system resource attributes. The same function may apply to a large number of resource attributes.

2. For each guarded flow indicated in the extended SRM, construct a formula of the form $G \rightarrow L_t \succeq L_s$ where G is the guard expression, L_t is the security level of the flow target, and L_s is the security level of the flow source. The first two formulas above are of this kind.

3. For each unguarded flow indicated in the extended SRM, construct a formula of the form $L_t \succeq L_s$ where L_t is the security level of the flow target, and L_s is the security level of the flow source. Unguarded flows include both flows from the resource attributes that appear in guard expressions (the third formula above) and flows from the sources of guarded assignments for which the guard expression is *true* (the last formula above).

5.2.2 Reasoning about SVCs

In mechanical flow tools, the attempts are made to prove SVCs with the aid of a mechanical theorem prover. Typically, the vast majority of the formulas are either trivially (read obviously) true or they can be proven by appealing to simple heuristics such as the fact that System High dominates all levels, etc. The same approaches can be used in analyzing manually generated SVCs, or even in the process of examining the SRM for possible security flaws. If the analyst cannot come up with a simple argument or informal proof for the security of a given flow, it should be put on a list for further examination and possible consideration as a security flaw.

5.2.3 Identifying security flaws

Those formulas that cannot be proven identify potential security flaws, however it is not always the case that an unprovable formula is a positive indicator of a flaw in the system. If the formula is untrue, constructing a counterexample may be a useful aid in identifying not only the flaw but in constructing a scenario for exploiting it. In many cases, the formula can neither be proven nor disproven. If this is the case, the system description may be missing some critical piece of information that would allow the proof to go through. In a manual analysis, it may be sufficient to appeal to that information, justifying the appeal with the appropriate sources, and declaring the flow to be secure.

If an SVC cannot be proven, it is still necessary to perform the analysis indicated in section 5.1.3 above to identify and characterize the flaw. In real systems, there are often many formulas that cannot be proven. Examination of the reasons results in a class of cases that will be discussed in the next section.

5.2.4 Formal flow violations

Formal flow violations are conditions that result in SVCs that are false or cannot be proven, but that cannot be shown to result in actual security flaws in the system. There is a tendency for some analysts to declare many of the formulas that they cannot prove to be the result of formal flow violations. Since some mechanical flow tools produce large numbers of unprovable or difficult to prove formulas, this can result in the overlooking of actual flaws. There are a number of situations that lead to formal flow violations that can be safely ignored. The safe approach is to understand the cause of each false or unprovable SVC and to provide a case for ignoring the apparent violation.

Some of the more common causes for the appearance of formal flow violations are:

1. The use of an inappropriate level of abstraction in describing an operation. Sets and mappings are often used in high level specifications to represent namespaces, file systems, etc. The operation that creates a new object may inspect the entire namespace to prevent duplicates and then modify it by adding the new entry. This results in the appearance of a bidirectional flow involving the size of the set even when names are polyinstantiated to prevent a namespace channel. No actual information is passed since the size of the namespace is not available directly, and the user is only able to know that it is greater than one after the insertion. Formulating the namespace in such a way that it is partitioned by level would avoid the formal flow violation but the system behavior would not change.

2. Conservative replacement in defining the flow semantics for the specification language being analyzed can suppress the detail needed to show that a given flow is secure.

3. The simplification of guard expressions can result in formal flow violations, particularly if the original and simplified forms have different orders of evaluation. This can be a problem in converting from forms that apply tests in sequence, such as nested `if ... then ... else ...` structures, to conjunctions of the nested tests.

5.3 Covert flow trees

CFTs[32] are a relatively new approach. Supported by an X-window based graphical tool, they provide a way of looking for scenarios that can be used to form covert channels. The information required to construct a covert flow tree is essentially the same information needed for the construction of a Basic Shared Resource Matrix. Each operation is characterized in terms of a reference list, a modify list, and a return list. The return list contains those resources that are referenced in modifying User Out. The CFT as presented by Porras and Kemmerer does not distinguish flows into multiple targets, but the extension to include this should be straight forward.

5.3.1 Constructing covert flow trees from dependency data

The construction of CFTs from the dependency information is similar to the piecemeal construction of the transitive closure of the SRM discussed above. The analyst identifies a resource attribute on which to focus the analysis. The tree is then constructed of actions that result in the modification of the resource attribute by the sender and recognition of the modification by the receiver. The left hand branch of the tree is the series of operations invoked by the sender to effect the modification; the right hand branch is the series of operations invoked by the receiver to recognize the modification. The trees are constructed by adding any operations that contain the target in their modify list to a direct recognition branch of the recognition path and those containing the target in their reference list to the inferred recognition branch. The process continues by adding direct and inferred recognition operations for the resources on the modify lists of the operations previously added recursively until all paths end in direct recognitions or a predetermined depth is reached and the remaining inferred recognition paths are terminated with labels of "false." A basic SRM, adapted from [31] is shown in figure 5.1. The covert flow tree for attribute "A" as derived from the SRM is shown in figure 5.2. Modification of an attribute occurs when an "M" appears in its row in the SRM. Direct recognition is not obvious from the basic form of the SRM, but is made explicit in the notation of [31]. Inferred recognition is due to transitive reads as discussed in section 5.1.4 above. The complete CFT as shown in figure 5.2 would be pruned by eliminating all paths that end with operations labeled "FALSE" prior to analysis. As can be seen from the SRM, no operations support the recognitions whose paths are identified in this fashion.

5.3.2 Identifying security flaws

Security flaws are identified using CFTs in much the same way as they are identified using the SRM, by finding a weak link and a scenario involving it. The CFT analysis yields sequences of operations to be performed by the sender and receiver to send information. These sequences are first simplified to remove pairs of operations that cancel and to add operations that must be used to establish an effective precondition for an operation in the sequence (e.g. opening a file before it can be written). The sequences are then examined to determine which ones represent legal operations and those are also removed. The remaining sequences are then analyzed to see if they contain a flawed operation that would allow the establishment of a covert channel.

Resource Attribute	Operation			
	Op 1	**Op 2**	**Op 3**	**Op 4**
A	M	R		R
B	M	M	R	M
C				M
D	R			
User In	R	R		R
User Out			M	M

Figure 5.1: A basic SRM

5.4 Non–interference formulations

The non-interference approach to covert channel analysis is advocated some developers and analysts; however, the techniques seem to have a large subjective component in the formulation of the view function. Is non-interference the *Emperor's new Clothes* of covert channel analysis?

5.4.1 Non-Interference

The non-interference approach formalizes the notion that one user should not be aware of any activity by another user that he does not dominate. This approach was introduced by Goguen and Meseguer [9] and has been applied to a number of systems including the SAT abstract model[10].

The technique requires that a view of the system state be constructed for each user. A system is said to be secure if a user's view of a state derived from a sequence of instructions is identical to a user's view of a state derived from the *purged* sequence of instructions. A purged sequence of instructions is produced by removing all instructions issued by users whose clearance is not dominated by that of the given user.

Proof of security is, in effect, by induction over all possible instruction sequences. The actual proof technique involves creation of an unwinding theorem that allows consideration of each operation independently. To avoid the problems associated with the independent treatment of operations (as in the information flow technique), the state view function is constructed so as to characterize potential views in states reachable from the one in question. Construction of the view function is nontrivial. The success or failure of the approach depends on the skill of the specifier in characterizing the system properly.

Although the intuitive notion of non–interference is simple, the analysis necessary is formal, based on a Formal Top Level Specification (FTLS) for the system in question and an equally formal description of the view function and unwinding theorem. To date, the literature does not contain any objective approach to the construction of the latter from the former. An early attempt[10] to apply the technique to the SAT, a very high level abstraction for a predecessor of the LOCK, found a level-based channel that was also identified using the SRM approach. The SRM approach also found a second channel based on subverting the SAT's domain mechanism. The non–interference approach did not find

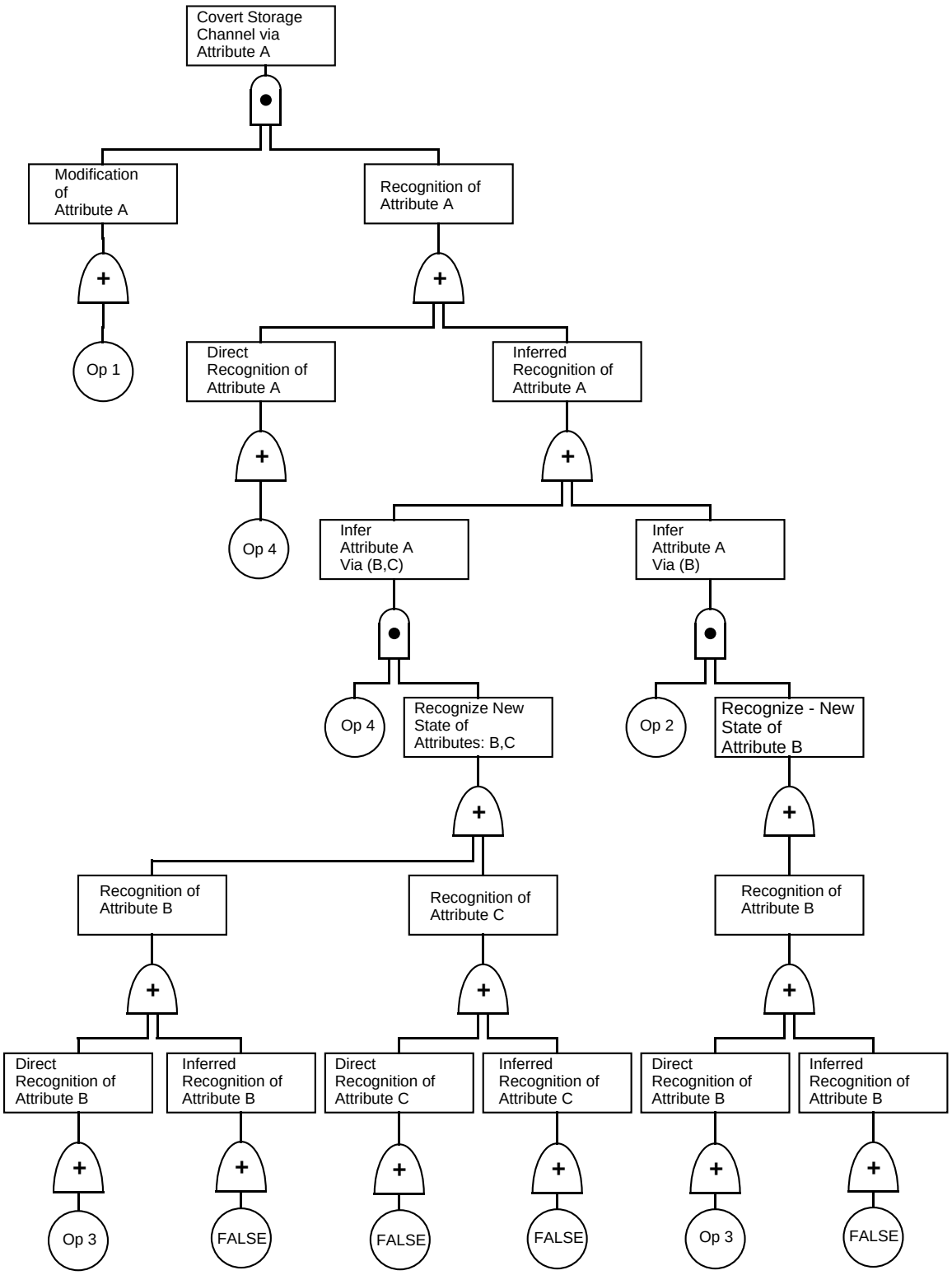

Figure 5.2: CFT for the basic SRM

this channel because the view function failed to consider domains. The CFT approach [31] has since discovered an additional domain based channel in the SAT specification.

The nature of the formulas that result from unwinding make relating their failed proofs to actual system flaws difficult at best. Using the non–interference approach without tool support is not likely to be fruitful. The technique may hold promise for A1 or beyond systems, but it should probably be avoided until the research community provides more concrete results and experience.

Section 6

Analyzing Covert Channels

Once a potential covert channel has been identified, its importance must be determined. This consists of determining its information carrying capacity as well as evaluating the importance and nature of the information that it can compromise. Trusted applications may have different criteria for analysis in these areas than general purpose systems and may be subject to different classes of threats. In addition to theoretical evaluations, experimental work may be required.

6.1 Exploiting Security Flaws

At the heart of every covert channel is a security flaw, that is, a situation in which invoking a system operation causes an information flow that violates the intent of the system's security policy. If the system is completely free of such flaws, it will be free of covert channels, but it is not necessarily the case that every flaw leads to one or more covert channels. For a usable covert channel to exist, it must be possible for one user to force a change in state through the flaw and for another, prevented by the security policy from communicating with the first, to detect the change. There may be many reasons that prevent the flaw from being exploited. Some of the more obvious ones might be:

1. The value of the resource does not, in fact, change. It is possible for the same value to be given a variable as it had before. It is not uncommon to unconditionally reset a flag at the end of an operation even though some paths through the operation render this unnecessary.

2. The change is not observable. The flaw is effectively "write only" and there is no way that the intended recipient can detect the change.

3. The change is always overwritten in the process of moving it to the recipient so that the information intended for transmission is destroyed before it can be read.

This is not to say that security flaws should be overlooked. Deciding that a flawed system does not present a covert channel is risky and should not be done lightly. Failure to identify an appropriate covert channel scenario for exploiting a given flaw may only mean that the analyst is not clever enough.

6.2 Covert Channel Scenarios

Once a flaw has been discovered, the analyst's job is to find a means by which the flaw can be exploited. Often the discovery of the flaw and the scenario occur simultaneously, especially when the analysis is guided by intuition rather than driven by formal techniques such as attempts to prove SVCs. There are no cut and dried, sure fire, approaches. The analyst needs a thorough understanding of the system and a good intuition for possible flaw mechanisms. The question of what is shared and why will probably provide the best clues. Unfortunately, scenarios for covert channels that have been discovered in operational systems do not appear in the literature and much of the covert channel analysis work that has been done on such systems appears to be classified. Reports such as [10] and [31] are useful because they provide detailed examples of such scenarios. See also the scenario in appendix B on page 78.

6.2.1 Half Bit Mechanisms

The transfer of a full bit of information over a covert channel requires that the receiver be able to tell that a bit was sent and what its value was. Since many signaling mechanisms only allow the receiver to determine that a change in a resource attribute has occurred, it is dangerous to assume that lack of a change has meaning. In this case, the mechanism has a capacity of one-half bit per use since it can signal either a "1" or a "0" but not both. Full bit mechanisms can be created from two half bit mechanisms. In searching for scenarios, the analyst should consider carefully those cases where only a change can be detected as these mechanisms may be far more common than higher capacity ones.

6.2.2 Signaling Protocols

A covert channel scenario defines a communications protocol between the sender and the receiver. The protocol may be unidirectional or bidirectional, synchronous or asynchronous. In a unidirectional protocol, the receiver is passive from the sender's viewpoint. The sender has no way of knowing whether the message is being received or not. This may be satisfactory if the sender is able to operate continuously, possibly using redundancy or error correcting techniques, or if it is possible to externally synchronize periods of operation so that there is some assurance that the receiver will get the message.

Bidirectional protocols involve an active receiver who is able to respond to the actions of the sender, controlling the rate of transmission and possibly the need for repetitions in the case of noisy channels. If the sender and receiver have tight control on their own scheduling and access to suitably accurate time references, synchronous operation is possible. In this case, half bit mechanisms can be used as full bit mechanisms since failure of a resource to change value in a fixed interval is just as positive a signaling mechanism as an observed change. In the absence of a mutual time reference, the operation is asynchronous and two half bit channels plus feedback from the receiver are required to create a full bit channel.

6.2.3 Talking to Outsiders

Covert channels are usually considered to have subjects of the system being analyzed as senders and receivers. In systems that are distributed, especially, there may be a threat

of compromise via signals intended for external observers who are not subjects under the control of a system or network TCB. Examples are signaling mechanisms involving packet headers, address modulations, frequency of transmission, etc. While signaling mechanisms of this sort are beyond the scope of this work, there is an increasing body of literature concerning them [4], and the techniques for identifying these mechanisms are similar to those discussed here. For example, it may be the case that a security flaw that appears to be "write only" with respect to the TCB results in an observable phenomenon to an outsider.

There is increasing economic pressure to use commercial, unsecured networks to connect secure systems, relying on encryption to protect the bodies of messages. A number of potential threats can be posited in such environments, and research is required to develop analysis techniques and countermeasures.

6.3 Trusted Applications - Trusted to do What?

Trusted applications are the subject of substantial controversy at the present time. One school of thought holds that conventional TCBs are adequate and that applications programs should not and need not be trusted. The other school holds that the notion of security enforced by a typical TCB is inadequate for many applications and that more flexibility is required so that some portion of the application will, of necessity, require trust and, effectively, extend the TCB. Both sides appeal more to rhetoric than to reason and experimental evidence to support either side is minimal.

One key aspect of the question is often overlooked. What is the trusted application trusted to do or not to do? If the application is to be given simultaneous accesses that violate the system's security policy, it must be trusted to enforce the intent of the policy, just as the TCB does. If the notion of trust involves some form of assured functionality, i.e. guaranteed delivery in a secure mail system, the application may not be in a position to compromise the MLS properties directly, but may require a high degree of assurance.

If the application is in a position to compromise MLS security, it may contain security flaws or otherwise be exploitable as part of a covert channel. Unfortunately, the systemwide nature of covert channel analysis does not seem to offer much hope for an effective incremental technique that can be applied to trusted applications. At the very least, it seems necessary to extend the basic system analysis to consider all cases in which flaws in a trusted application could be exploited in connection with information transfers in the TCB and vice versa. This is clearly another area that requires research.

6.4 Analyzing Threats

The NCSC's evaluated products list is based on a "one size fits all" philosophy to some extent. The evaluators pay only limited attention to the intended use of the systems being evaluated and the emphasis is increasingly on components or building blocks for systems. This leaves the consideration of the impact of security flaws and covert channels to accreditors and customers. The original guidelines for high assurance systems suggested that channels with bandwidths below 100 bits per second, the speed of a teletype machine, were acceptable. In reality, what is acceptable depends on what is being protected, what

its sensitive lifetime is, and how big it is. For example, high volume operational reports may be downgraded on a daily basis. If these are large enough and only compromise of the whole report is meaningful, the material may be publicly releasable long before it could be transmitted over a 100 bit per second line. On the other hand, if the system is used to manage small, highly sensitive, long-lived items, such as the master keys for an encryption key distribution system, covert channels capable of passing a few hundred bits per year may not be tolerable.

Another consideration is the threat environment in which the system resides. Systems that are widely accessible to a variety of cleared and uncleared personnel may be much more at risk than systems that operate in closed environments. In the early days of multi-level security, a stated goal was to build systems that would protect top secret material in environments as open as university computing centers. Few, if any, still consider this a reasonable goal, though the trend towards the use of public networks to transfer sensitive materials seem to pose similarly unrealistic goals.

Systems that are dedicated to particular missions, use a small, stable software base, and allow access only by cleared personnel probably are much less at risk than systems for general use. Network access serves to broaden the user community and further increase the risk.

6.5 Channel Capacity

The capacity of a covert channel is a function of three things:

1. The quantity of information that can be transmitted per execution of a scenario,
2. The time required to exercise the scenario, and
3. The effect of other system activities on the effectiveness of the transfer.

The last factor is often dependent on the environment, workload of the system, etc., and the conservative approach is to assume no degradation.

The covert channel analysis should provide an answer to the first question, the per execution capacity. The time required can often be estimated from the system's design characteristics, but should be verified experimentally. One factor that must be considered in analyzing channel capacity is the effect of performance enhancements. Upgrading a system with faster components is common and can have significant effects. Users and administrators need to understand the factors involved in channel performance and consider carefully the effects of potential upgrades.

In a uniprocessor system, context switching time may dominate many covert signaling mechanisms. In shared memory multiprocessors, this factor may be negligible. As the trend towards such systems accelerates, fast covert channels are likely to become much more common.

6.6 Countermeasures

Once covert channels have been found, there are a number of choices that can be made. These range from living with the channel to reducing its capacity to eliminating it alto-

gether.

6.6.1 Auditing

If an operation that contains a security flaw is infrequently used and leads to a low capacity channel, auditing may be a satisfactory countermeasure. Most secure systems have an audit requirement, so the means for recording the use of a flawed operation are already present. It is unlikely that the flawed operation is used only as part of a covert channel so some way to focus the audit analysis on malicious usage is also required. This may require auditing other operations in the scenario as well.

Auditing suffers from two problems. In the first place, use of the channel is typically discovered after the fact, possibly long after. In the second place, the analysis of audit data is time consuming and more art than science.

6.6.2 Reducing Channel Capacity

Increasing the time required to execute a covert channel scenario is one way to reduce the bandwidth of the channel. In general, it leads to unacceptable results as it also reduces the performance of the system. There appears to be one class of situations in which adding delay may be effective from a practical standpoint. When the normal usage of the flawed operation requires human interactions, it may be appropriate to add delays to prevent the same actions from being spoofed at electronic speeds by a program. Channels involving pointer movements, etc., in windowing systems could fall into this category.

Often channels are noisy because the sender's attempts to use a channel are interleaved with the activities of other users. These activities appear as noise or uncertainties in the values obtained by the receiver. It is sometimes possible to add artificial activities to increase the noise. These activities require system resources and may reduce the capacity of the system. The technique has been effectively used to thwart traffic analysis by presenting a constant picture of activity to the observer. There are no published instances of its usage for covert channel mitigation in computer systems.

It may also be possible to reduce the capacity of a covert channel by restricting the information that can be carried per scenario execution. For example, cut and paste in X windows requires an exchange of messages between the cutter and the paster. If we wanted to allow a high level user to paste a low level cutting using the normal exchange mechanisms, we could limit the choice of formats, conversion properties, etc., used in the message and place an upper bound on the capacity of each message to transfer information between the paster and cutter.

6.6.3 Closing the Channel

The only truly effective way to deal with a serious covert channel is to restructure the system so that it is eliminated. The importance of the early identification of security flaws cannot be overemphasized here since the cost of redesigning even a portion of a "finished" system can be very high. As multiprocessors become more common, channels such as the shared memory buss channel discussed informally at the IEEE Symposium on Security and Privacy in Oakland in 1991 likely to become more of a problem. This in turn, should lead

to new approaches to secure architectures that avoid sharing resources capable of providing high capacity channels across security perimeters.

Section 7

Evaluating a Covert Channel Analysis

This chapter has been primarily concerned with what a Covert Channel Analysis (CCA) is and how to conduct one. This section concentrates on evaluating the results of analyses performed by others. It provides guidelines for judging both the adequacy of the effort and the sufficiency of the evidence developed.

7.1 Looking at Plans

Any trusted system or application development proposal ought to address CCA. Careful attention to preliminary plans or approaches during source selection can prevent problems later. The detailed CCA plan ought to be an early deliverable even though the final CCA will not be performed until near the end of the development cycle.

7.1.1 What is to be analyzed?

The final covert channel analysis will be performed on the DTLS of the system, possibly augmented by an examination of low level shared resources such as devices and their drivers. The covert channel analysis plan should specify exactly the system representation that is to be analyzed. It should note whether the system representation is to be specially created or modified to support covert channel analysis. In general, the representation to be analyzed for covert channels should be the same one used to guide and describe the actual implementation of the system. If tools, such as those described in Appendix A, are to be used, special descriptive formats may be required, but otherwise they should be avoided. If they cannot be avoided, the plan should include the measures that will be taken to ensure that the system representation to be analyzed for covert channels is an accurate depiction of the system as a whole.

As was noted in Section 2, a specification intended for covert channel analysis must be complete and definitional. The covert channel analysis plan, as part of the overall system development plan, should indicate the steps to be taken ensure that the DTLS is suitable for covert channel analysis. This may not be a simple task as many system developers see security requirements as an imposition on their development style and attempt to isolate

security activities such as covert channel analysis so that the "real" developers can carry out business as usual. If this happens, the system as a whole will suffer. A careful examination of the covert channel analysis plan may help to identify a marginal or inadequate development plan.

The approach to be used for covert channel analysis should be appropriate for the system representation to be analyzed and the degree of assurance sought. For B2 systems intended for use in limited threat environments, an "ad hoc" analysis may be satisfactory, though a more rigorous approach is to be preferred. At the B3 assurance level, an analysis based on the shared resource matrix is the least that should be accepted.

Although the TCSEC does not require an analysis for timing channels at the B levels of assurance, there is increasing evidence that high speed timing based covert channels are relatively easy to construct in multiprocessor systems and in systems with intelligent peripherals such as modern disk controllers. A good covert channel analysis plan will address these issues when the system contains these kinds of shared resources. Inserting explicit requirements for a low level covert channel analysis examining the hardware and peripherals into the procurement documents should be considered.

7.1.2 When will CCA be done?

To avoid the possibility of substantial redesign late in development, preliminary CCA should be done throughout development, if for no other reason than to make sure that design decisions do not require the system to be insecure from a covert channel standpoint. If covert channel analysis activities are only planned for late stages of the system development, the development as a whole is placed at substantial risk if serious flaws are identified. A low risk development plan will include consideration of covert channel issues throughout the development cycle. Some discussion of covert channel issues should take place at preliminary design review (PDR) time. By the time Critical Design Review (CDR) occurs, the developer should be able to present preliminary analysis results to show that covert channels have been considered in all security relevant design decisions and that the decisions have been made to minimize security flaws that could be exploited as covert channels.

7.1.3 Who is doing the work?

The single most important factor in obtaining a good covert channel analysis is the quality and experience of the analyst. For this reason, the covert channel analysis plan should identify the analyst(s) and their background. It is preferable to have an experienced team or at least an experienced team leader. While a number of consulting companies offer training in covert channel analysis, academic exposure to covert channel techniques without practical experience is not sufficient to ensure adequate results. Ideally, the team members will have a combination of academic and practical experience in covert channel analysis. They will also be familiar with the system being analyzed.

Even small developments will benefit from a team approach to covert channel analysis. Much of the work is simply tedious, and it is easy to overlook details that are important. A very effective approach requires that individual team members convince the team of the security (or insecurity) of the information flows identified in the system. In the case of identified security flaws, team brainstorming is particularly useful in developing covert

channel scenarios to exploit the flaw. For these reasons, a plan that proposes a single analyst should be considered to be unsatisfactory.

The proposed use of consultants to perform a covert channel analysis should be judged carefully. It is likely that developers whose primary business is not the development of secure systems will not have staff members experienced in this area, and hiring a consultant may be the only way to obtain the required expertise. At the same time, the use of a consultant almost guarantees that covert channel issues will not receive adequate attention during the design and development process unless the consultant has a presence and some influence throughout the project. If a consultant is to be used to perform a covert channel analysis late in development, it is important that some of the developers have training or experience in the area so that covert channels will be considered during development.

7.2 Evaluating the Results

Once a covert channel analysis has been performed and a report on its findings prepared, its adequacy and results can be judged. There are a number of factors to be considered in evaluating a covert channel analysis.

7.2.1 What did they find?

Except in rare cases, some security flaws will be identified during a covert channel analysis. If none are reported, the design principles that prevented them should be carefully explained so as to convince the reviewer that this rather surprising result should have been expected. If covert channels are found, the underlying flaws should be identified and their presence justified in terms of some system requirement or requirements. The scenarios that permit the flaws to be exploited as covert channels should be exhibited along with an analysis of each scenario's information carrying capacity and bandwidth. Both analytical and experimental results should be given.

7.2.2 How is the investigation described?

A good covert channel analysis report should contain a description of the process followed that is sufficiently clear and detailed to convince the reader of the plausibility of the results. The description should identify the system representation(s) analyzed and their place in the development hierarchy. It should also describe the techniques used and the tools used, if any, to support the analysis. The qualifications of the team and the roles of the various team members should be described. The level of effort expended and the time period over which the effort was expended should also be described.

7.2.3 Looking at level of effort expended

The question of how much covert channel analysis is enough is a difficult one. The size of the system is one factor. More important is the richness of the system dependency structure. Well-designed systems decompose into subsystems that are only weakly interdependent. This greatly simplifies analysis and reduces the effort required. The effort expended should be proportional to the number of "M–R" dependency pairs that can be obtained from the

SRM describing the system. It is not possible to quantify the time required to consider each dependency since a single argument may be adequate to demonstrate the security of whole classes of dependencies. However, an expenditure of less than 10–15 minutes per dependency should be carefully justified. Expenditures in the 15 minute to one hour per dependency range are probably adequate, although the development of scenarios when flaws are discovered may require several days.

7.2.4 Detecting "hand waving"

Because of the large number of dependencies present in complex systems, the analyst may be tempted to dispose of flaws via "hand waving." The reviewer of a covert channel analysis must beware of this practice and attempt to identify it when it occurs. Signs of "hand waving" include the following:

1. Excessive use of the term "formal flow violation" to dismiss security flaws as insignificant. Each individual use of this term should be accompanied by an explanation that convinces the reader that the apparent flaw is not, in fact, a problem.
2. Wholesale justifications of the apparent security of large groups of apparently unrelated dependencies.
3. Statements to the effect that flaws that have been identified cannot be exploited without a convincing argument to support the claim.
4. Statements of any kind that indicate only a superficial knowledge of the system and its operation.
5. Unclear or unconvincing writing.

7.2.5 How did the developers respond?

If the covert channel analysis finds substantial flaws in the system, remedial action on the part of the developers is usually required. The developer's response to the CCA report is another measure of its adequacy. If the developers appear to understand the problems and are willing to work with the analysts to eliminate flaws or restrict use of the flawed mechanisms, there is ground for optimism. If the reaction is denial or confusion, this may indicate either an inadequate exposition of the analysis or immaturity on the part of the developers.

7.3 How To Hedge Your Bets

Procurement regulations usually preclude the customer from taking an active role in the design and development of a system. As a result, even if the customer has the knowledge and manpower to resolve design problems, there is little that can be done to force changes in the developer's approach. On the other hand, structuring the contract deliverables and reviews to ensure that covert channel issues are adequately addressed during development should be possible. Because of their system wide impact, covert channels have the potential to be

a "show stopper" and to prevent deployment of an otherwise useful and desirable system. Given the realities, it is clear that there is little that the customer representative can do in the face of an antagonistic or apathetic developer. The best approach is to structure the procurement so that covert channel issues are considered up front and throughout the development cycle. This can be done in a variety of ways:

1. Make a minimal level of covert channel analysis experience, as evidenced by a combination of training and practice, a prerequisite for participation in the procurement. Requiring that training be obtained early in the contract performance, if the expertise is not already available, could be made a condition for participation.

2. Require preliminary CCA to be done prior to PDR and again prior to CDR and schedule review of the results at these review meetings.

3. Monitor design decisions to ensure that Covert Channel issues are addressed on a continuous basis.

4. Require Covert Channel impact analysis on any changes of scope or any major change orders whether they originate with the customer or the developer.

5. Attempt to maintain a good rapport with the developers to identify problems as early as possible.

6. Develop a disaster plan. How can the mission that was to be supported by the procurement be sustained if the system proves to be unacceptable due to covert channels found late in the development cycle?

Section 8

Conclusions and Acknowledgments

This work has been a long time in preparation. It was begun in 1991 while the author was at the University of North Carolina. It went through a number of drafts, guided always by the helpful comments of the technical staff at NRL, but never quite finished. When the author moved to Portland State University in the fall of 1993, it was put aside for a bit and a semi final draft was produced in the fall of 1994. In May of 1995, Richard Kemmerer reviewed it and this final version incorporates his suggestions and corrects the typographical errors that he found. Comments and suggestions on earlier drafts came from numerous members of the technical staff at NRL, Judy Froscher, Oliver Costich, and Charles Payne, among others. Any remaining errors are mine. Over the years, my understanding of the problems associated with covert channel analysis in general has been sharpened by conversations with Dick Kemmerer who introduced me to the subject, with Bret Hartman Tad Taylor and Craig Singer who worked with me on the development of the Gypsy Information Flow Tool at Computational Logic, Inc. and with Charlie Martin.

As is the case with many aspects of computer security, this work is dated even before it is released. A recent paper by Steve Eckmann[5] explores ways in which a class of formal flow violations can be eliminated. The widespread growth of distributed computing environments and applications makes passing information from a TCB subject to the outside world a realistic possibility. Vehicles for this kind of signaling include message headers, manipulation of signaling protocols[25], etc. The high speeds with which both processors and networks operate make it possible for significant covert signaling bandwidths to be realized even with fairly elaborate signaling scenarios. Multiprocessor architectures in which processors operate at different security levels reduce or eliminate the need for context switches between the covert sender and receiver, greatly increasing the transmission rate.[38] Meanwhile, advances in the area of theories of secure composition hold some promise that we will be able to build systems from subsystems that are secure in some sense that implies the absence of covert channels with confidence that the resulting system is also secure in the same sense[27]. From a more immediately practical standpoint, developers are starting to investigate modular approaches to covert channel analysis that limit the amount of analysis necessary to determine what, if any, new covert channels have been introduced by system modifications or maintenance[14]. The author has some hope

that these techniques could prove useful in performing covert channel analyses on systems containing COTS (Commercial, Off The Shelf) subsystems.

The reader is urged to recognize that techniques for developing trusted systems are evolving rapidly. Follow the current literature in the field for new developments and abstract from the guidelines and techniques in this document to the situation at hand.

John McHugh
Portland State University
Portland Oregon
16 December 1995

Bibliography

[1] David E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR 2997, Mitre Corp., 1975.

[2] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *CACM*, 18-8, 1975.

[3] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[4] Albert Donaldson, John McHugh and Karl Nyberg. Covert channels in trusted lans. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[5] S. T. Eckmann. Eliminating formal flows in automated information flow analysis. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 30–38, May 1994.

[6] Jeremy Epstein, John McHugh Rita Pascale, Charles Martin, Douglas Rothnie, Hilarie Orman, Ann Marmor-Squires, Martha Branstad, and Bonnie Danner. Evolution of a trusted b3 window system prototype. In *1992 IEEE Symposium on Security and Privacy*, 1992.

[7] R. J. Feiertag, K. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proc. 6th Symp. on Operating System Principles*, pages 57–65. ACM, November 1977.

[8] M. Gasser, J. K. Millen, and W. F. Wilson. A note on information flow into arrays. Technical Report M79-234, MITRE Corporation, Bedford, MA, December 1979.

[9] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[10] J. T. Haigh, R. A. Kemmerer, J. McHugh and W. D. Young. Experience using two covert channel analysis techniques on a real system design. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 14–24. IEEE, 1986.

[11] D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1252, September 1988.

[12] Wei-Ming Hu. Lattice scheduling and covert channels. In *1992 IEEE Symposium on Security and Privacy*, pages 52–61, Oakland, CA, May 1992. IEEE Computer Society, Computer Society Press.

[13] Richard A. Kemmerer. Shared resource matrix methodology: A practical approach to identifying covert channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[14] Richard A. Kemmerer. Private Communication, November 1995. A paper on this work has been prepared and is awaiting publication approval.

[15] B. W. Lampson. A note on the confinement problem. *CACM*, 16(10):613–615, October 1973.

[16] Steven B. Lipner. A comment on the confinement problem. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 192–197, November 1975.

[17] John McHugh. *Towards Efficient Code from Verified Programs.* PhD thesis, The University of Texas at Austin, 1983. Also ICS-Report 40.

[18] John McHugh. Active vs. passive security models: The key to real systems. In *Proceedings of the 1987 Aerospace Security Conference.* AIAA, December 1987.

[19] John McHugh. A formal definition for information flow in the gypsy expression language. In *Proceedings of The Computer Security Foundations Workshop*, pages 147–165, Bedford, MA, June 1988. Mitre Corporation.

[20] John McHugh. An assignment in covert channel analysis. Technical Report TR-90-01, Baldwin/McHugh Associates, December 1990.

[21] John McHugh. A worked example in covert channel analysis. Technical Report TR-90-02, Baldwin/McHugh Associates, December 1990.

[22] John McHugh. An assignment in manual covert channel analysis. Technical Report TR-91-01, Baldwin/McHugh Associates, November 1991.

[23] John McHugh. A worked example in manual covert channel analysis. Technical Report TR-91-02, Baldwin/McHugh Associates, November 1991.

[24] John McHugh and Robert L. Akers. *Specification and Rationale for the Implementation of an Analyzer for Dependencies in Gypsy Specifications.* Computational Logic Inc., May 1987.

[25] John McHugh and Leslie Young. A taxonomy of signaling channels in atm networks, (with examples), , 1994. Technical Report TR 94-3, Computer Science Department, Portland State University, July 1994.

[26] John McLean. A formal method for the abstract specification of software. *JACM*, 31(3):600–627, July 1984.

[27] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 79–96, May 1994.

[28] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[29] Descriptive top-level specification: *Handbook for the Computer Security Certification of Trusted Systems*. Naval Research Laboratory, Washington, DC, 1994.

[30] Security policy model: A chapter of the *handbook for the computer security certification of trusted systems*. Naval Research Laboratory, Washington, DC, 1994.

[31] Phil A. Porras and Richard A. Kemmerer. Covert tree analysis approach to covert storage channel identification. Technical Report TRCS 90–26, University of California at Santa Barbara, Computer Science Department, December 1990.

[32] Phil A. Porras and Richard A. Kemmerer. Covert flow trees: A technique for identifying and analyzing covert storage channels. In *1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 36–51, Oakland, CA, May 1991.

[33] P.D. Reed and R.K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, Vol. 22(2):115–124, February 1979.

[34] John Rushby. Mathematical foundations of the MLS tool for revised SPECIAL. Draft internal note, Computer Science Laboratory, SRI International, Menlo Park, California, 1984.

[35] Marvin Schaefer. Personal Communication, December 1992.

[36] Craig D. Singer. An extension of the gypsy information flow semantics for dynamic and indexed types. Master's thesis, Duke University, April 1988.

[37] C. R. Tsai, Virgil D. Gligor, and C. S. Chandersekaran. A formal method for the identification of covert storage channels in source code. In *1987 IEEE Symposium on Security and Privacy*, pages 74–86, Oakland, CA, April 1987. IEEE Computer Society, Computer Society Press.

[38] *Unpublished.* A large bandwidth covert channel in shared memory multi-processors. Informal pannel session, 1991 IEEE Symposium on Security and Privacy, May 1991. At a panel presentation at the 1991 IEEE Computer Society Symposium on Research in Security and Privacy, a scenario was presented that resulted in a covert channel between the CPUs of a shared memory multiprocessor that could transfer several hundred megabits per second. I have been unable to find a written reference to this channel and would be grateful for further information – John McHugh.

[39] John C. Wray. An analysis of covert timing channels. In *1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7, Oakland, CA, May 1991.

Appendix A

Mechanical Tools for CCA

Several tools have been built to support the mechanical Covert Channel Analysis of formal specifications. While these are primarily of interest for systems being evaluated at the A1 level of the TCSEC, they are worth mentioning here.

At the time of writing, early 1993, it appears that the NSA which has maintained both the GVE and FDM systems for a number of years has dropped support for these systems, and it is not clear what their future availability may be.

A.1 The Gypsy Information Flow Tool (GIFT)

The GIFT is an integral part of the Gypsy Verification Environment (GVE). The GIFT is capable of producing both shared resource matrices and information flow formulas (SVCs) from Gypsy specifications that conform to the GIFT's specification conventions. These conventions require the user to define a single type for the security state of the system under analysis and to express the interface to the TCB of the system in terms of Gypsy procedures that have a "var" or modifiable parameter to represent the state. It is assumed that the TCB interface routines are invoked in an environment that contains the actual state. The specification of each TCB routine must be definitional in form and must contain an equality defining the output value of each var parameter in its exit assertion. The defining form must contain only input parameters, constants, and literals. The GIFT will reject specifications that do not conform to its conventions.

Given a conforming TCB specification, the GIFT performs a dependency analysis similar to that described in section 3 above. The output can be presented either as a basic SRM, a detailed SRM, or as lists of dependencies. If SVCs are desired, the user must define a security policy and supply a set of functions that return security attributes for the resources and resource attributes of the security state. Isolation (equality), total order, partial order, and lattice policies are supported, with the user being required to show that the security level comparison function possesses the appropriate mathematical properties for the type of policy declared. SVCs are constructed from the dependency and policy information. The SVC generator attempts to prove the formulas using heuristics, such as the fact that "model high" dominates all security levels, as well as using the GVE's algebraic simplifier. SVCs that cannot be proven by other means are left for the user to prove using the GVE's interactive theorem prover.

A.2 Ina Flow

The formal development methodology (FDM) uses a specification language called Ina Jo. The information flow tool incorporated into FDM is known as Ina Flo. Ina Jo specifications are based on a nondeterministic state machine model, so no special conventions are required to ensure that a specification being analyzed represents a state machine. On the other hand, nondeterminism is unacceptable in specifications intended for flow analysis. Ina Flo conservatively assumes that a nondeterministic specification references the entire state, which is unlikely to result in a satisfactory demonstration of security. The PreMLS component of FDM aids the user in writing deterministic specifications.

In using Ina Flo, the user is required to define a security type and a comparison function and is required to assign security levels to all state components. The tool generates formulas that the user must prove to show that the comparison function is reflexive and transitive, and to show that the constants, SysLo and SysHi (if used), are dominated by and dominate all levels, respectively.

Given an appropriately formed specification, Ina Flo performs an information flow or dependency analysis and produces SVCs based on the dependencies, the security labeling, and the comparison function. These SVCs are called flow conjectures in Ina Flo and are proved using the FDM interactive theorem prover.

A.3 The EHDM MLS Tool

EHDM is a derivative of the HDM or Hierarchical Development Methodology developed at SRI International during the 1970s and early 1980s. The original HDM had the first widely available covert channel tool [7]. Like its successor, the EHDM MLS tool requires a special specification form in which state variables are seen as indexed by security level information. The SVCs produced by the EHDM MLS tool are couched in terms of the index expressions for state variables that are modified and the sources of the modifications. In building the current EHDM tool, considerable effort has gone into optimizing the SVCs so as to facilitate their proofs. These optimizations go substantially beyond those used in the GIFT.

Although, at first glance, the approach taken to labeling in the EHDM MLS tool seems to be quite different from that used in the GIFT and in Ina Flo, the two are, in fact, equivalent from a covert channel analysis standpoint. In EHDM, the security labels are, in effect, built in by imposing the "indexed by security level" convention while the association of levels with state components is external to the specification per se in the other tools. This has the general effect of requiring a separate MLS specification when EHDM is used, although it is probably the case that the restrictions imposed by the GIFT and by Ina Flo also result in variants of a specification being constructed for CCA purposes.

A.4 Other Tools

The three tools mentioned above are the primary sources available to the analyst wanting to perform a covert channel analysis on a formal specification for an A1 system. Several others have been built to perform part of the analysis. Tsai's tool [37] supports the analysis

of C code. A dependency analysis tool has apparently been built at the Mitre Corporation. The covert flow tree tool has been discussed in section 5.3 above.

Appendix B

A worked Example

This Appendix provides a complete worked example of the CCA of an example application or system. The current example is based on [20] and [22].

B.1 Requirements

You are to design part of a multi-level secure file server for use within a secure distributed system. You may assume that the file server operates as a physically isolated node within the system, and that its only communications with the outside world are via messages sent in by its users, and its response to those messages. You may also assume that the server resides in a trusted environment, and that each request is properly labeled with the unique identity of the requester and with the requester's current security level. Note that this system is an over simplification, for example, the actual read and write operations have been omitted. A slightly more complete version of the system is described in [20] and analyzed using the Gypsy Information Flow tool in [21].

The file system supports the following operations.

Create File The user supplies the name of the file to be created. If the file name already exists at the level of the request, the request is refused. Otherwise the file is created and given the level of the request as its security level.

Delete File The user supplies the name of the file to delete, and if the file exists at the level of the request and is not in use by anyone other than the requester, it is deleted. Otherwise, the request is denied.

Open File for Reading The user must supply the name and level of the file to be read. If the file exists at that level, and the level of the request dominates the level to be read, and the file is not open for writing, the identity of the user is added to the list of readers for the file. Otherwise, the request is denied.

Open file for Writing If the file exists at the level of request and is not open for either reading or writing, the requester is recorded as having the file open for writing. Otherwise, the request is denied.

Close File The request will indicate the name of the file, as well as the level at which it is to be closed. If the file exists at the level of requester, and that is the requested level, and the requester has the file open for reading or writing, the file is closed. If the file exists at the level given in the request and the level of request dominates this level, and the requester is on the list of users that have the file open for reading, the requester is removed from the list. If the list becomes empty as a result, the file is closed. In any event, no response is sent to the user.

B.2 A Skeleton

The following provides a framework for one solution to the problem posed above. There are, of course, others.

B.2.1 Preliminaries

We need to define the vocabulary of the DTLS. Since we are not writing a formal specification, type definitions and data structures are not explicitly required. On the other hand, the specification will be easier to read if we have a set of conventions.

1. Requests for file system operations are issued on behalf of **Users**.
2. Each user has a security clearance at some **Level**.
3. The security levels form a lattice and can be compared with the **Dominates** operation which is denoted $\preceq$. We say that if $L_1 \preceq L_2$[1] then information is permitted to flow from objects classified at L_1 to objects classified at L_2.

B.2.2 Files

1. Each file in the system has a **Name**.
2. Each file is classified at some **Level**.
3. Names are unique within a level, but may be polyinstantiated, i.e. the same name may be used to denote different files at different levels.
4. Some mechanism must be provided to record whether a file is open and, if it is, whether it is open for reading or for writing.
5. Some mechanism must be provided to determine the identities of the users who have the file open.

B.2.3 The File System and Security State

1. The File System consists of the collection of files that exist at any given time.
2. The security state of the file system consists of the files and their explicit or implicit attributes.

[1] Read as L_2 *dominates* L_1 or as L_1 *is dominated by* L_2.

B.2.4 Requests and Responses

1. Each file system operation can be modeled as a request, possibly followed by a response.

2. The environment in which the file system exists can be assumed to ensure that each request is valid with respect to the **User** and **Level** information that it contains.

B.2.5 The TCB Interface

1. Without loss of generality, each file system operation can be modeled as a procedure with input parameters derived from a request and possibly output parameters that would constitute the contents of a reply.

2. These procedures can be viewed as operations on a global state that is otherwise inaccessible.

B.3 The DTLS

B.3.1 Data and Data Structures

By convention, we will use **bold face** type to refer to files and their components. We assume that each file has a logical structure as follows:

filename The name of the file.

label The classification of the file.

readers A list of the users who have the file open for reading. If this list is empty, the file is not open for reading.

writer The user who has the file open for writing. If this is empty, the file is not open for writing.

contents A list of data blocks that contain the information stored in the file. These are numbered sequentially from 1 to the size of the file. Since we are not modeling the actual read and write operations, this component is not strictly necessary.

Each file is accessed by its name and label, which are fixed when the file is created. The other elements may be referenced using a record like notation. For example, to refer the 7th data block of a secret file called "warplans" we would say:

file("warplans", "secret").contents(7)

By convention, we will use a sans serif font to refer to variables and formal parameters. In general, we will use the following variable names in the descriptions that follow:

user for the identity of the user making a request on the file system.

clearance for the security clearance of the user making a request. This variable can be used interchangeably with level in internal operations and is of the same type as the **label** component of a file.

name for the name of the file to which the request applies

level for the classification of the file to which the request applies. This variable can be used interchangeably with clearance in internal operations and is of the same type as the **label** component of a file.

set for either the **readers** or **writer** component of a file.

B.3.2 Internal routines

We will use a `teletype` font for TCB routines and their pseudocode descriptions. The following routines are used within the TCB, but are not accessible to outside users.

`Exists(name, level )` A function that returns **T** if a file with the given name exists at the given level, otherwise it returns **F**.

Providing a body for this routine would require us to develop the details of the mechanism that supports file lookup, etc. This could be done by using, say a mapping from {name×level} to the file attribute structures. For now, we will simplify things by assuming that the state of the file system is unstructured, but that mechanisms are available to provide functionality for this routine as well as for `NewFile` and `RemoveFile` below.

`ReadOpen(name, level )` A function that returns **T** if a file with the given name exists at the given level and the **readers** list of the file is not empty, otherwise it returns **F**.

```
ReadOpen ( name, level ) =
  if Exists ( name, level )
    then
      return ( file ( name, level ) . readers ≠ Empty )
    else
      return ( F )
    end
```

`IsReader (name, level, user )` A function that returns **T** if a file with the given name exists at the given level and the **readers** component of the file contains the user, otherwise it returns **F**.

```
IsReader ( name, level, user ) =
  if Exists ( name, level )
    then
      return ( user in file ( name, level ) . readers )
    else
      return ( F )
    end
```

`IsOnlyReader (name, level, user )` A function that returns **T** if a file with the given name exists at the given level and the **readers** component of the file contains only the user, otherwise it returns **F**.

```
IsOnlyReader ( name, level, user ) =
  if Exists ( name, level )
    then
      return ( user = file ( name, level ) . readers )
    else
      return ( F )
    end
```

Note that we use = here to mean that the **readers** component has exactly one element and that element matches the user in question.

`WriteOpen(name, level )` A function that returns **T** if a file with the given name exists at the given level and the **writer** component of the file is not empty, otherwise it returns **F**.

```
WriteOpen ( name, level ) =
  if Exists ( name, level )
    then
      return ( file ( name, level ) . writer ≠ Empty )
    else
      return ( F )
    end
```

`IsWriter (name, level, user )` A function that returns **T** if a file with the given name exists at the given level and the **writer** component of the file matches the user, otherwise it returns **F**.

```
IsWriter ( name, level, user ) =
  if Exists ( name, level )
    then
      return ( user = file ( name, level ) . writer )
    else
      return ( F )
    end
```

`Insert ( user, set )` This is a procedure that modifies either the **readers** or **writer** component of a file, denoted by set, by adding the user to it. We use the notation $\oplus$ to denote the insertion. If the user is already in the set the $\oplus$ operation has no discernible effect.

```
Insert ( user, set ) =
  set := set ⊕ user
```

`Remove ( user, set )` This is a procedure that modifies either the **readers** or **writer** component of a file, denoted by set, by removing the user from it. We use the notation $\ominus$ to denote the removal. If the user is not in the set the $\ominus$ operation has no discernible effect.

```
Insert ( user, set ) =
  set := set ⊖ user
```

`NewFile(name, level )` A procedure with a global effect on the system state. If a file with the given name does not exist at the given level, one is created having the given name as its **filename**, the given level as its **label** and with its **readers**, **writer**, and **contents** components empty. If a file with the given name exists at the given level, the operation has no effect on the system state.

`RemoveFile(name, level )` A procedure with a global effect on the system state. If a file with the given name exists at the given level, it is removed. If a file with the given name does not exist at the given level, the operation has no effect on the system state.

`Respond( ... )` A procedure that sends its arguments to the user who made the current request. If this routine is not called, the user receives no return information from the request.

B.3.3 The TCB routines

`Create File` The user supplies the name of the file to be created. If the file name already exists at the level of the request, the request is refused. Otherwise the file is created and given the level of the request as its security level.

```
Create_File ( user, clearance, name ) =
  if Exists ( name, clearance )
    then
      Respond ( "File name already exists." )
    else
      NewFile ( name, clearance )
    end
```

`Delete File` The user supplies the name of the file to delete, and if the file exists at the level of the request and is not in use by anyone other than the requester, it is deleted. Otherwise, the request is denied.

```
Delete_File ( user, clearance, name ) =
  if Exists ( name, clearance )
    then
      if (   (    ReadOpen ( name, clearance )
                  and IsOnlyReader ( name, clearance, user ) )
          or (    WriteOpen ( name, clearance )
                  and IsWriter ( name, clearance, user ) )
```

```
        or (    not WriteOpen ( name, clearance )
             and not ReadOpen ( name, clearance ) ) )
      then
        RemoveFile ( name, clearance )
      else
        Respond ( "File name is in use." )
  else
    Respond ( "File name does not exist." )
  end
```

`Open File for Reading` The user must supply the name and level of the file to be read. If the file exists at that level, and the level of the request dominates the level to be read, and the file is not open for writing, the identity of the user is added to the list of readers for the file. Otherwise, the request is denied.

```
Open_File_for_Reading ( name, level, clearance, user ) =
  if level ≼ clearance
    then
      if          Exists ( name, level )
         and not WriteOpen ( name, level )
        then
          Insert ( user, file ( name, level ) . readers )
        else
          Respond ( "Can't open name for reading at level " )
        end
    else
        Respond ( "Attempted Security violation" )
    end
```

`Open file for Writing` If the file exists at the level of request and is not open for either reading or writing, the requester is recorded as having the file open for writing. Otherwise, the request is denied.

```
Open_File_for_Writing ( name, clearance, user ) =
  if          Exists ( name, clearance )
     and not WriteOpen ( name, clearance )
     and not ReadOpen ( name, clearance )
    then
      Insert ( user, file ( name, level ) . writer )
    else
      Respond ( "Can't open name for writing" )
    end
```

`Close File` The request will indicate the name of the file, as well as the level at which it is to be closed. If the file exists at the level of requester, and that is the requested level, and the requester has the file open for reading or writing, the file is closed. If

the file exists at the level given in the request and the level of request dominates this level, and the requester is on the list of users that have the file open for reading, the requester is removed from the list. If the list becomes empty as a result, the file is closed. In any event, no response is sent to the user.

```
Close_File ( name, level, clearance, user ) =
  if Exists ( name, level )
    then
      Remove ( user, file ( name, level ) . readers )
      Remove ( user, file ( name, level ) . writer )
    end
```

Surprisingly simple. The `Removes` are no–ops if the user is not a reader or writer of the file in question. If the user is a reader or writer, it must be the case that an open succeeded and that the security checks were satisfied at that time. We assume that the user cannot change levels while holding open files.

B.4 DTLS Analysis

B.4.1 The Canonical State

Given the relative simplicity of the example, it is not surprising that the system state is correspondingly simple. The state components and attributes that are affected by the TCB operations are:

size (files) This attribute is affected by the creation and deletion of files.

files (N#) (L#) . readers This is an explicit component of each file. The notations **N#** and **L#** record the fact that the **filename** and **classification** are used to access a particular file and its components.

files (N#) (L#) . writer Another explicit component of each file.

files (N#) (L#) . contents Likewise. This is not mentioned in any operation except `NewFile`.

size (files (N#) (L#) . readers) A file attribute that can be affected by opening the file for read or by closing it.

size (files (N#) (L#) . writers) Another file attribute that can be affected by opening the file for write or by closing it.

domain (files (N#) (L#)) The name space for the file system is {name×level}. Creating a file modifies this name space as does deleting one.

B.4.2 Dependency Analysis

We analyze the DTLS to determine what interactions between the requesters and the state components occur for each request and each state component. We will record the results of our initial analysis as a simple shared resource matrix which is shown in figure B.1. To make the matrix more compact, we introduce some obvious abbreviations for the state components and for the operations.

State Component	TCB Operation				
	C_F	D_F	O_R	O_W	CL_F
size(F)	RM	RM	R	R	R
F(N#)(L#).R		R	M	R	RM
F(N#)(L#).W		R	R	RM	RM
F(N#)(L#).C					
size(F(N#)(L#).R)		R	M	R	RM
size(F(N#)(L#).W)		R	R	RM	RM
domain(F(N#)(L#))	RM	RM	R	R	R
User In	R	R	R	R	R
User Out	M	M	M	M	

Figure B.1: Basic Shared Resource Matrix

Several things are worth noting about the results. The first is that the accesses to the size attributes of the **readers** and **writer** components mimic those to the components themselves. This is to be expected since the access forms that we use do not reference individual elements. As a result, finding a match tells us that the size attribute is not zero, while not finding a match does not tell us that it is. Similarly, checking to see if a component is empty can tell us that its size is zero.

As expected, the contents component is never referenced. The explanation for the entries is as follows, along with a detailed shared resource matrix for each routine. In principle, nothing prevents the creating of SRM wallpaper for large systems, and such an approach is useful when looking for scenarios with which to exploit a flaw in the system.

`Create File`

The guard `Exists` references the file system namespace. If the guard is true, information is passed to the user. If it is false, the creation of the file modifies both the size of the file system and the namespace. One might be tempted to consider the components and attributes of the newly created file as modified as well, but we will not do this because their values are not dependent on the request in any way and the fact of their existence is reflected in the name space modifications.

```
Create_File ( user, clearance, name ) =
  if Exists ( name, clearance  )
    then
```

```
  Respond ( "File name already exists." )
else
  NewFile ( name, clearance )
end
```

Guard	Guard Expression
G1	Exists (name, clearance)
G2	not Exists (name, clearance)

Figure B.2: Guards for Create File

State Component	Create File		
	G1	G2	G2
size(F)	R	RM	R
F(N#)(L#).R			
F(N#)(L#).W			
F(N#)(L#).C			
size(F(N#)(L#).R)			
size(F(N#)(L#).W)			
domain(F(N#)(L#))	R	R	RM
User In	R	R	R
User Out	M		

Figure B.3: Detailed Shared Resource Matrix for Create File

This operation is relatively straightforward. The results are shown in figure B.3. Note that there are two flows under the second guard, one into the size attribute and one into the domain attribute. While we could combine these because they have exactly the same sources, we will treat them separately. In general, distinct targets have different sources and should be separated.

`Delete File`

The guard `Exists` references the name space. If it is not true, information is passed back to the user. If it is true, then an inner guard references the **readers** and **writer** components of the file in question. Again, information is passed back to the user, via a response if the file cannot be deleted, or the lack of a response if it is deleted. In the latter case, both the size and contents of the name space are modified.

```
Delete_File ( user, clearance, name ) =
  if Exists ( name, clearance )
```

```
then
  if (   (    ReadOpen ( name, clearance )
           and IsOnlyReader ( name, clearance, user ) )
       or (    WriteOpen ( name, clearance )
           and IsWriter ( name, clearance, user ) )
       or (    not WriteOpen ( name, clearance )
           and not ReadOpen ( name, clearance ) ) )
     then
       RemoveFile ( name, clearance )
     else
       Respond ( "File name is in use." )
else
  Respond ( "File name does not exist." )
end
```

Guard	Guard Expression
G1	Exists (name, clearance) → ((ReadOpen (name, clearance) and IsOnlyReader (name, clearance, user)) or (WriteOpen (name, clearance) and IsWriter (name, clearance, user)) or (not WriteOpen (name, clearance) and not ReadOpen (name, clearance)))
G2	Exists (name, clearance) → not ((ReadOpen (name, clearance) and IsOnlyReader (name, clearance, user)) or (WriteOpen (name, clearance) and IsWriter (name, clearance, user)) or (not WriteOpen (name, clearance) and not ReadOpen (name, clearance)))
G3	not Exists (name, clearance)

Figure B.4: Guards for Delete File

Again, there are two flows under the first guard as shown in figure B.5. The `if then` structure of the specification has been converted to an implication. This allows us to use the outer guard in proving the security of flows from the inner guard if we are attempting information flow proofs. If analysis of this routine were to indicate a possible security flaw, decomposing the guards further might help in isolating the problem. This is particularly true in the case of **G1** which controls a state modification.

State Component	Delete File			
	G1	G1	G2	G3
size(F)	RM	R	R	R
F(N#)(L#).R	R	R	R	
F(N#)(L#).W	R	R	R	
F(N#)(L#).C				
size(F(N#)(L#).R)	R	R	R	
size(F(N#)(L#).W)	R	R	R	
domain(F(N#)(L#))	R	RM	R	R
User In	R	R	R	R
User Out			M	M

Figure B.5: Detailed Shared Resource Matrix for Delete File

`Open File for Reading`

The outer guard compares the user's clearance with information provided by the user. In the Gypsy formulation, the former would be part of the state. Here it is a "trustworthy" parameter. If the outer guard is not satisfied, the user is notified. If the outer guard is satisfied, the inner guard references the name space and the **writer** component of the file in question. Under the true branch of the inner guard, the **readers** component of the file is modified which also modifies its size. Under the false branch, the user is notified.

```
Open_File_for_Reading ( name, level, clearance, user ) =
  if level ≼ clearance
    then
      if          Exists ( name, level )
         and not WriteOpen ( name, level )
        then
          Insert ( user, file ( name, level ) . readers )
        else
          Respond ( "Can't open name for reading at level " )
        end
    else
        Respond ( "Attempted Security violation" )
    end
```

Note that under the first guard of figure B.7, security level considerations play a role in determining whether or not the state is modified. Under the second guard, they determine whether the user receives a message. This is one place where we should look for a possible security flaw.

The third guard is interesting in that it doesn't reference the state at all. The user is presumed to know his or her own clearance. Asking for read access to a file at a higher level is either stupid or a mistake.

Guard	Guard Expression
G1	level $\preceq$ clearance $\rightarrow$ (Exists (name, level) and not WriteOpen (name, level))
G2	level $\preceq$ clearance $\rightarrow$ not (Exists (name, level) and not WriteOpen (name, level))
G3	not level $\preceq$ clearance

Figure B.6: Guards for Open File For Reading

State Component	Open File for Reading			
	G1	**G1**	**G2**	**G3**
size(F)	R	R	R	
F(N#)(L#).R	M			
F(N#)(L#).W	R	R	R	
F(N#)(L#).C				
size(F(N#)(L#).R)		M		
size(F(N#)(L#).W)	R	R	R	
domain(F(N#)(L#))	R	R	R	
User In	R	R	R	R
User Out			M	M

Figure B.7: Detailed Shared Resource Matrix for Open File For Reading

Open file for Writing

The guard references the name space, and the **readers** and **writer** components of the file in question. If it is true, the **writer** component of the file in question is modified which also modifies its size, otherwise the user is notified.

```
Open_File_for_Writing ( name, clearance, user ) =
  if          Exists ( name, clearance )
     and not WriteOpen ( name, clearance )
     and not ReadOpen ( name, clearance )
    then
      Insert ( user, file ( name, level ) . writer )
    else
      Respond ( "Can't open name for writing" )
    end
```

Guard	Guard Expression
G1	Exists (name, clearance) and not WriteOpen (name, clearance) and not ReadOpen (name, clearance)
G2	not Exists (name, clearance) or WriteOpen (name, clearance) or ReadOpen (name, clearance)

Figure B.8: Guards for Open File for Writing

State Component	Open File for Writing		
	G1	G1	G2
size(F)	R	R	R
F(N#)(L#).R	R	R	R
F(N#)(L#).W	RM	R	R
F(N#)(L#).C			
size(F(N#)(L#).R)	R	R	R
size(F(N#)(L#).W)	R	RM	R
domain(F(N#)(L#))	R	R	R
User In	R	R	R
User Out			M

Figure B.9: Detailed Shared Resource Matrix for Open File for Writing

In this case, the second guard of figure B.9 has been transformed to a more reasonable form.

`Close File`

Despite the simple formulation, complex flows are involved. The guard examines the name space. The remove operation both examines and modifies the component in question. A close inspection of the semantics of the $\ominus$ operation indicates that it should probably be treated as a guarded operation as well, making no change in the set if the element to be removed is not there. Note that there is no flow to the user under this operation.

```
Close_File ( name, level, clearance, user ) =
  if Exists ( name, level )
    then
      Remove ( user, file ( name, level ) . readers )
      Remove ( user, file ( name, level ) . writer )
    end
```

Guard	Guard Expression
G1	`Exists ( name, level )`
G2	`not Exists ( name, level )`

Figure B.10: Guards for Close File

State Component	Close File				
	G1	**G1**	**G1**	**G1**	**G2**
size(F)	R	R	R	R	R
F(N#)(L#).R	RM	R			
F(N#)(L#).W			RM	R	
F(N#)(L#).C					
size(F(N#)(L#).R)	R	RM			
size(F(N#)(L#).W)			R	RM	
domain(F(N#)(L#))	R	R	R	R	R
User In	R	R	R	R	R
User Out					

Figure B.11: Detailed Shared Resource Matrix for Close File

By looking into the `Remove` routines, we can see that the modifications of the **readers** and **writer** component are disjoint. If we treated the $\ominus$ operator as a conditional, the first guard of figure B.10 would expand into two distinct cases, each of which would have two

modify branches, exactly like those shown and a "no–op" branch. The second guard is shown only for completeness. Since no modifications occur under it, it does not need to be considered at all, and would usually be omitted.

B.5 Security Analysis

If the system has security flaws, they must involve operations in which the dependency analysis indicates a flow between security levels. Looking back at the specifications, we find that only the operations `Open File for Reading` and `Close File` can affect a file that is not at the level of the user who issues the request. Both of these operations involve a flow of information from the user to a component of the file. By itself, this is not a positive indication that a flaw exists. After all, it would be possible to treat the **readers** component of the file as being "system high" so that flows into it from any level would be secure.

Suppose we do this. We can now argue that `Open File for Reading` and `Close File` are secure since all their information flows are into the state are upward. `Open File for Reading` produces a flow to the user, but it doesn't depend on the **readers** component so we are still secure, provided the **writer** component of the file is classified at a level below the clearance of the requester. Let's consider it to be at the same level as the file. `Close File` has no flow back to the user, but it may alter the **writer** component of the file. We note that this will be the case only if the user seeking to close the file matches the contents of the **writer** component. This, in turn can only be the case if that user opened the file for writing, an act that required equality of levels. Thus `Close File` is secure as well.

Now we must look at the cases where the user can obtain information from the **readers** component of a file. From the basic SRM in figure B.1 on page 70, we can see that this occurs in `Delete File` and `Open File for Writing`. Inspecting the specifications and the detailed flows for `Delete File`, we see that a file cannot be deleted if its **readers** component contains a reader other than the user trying to delete the file. Since this information must be classified at a level dominated by that of the user seeking to delete the file, our previous assignment of "system high" to the component is generating a security flaw here. `Open File for Reading` manifests a similar problem. A bit of thought will convince us that there is no level that can be assigned to the **readers** component of a file that will solve the problem.

In addition, it is necessary to look at the possibility of a flaw involving the global, file-system-wide attributes. We note that both `Create File` and `Delete File` modify the size of the file system and its name space. Can either of these operations be flawed?

We argue that they are secure, despite the fact that the SRM indicates flows to and from them in both operations. Note that the references to these components are in terms of the `Exists` function, which checks to determine if the file system contains a particular file. If the file is found, the user knows that the size of the file system is at least 1 because it contains the file mentioned in the request. None of the operations releases any more information than that. On the other hand, we can argue that there is no way to assign a classification to these resources so that the information flows involved in creating and deleting files will be strictly secure. This is a good example of a formal flow violation.

Can anything be done to remove this non-flaw? One approach would be to look at what

the `Create File` and `Delete File` operations really need. Both must look at the portion of the name space that covers files at the level of the requested operation. If we kept a separate namespace for each level, these operations could be made secure as could all the operations that need to check for the existence of a file at a given level. In this case, the formal flow violation is due to representing an operation at too high a level of abstraction. While this is not always the cause of such violations, it is usually a good starting point for subsequent analysis.

B.6 The Channel

A high level user can open a file for reading that is at a level below him. This causes a flow of information from his level to the level of the file when the user's name is added to the user set of the file. This can be detected by a user at the level of the file when an attempt to open the file for writing fails. Closing the file by the high level reader will allow the low level write open to succeed. The high and low users can establish synchronization with an additional file into which the low level user writes to indicate a successful exchange.

For example, consider the following scenario where the files *sync*, *mark* and *space* are at the low level and the users are Low and High.

1. Low creates files *sync*, *mark* and *space*.
2. Low opens *sync* for writing, writes "I am ready" in it, then closes it.
3. High opens *sync* for reading, reads it, and closes it, repeating until the "I am ready" message is seen.
4. High opens *mark* for reading to send a 1, *space* to send a 0.
5. Low repeatedly attempts to open both *mark* and *space* for writing. If this succeeds for both, they are closed and this step is repeated. If low succeeds for one, but not for the other, a bit has been sent.
6. High closes the open file *mark* or *space*.
7. Low repeatedly attempts to open the file it failed to get in step 5) until it succeeds.
8. Low acknowledges the end of the cycle by writing a message to *sync* as in step 2) above.
9. High looks for the acknowledgment as in step 3) above.
10. The cycle repeats from step 4) until the message has been transmitted.

With mutual prearrangement, a very effective communications path can be set up. By using "private" files, i.e. ones that other users would not have any reason to open for reading, the channel can be quite noise free. Its bandwidth depends on system response time, but can be quite high if only a few milliseconds are required for a cycle of requests.