Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels

RICHARD A. KEMMERER
University of California, Santa Barbara

Recognizing and dealing with storage and timing channels when performing the security analysis of a computer system is an elusive task. Methods for discovering and dealing with these channels have mostly been informal, and formal methods have been restricted to a particular specification language.

A methodology for discovering storage and timing channels that can be used through all phases of the software life cycle to increase confidence that all channels have been identified is presented. The methodology is presented and applied to an example system having three different descriptions: English, formal specification, and high-order language implementation.

Categories and Subject Descriptors: C.2.0 **[Computer-Communication Networks]**: General—*security and protection*; D.4.6 **[Operating Systems]**: Security and Protection—*information flow controls*

General Terms: Security

Additional Key Words and Phrases: Protection, confinement, flow analysis, covert channels, storage channels, timing channels, validation

1. INTRODUCTION

When performing a security analysis of a system, both overt and covert channels of the system must be considered. *Overt* channels use the system's protected data objects to transfer information. That is, one subject writes into a data object and another subject reads from the object. Subjects in this context are not only active users, but are also processes and procedures acting on behalf of the user. The channels, such as buffers, files, and I/O devices, are overt because the entity used to hold the information is a data object; that is, it is an object that is normally viewed as a data container. *Covert* channels, in contrast, use entities not normally viewed as data objects to transfer information from one subject to another. These nondata objects, such as file locks, device busy flags, and the passing of time, are needed to register the state of the system.[1]

[1] Note that this definition of covert channels differs from that introduced by Lampson in his original note on the confinement problem [1]. The covert channels discussed in this paper include both storage and timing channels.

This research has been supported in part by the National Science Foundation under grant ECS81-06688.

Author's address: Computer Science Dept., University of California, Santa Barbara, CA 93106

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1983 ACM 0734-2071/83/0800-0256 $00.75

Overt channels are controlled by enforcing the access control policy of the system being designed and implemented. This policy states when and how overt reads and writes of data objects may be made. Part of the security analysis must verify that the implementation of the system correctly implements the stated access control policy. An example of verification is the UCLA Data Secure UNIX[2] project [2]. Access control is not further addressed in this paper.

Recognizing and dealing with storage and timing channels are more elusive. Objects used to hold the information being transferred are normally not viewed as data objects, but can often be manipulated maliciously in order to transfer information. In addition, the use of a storage or timing channel requires collusion between a subject with authorization to signal or leak information and an unauthorized subject. Note that the subject with authorization could be a malicious program acting without the knowledge of the user.

There are many examples of these channels and methods for blocking them [1, 3–7]. However, methods for discovering these channels have for the most part been ad hoc, giving little assurance that all storage and timing channels have indeed been discovered. The most systematic of these methods validates a specification for a multilevel, secure version of Multics [6]. In [6] an automated tool used formal specifications to generate tables describing which objects were read or written by a particular operation. However, before generating the tables each operation had to be divided into different parts, each mediated by a different subject. Previous work on flow analysis [8, 9] has also located storage and timing channels; however, these systems, like [6], were tightly coupled to a restricted subset of a particular specification language.

This paper presents a *shared resource matrix methodology* that can be applied to a variety of system description forms and which can increase the assurance (although it does not guarantee it) that all channels have been found. It is easily reviewed, disregards resources that are not shared, and is iterative as the design is refined or changed. It can be used in all phases of the software life cycle on systems whose constituent parts are in varying phases of development.

The next section introduces the methodology; Section 3 illustrates it, using an example system; and the last section discusses experience with the methodology.

2. THE SHARED RESOURCE MATRIX METHODOLOGY

Storage and timing channel analysis is performed in two steps in the shared resource matrix methodology. First, all shared resources that can be referenced or modified by a subject are enumerated, and then each resource is carefully examined to determine whether it can be used to transfer information from one subject to another covertly. The methodology assumes that the subjects of the system are processes and that there is a single processor which is shared by all of the processes. The processes may be local or distributed; however, only one process may be active at any one time.

To determine which shared resources can be modified or referenced one must first identify the shared resources. A *shared resource* is any object or collection of objects that may be referenced or modified by more than one process. It is

[2] UNIX is a trademark of Bell Laboratories.

RESOURCE ATTRIBUTE \ PRIMITIVE		WRITE FILE	READ FILE	LOCK FILE	UNLOCK FILE	OPEN FILE	CLOSE FILE	FILE LOCKED	FILE OPENED
PROCESS	ID								
	ACCESS RIGHTS			R		R		R	R
	BUFFER	R	M						
FILES	ID								
	SECURITY CLASSES			R		R		R	R
	LOCKED BY	R		M	R				
	LOCKED	R		R,M	R,M	R		R	
	IN-USE SET		R	R		R,M	R,M		R
	VALUE	M	R						
CURRENT PROCESS		R	R	R	R	R	R		

Fig. 1. Resource matrix filled in from English system description.

necessary to further refine each shared resource by indicating its attributes, because two processes may view different attributes of the same shared resource. For example, the first process may be able to determine only whether a shared file is locked, while the second process may only view the size of the file. Attributes of all shared resources are indicated in row headings of the shared resource matrix. Figure 1 is a matrix for the sample system discussed in Section 3.

Next, one must determine all operation primitives of the system being analyzed. Some examples of primitives are Write_File, Read_File, Lock_File, and File_Locked. The primitives of the system make up the column headings of the shared resource matrix.

After determining all of the row and column headings one must determine for each attribute (the row headings) whether the primitive indicated by the column heading modifies or references that attribute. This is done by carefully reviewing the description for each of the primitives, whether it is an English requirement,

formal specification, or implementation code. This task is performed differently for each phase of the software life cycle. (The example presented in Section 3 discusses the details of the different approaches.) The matrix generation is completed when each element of the matrix has been considered and marked, indicating whether a modification or reference could occur.

The generated matrix is then used to determine whether any channels exist. Two types of channels are considered: storage channels and timing channels. With a storage channel the sending process alters a particular data item, and the receiving process detects and interprets the value of the altered data to receive information covertly. With a timing channel the sending process modulates the amount of time required for the receiving process to perform a task or detect a change in an attribute, and the receiving process interprets this delay or lack of delay as information.

In order to have a storage channel, the following minimum criteria must be satisfied:

(a) The sending and receiving processes must have access to the same attribute of a shared resource.
(b) There must be some means by which the sending process can force the shared attribute to change.
(c) There must be some means by which the receiving process can detect the attribute change.
(d) There must be some mechanism for initiating the communication between the sending and receiving processes and for sequencing the events correctly. This mechansim could be another channel with a smaller bandwidth.

If criteria (a)–(c) are satisfied, one must find a scenario that satisfies criterion (d). If such a scenario can be found, a storage channel exists. This last step requires imagination and insight into the system being analyzed. However, by using the shared resource matrix approach, attributes of shared resources that do not satisfy criteria (a)–(c) can readily be identified and discarded.

Timing channels are discovered in a similar manner; however, different criteria are used. The minimum criteria necessary in order for a timing channel to exist are as follows:

(a) The sending and receiving processes must have access to the same attribute of a shared resource.
(b) The sending and receiving processes must have access to a time reference such as a real-time clock.
(c) The sender must be capable of modulating the receiver's response time for detecting a change in the shared attribute.
(d) There must be some mechanism for initiating the processes and for sequencing the events.

Any time a processor is shared there is a shared attribute: the response time of the CPU. A change in response time is detected by the receiving process by means of monitoring the clock.

For a channel to be of concern, the sending and receiving processes must be in distinct protection domains and must not be allowed to communicate with each other directly. Therefore, any channels that exist between processes in the same

protection domain can be ignored. In particular, if a process can sense only modifications made by itself, no channel exists.

Many storage and timing channels are a necessary part of the normal operation of the system; therefore, when a channel has been identified it is necessary to determine the bandwidth of the channel. That is, it is necessary to determine how many bits per second can be transferred between two cooperating processes using the identified channel. By determining the baud rate for a channel, one can decide whether to block the channel, add noise to decrease its bandwidth, or simply ignore it.

3. ILLUSTRATING THE METHODOLOGY ON A SAMPLE SYSTEM

The methodology has been successfully applied to the design of a secure network front end [10]; however, because the software architecture is proprietary, it could not be reported on in this paper. Instead, a pedagogical example is used. The advantage of using a toy system is that the process of applying the methodology is made more obvious to the reader. The danger of this approach is that the example begs the methodology, and the channels discovered may appear to be obvious. The example system considered here consists of two types of objects: processes and files. A process may read or write a file, open or close a file for reading, and lock or unlock a file for writing. It may also query to see whether a file is locked or opened.

The intent of the example is to show how the shared resource matrix approach can be used through the entire software life cycle to detect potential storage and timing channels. Discovery of a channel in the early phases of the software life cycle allows the designer to try to block the channel before too many design decisions have been made. However, constructing the matrix from an English description or a formal specification cannot uncover all channels. Therefore, it is important that the methodology also be applied to later phases of the software life cycle, particularly to implementation code. In the following sections an English description of the system is considered, then a formal specification, and finally implementation code.

3.1 English Requirements for the Sample System

Each process has a constant set of access rights. An *access right* consists of a security class and a read/write field. The *read/write* field indicates whether the process can read, write, or read and write objects of the indicated security class. Each file has a constant set of security classes. A file may be open for reading, locked for writing, or not in use. If a file is open for reading, then its in-use set contains the id's of the processes that currently have the file open for reading. If a file is locked for writing, then the value of its *locked by* attribute is the process that locked it; only this process can modify or unlock the file. For a process to read information from a file, each member of the file's security class set must exist in the access rights set of the process with either read or read/write access. If this is the case, then the process is said to have *read access* for the file. *Write access* is similarly defined.[3]

[3] The security model presented here is not the Bell-LaPadula security model [11]; however, both the *-property and the simple security condition can be represented using the proposed model.

Only one process, the current process, is active at a time. Each operation is uninterruptable and runs to completion before another is invoked. These restrictions avoid the combinatoric disaster that may result from introducing concurrency. More important, they are necessary if the system is to be formally verified. The operations are discussed in more detail in the following paragraphs.

The Write_File operation is used by a process to change the contents of a file. If the file is locked by the current process, the value of the file is modified to contain the contents of the current process's buffer.

The Read_File operation is used by a process to interrogate the contents of a file. If the current process is included in the in-use set for the file specified, the value of the file is copied to the current process's buffer.

The Lock_File operation is used by a process to modify the contents of a particular file. A process must lock a file before modifying it and must unlock the file after the modification is complete. If the current process has write access for the specified file, if the file specified is unlocked, and if its in-use set is empty, then the file is locked, and its locked by attribute is set to the id of the current process.

The Unlock_File operation makes a file accessible when a process is done modifying its contents. If the specified file's locked by attribute is the current process, the file is unlocked.

The Open_File operation is used by a process to initiate retrieval of the contents of a file. This primitive guarantees that no other process is modifying the contents of the file being interrogated. If the current process has read access for the specified file and the file is not locked, the current process's id is added to the in-use set for this file.

The Close_File operation is used when a process has completed interrogation of a file and wants to release it so that it can be modified. If the current process's id is an element of the in-use set for the specified file, then it is removed from that set.

The File_Locked operation is used by a process to determine whether a file is locked. If the current process has write access for the specified file, then, if the file is locked, a value of true is returned. If the file is unlocked the value false is returned. If the current process lacks write access for the specified file the result is undefined.

The File_Opened operation is used by a process to determine whether a file is open for reading. If the current process has write access for the specified file, then, if the file's in-use set is nonempty (i.e., the file is open for read), a value of true is returned. If it is empty the value false is returned. If the current process does not have write access for the specified file, the result is undefined.

For all operations, if the required conditions, such as file unlocked, are not met, then the operation has a null effect.

With this limited set of operations and no mechanism to cause a process to release a file, there is a potential for deadlock. In addition, a real system requires some fair method of scheduling processes, such as allowing each process to execute n operations before switching processes in a round-robin fashion. These issues, which are of concern in real-system design, are, for the most part, ignored in the remainder of the paper. However, an example of a timing channel premised on this approach to scheduling is presented in Section 3.2.3.

3.2 Applying the Methodology to the English Requirements

3.2.1 *Constructing the Matrix.* The first thing to do when applying the shared resource approach to the English requirements is to determine the objects and their attributes. There are two types of objects: processes and files. The attributes of a process are id, access rights, and buffer. The attributes of a file are id, security classes, locked by, locked, in-use set, and value. In addition, an object *current process* indicates which process is currently active.

The operational primitives of the system are the eight operations presented in the section above. Using this information, the skeleton of the matrix can be constructed and filled in by carefully determining whether the primitive indicated by each column heading modifies or references each attribute. When working with English requirements, keywords such as *checks*, *reads*, *if*, and *copy from* lead one to find attributes that are referenced. Keywords such as *change*, *set*, *replace*, and *copy to* lead one to attributes that are modified. Consider the description of Write_File:

> *If* the file is locked and the current process locked it, then the value of the file is *modified* to contain the contents of the current process's buffer.

When encountering the keyword *if*, one knows that what follows probably indicates attributes whose values are referenced. Therefore, for this operation the file's locked and locked by attributes, as well as the current process, are referenced. The keyword *modify* alerts one to look for what is modified and by what. For this operation the file's value attribute is modified using the process's buffer attribute. Thus the buffer, locked by, locked, and current process rows of the Write_File column contain Rs for reference, the value row contains an M for modify, and the other rows of this column remain blank. This process is repeated for all of the primitives, yielding the matrix of Figure 1.

The attributes referenced by one primitive may have been modified by another primitive, which referenced additional attributes. In order to illuminate these more sophisticated channels, involving multiple attributes, it is necessary to generate the transitive closure of the shared resource matrix. For instance, suppose an operation login references the password file and modifies the Active_User attribute. Furthermore, suppose a second operation references the Active_User attribute. The shared resource matrix for these two operations would indicate a reference to Active_User, but no reference to the password file in the column that corresponds to the second operation. However, it may be the case that the Active_User attribute is modified in a manner which compromises a user's password. Thus, it is necessary to indicate this indirect reference in the matrix. Then, when analyzing the matrix for possible channels, one must ensure that the modification to Active_User does not reveal information about user's passwords.

The transitive closure of the matrix is generated by looking at each entry that contains an R. If there is an M in the row in which this entry appears, then it is necessary to check the column that contains the M to see if it references any attributes that are not referenced by the original primitive. That is, if the column that contains the M has an R in any row in which there is not an R in the corresponding row of the original column, then an R must be added to that row in the original column.

RESOURCE ATTRIBUTE \ PRIMITIVE		WRITE FILE	READ FILE	LOCK FILE	UNLOCK FILE	OPEN FILE	CLOSE FILE	FILE LOCKED	FILE OPENED
PROCESS	ID								
	ACCESS RIGHTS	R	R	R	R	R	R	R	R
	BUFFER	R	R,M						
FILES	ID								
	SECURITY CLASSES	R	R	R	R	R	R	R	R
	LOCKED BY	R	R	R,M	R	R	R	R	R
	LOCKED	R	R	R,M	R,M	R	R	R	R
	IN-USE SET	R	R	R	R	R,M	R,M	R	R
	VALUE	R,M	R						
CURRENT PROCESS		R	R	R	R	R	R	R	R

Fig. 2. Transitive closure of matrix for English description.

For instance, consider the column for Write_File in Figure 1. There is an R in the locked row of this column, and the locked attribute is modified by the Lock_File primitive. Therefore, it is necessary to see which attributes were referenced to make this modification. The attributes access rights, security classes, locked, in-use set, and current process are referenced. Access rights, security classes, and in-use set are not directly referenced by the Write_File primitive, so they must be added to that column.

This process is repeated until no new entries can be added to the matrix. The resulting matrix is the transitive closure (with respect to references) of the original matrix.[4] The transitive closure matrix for the example system is shown in Figure 2.

Although the matrix construction has been performed manually, much of the generation could be automated. A prime candidate for automation is the gener-

[4] Note that this is not the standard mathematical transitive closure, since it relates to the modify operator as well as to the reference operator.

ation of the transitive closure of the matrix. This process is not dependent on the form of the system description; therefore mechanizing the process would not restrict the versatility of the approach. A Pascal program for generating the transitive closure of a matrix is presented in [12].

3.2.2 *Analyzing the Matrix.* Now that the shared resource matrix is complete, it may be used to locate potential storage and timing channels. In this section only storage channels are considered. An example of a timing channel is given in Section 3.2.3. From the criteria presented in Section 2 it can be seen that the only attributes that need be considered are those whose rows contain both an R and an M. Thus, for the example, only locked by, locked, in-use set, buffer, and value need to be considered.

For an attribute to be a potential storage channel one must be able to transfer information from one process to another in a direction that is not allowed by the access control mechanism. Therefore, it is not necessary to consider cases in which the access control mechanism requires the sending process to have write access and the receiving process to have read access to the same object; because, if they satisfy these requirements, the sender can modify the object and the receiver can reference the object. Thus, no storage channel is needed to communicate.

When analyzing a reference to a shared attribute, one can arrive at four possible conclusions:

(1) Another legal channel exists between the two communicating processes, so this channel is of no consequence.
(2) No useful information can be gained from this channel.
(3) The sending and receiving processes are the same.
(4) A potential storage channel exists.

In the following paragraphs an example of each of these conclusions is presented. The reader who is not interested in the details of the analysis for shared attributes may skip ahead to the last paragraph of this section, where the analysis is summarized.

The first attribute considered is the locked by attribute. This attribute can be modified only by the Lock_File primitive, and this requires the process executing the primitive to have write access to the file. Thus, the sending process must be in a protection domain that allows write access to the file specified. All of the primitives can reference the locked by attribute; therefore, it is necessary to determine for each of these references whether the reference can occur when the executing process is in a protection domain that does not require read access.

When the Write_File primitive is executed, the locked by attribute is referenced. If the value of the locked by attribute is the current process, then the locked by attribute was set by the current process (by executing a Lock_File). Since the process executing the Write_File primitive does not need read access, a potential storage channel may exist. However, the current process is the same process that modified the attribute, and this channel gains nothing. If the current process did not lock the file, then it can get no new information from the locked by attribute. That is, the current process only knows that it did not lock the

file—which it already knows anyway. Thus, no useful information would be gained by using the Write_File primitive to reference the locked by attribute.

The Read_File primitive requires the executing process to be in the in-use set. Since a process can become a member of a file's in-use set only by executing the Open_File primitive, the executing process needs read access in order to reference the locked by attribute. Therefore, the sending and receiving process can communicate directly through the specified file, and this is not a candidate storage channel.

The reference indicated for the Lock_File primitive is a transitive reference generated because the Lock_File primitive references the locked attribute, which is modified by the Unlock_File primitive, which in turn references the locked by attribute. The only information transferred by this reference is the fact that the process that last unlocked the specified file is the same process that locked it. Since this is always the case, no new information can be obtained from this indirect reference to the locked by attribute. There are a number of indirect references generated by the methodology, and each must be checked to see whether it can be used to transmit information that is not otherwise available.

None of the other references to the locked by attribute yield potential storage channels.

The in-use set attribute can be modified by the Open_File and Close_File primitives. The Open_File primitive requires the current process to have read access for the file in order to modify the in-use set, and the Close_File primitive requires the executing process to be a member of the in-use set for the modification to take place; therefore, the process must have read access for the specified file. Thus, both primitives require the executing process to have read access for the modification to take place. Since the protection domain of the modifying process must have read access, and the in-use set attribute can be referenced by all of the primitives, all of the primitives must be considered when searching for potential storage channels that use this attribute.

The Lock_File primitive references the in-use set attribute to determine whether it is empty. Whether the in-use set is empty can be detected by any process with write access; therefore, this attribute may be a potentail storage channel. The following scenario shows that this reference to the in-use set *can* be used as a storage channel. If the in-use set is empty, a process with read access could signal a 1 by executing the Open_File primitive, or a 0 by not executing the primitive or by executing a Close_File when the in-use set contains only that process's id. A process with only write access could then determine the setting by executing a Lock_File primitive and interpreting a successful result as a 0 and an unsuccessful result as a 1. (Note that this assumes that the file is not locked. Furthermore, since the Lock_File primitive does not explicitly return a success or failure code, the process will have to use the File_Locked primitive to check the result.) By using this procedure on a number of files to which the sender has read access and the receiver has write access, a large bandwidth channel can be achieved.

The Open_File and Close_File primitives reference the in-use set only to include/remove the executing process's id in/from the set. This reference provides no information to the executing process. However, if the in-use set were a finite

Table I. Summary of Matrix Analysis

Primitive Sensing Change:	Write File	Read File	Lock File	Unlock File	Open File	Close File	File Locked	File Opened
Attribute Modulated:								
Locked By	S	L	N	S	L	L	N	N
In-Use Set	N	S	**P**	N	N	N	N	**P**
Locked	S	L	**P**	S	L	L	**P**	N
Buffer	S	S	—	—	—	—	—	—
Value	S	S	—	—	—	—	—	—

Key for Table:
- L Legal channel exists with access control mechanism
- N No useful information can be gained from channel
- S Same process sending and receiving information
- **P** Potential covert channel

set whose maximum size was less than the number of processes that were allowed read access, then the set could be overflowed, causing a resource error. Thus, at the implementation level, where resources are finite and resource exhaustion can occur, more storage channels may exist.

A complete analysis of all of the shared attributes is presented in [12]. Table 1 contains a summary of this storage channel analysis. Two attributes that could be used as potential storage channels have been discovered. After the storage channels are located, each must be analyzed to determine its worst-case (i.e., largest) bandwidth. A decision is then made to determine whether to block the potential channel or ignore it.

3.2.3 *Timing Channels.* In order to provide an example of a timing channel, assume that the processes are scheduled in a round-robin fashion, with each process being allowed to execute n operations before giving up the CPU. In addition, assume there is another operation called Process_Sleep, which a process may invoke if it wants to give up the CPU before it has executed n operations. Finally, assume that each process has access to a real-time clock.

The closure of the shared resource matrix with the Process_Sleep operation added is shown in Figure 3. Notice that a process can modify the current process attribute by invoking the Process_Sleep operation. Thus the current process attribute must now be analyzed as a candidate channel. In analyzing this attribute for a storage channel, one discovers that the only information that the executing process can glean is that it (the executing process) is the current process, which is not useful information.

Next, this attribute is analyzed to determine if it can be used as a timing channel. The only information that a process can obtain is that it is the currently executing process, but if the executing process can determine how much time has elapsed since it last had control of the CPU, and if another process can vary this amount of time, then the current process attribute can be used as a timing channel. The following paragraphs present a scenario for using this channel.

Consider a sending process S and a receiving process R. Since S and R can surrender the processor at will, while remaining ready for reinvocation at the next

RESOURCE ATTRIBUTE \ PRIMITIVE		WRITE FILE	READ FILE	LOCK FILE	UNLOCK FILE	OPEN FILE	CLOSE FILE	FILE LOCKED	FILE OPENED	PROCESS SLEEP
PROCESS	ID									
	ACCESS RIGHTS	R	R	R	R	R	R	R	R	
	BUFFER	R	R,M							
FILES	ID									
	SECURITY CLASSES	R	R	R	R	R	R	R	R	
	LOCKED BY	R	R	R,M	R	R	R	R	R	
	LOCKED	R	R	R,M	R,M	R	R	R	R	
	IN-USE SET	R	R	R	R	R,M	R,M	R	R	
	VALUE	R,M	R							
CURRENT PROCESS		R	R	R	R	R	R	R	R	R,M
SYSTEM CLOCK		R	R	R	R	R	R	R	R	R

Fig. 3. Transitive closure of matrix for English description with timing example added.

scheduling slice, and the scheduling algorithm used is round-robin, S and R can take turns using the CPU. The scenario is as follows. S and R calibrate the process switch time by taking turns for a while. Call this time T_s. T_s has some variance, and could be multimodal in a system with recurring regular events, such as timer interrupts. S and R agree upon a code for transmitting messages, which may be based upon the results of the calibration (in which case S and R must arrive independently at the same code). The code must have the property that the normal variance of the process switch time will not result in transmission errors. Also, some selective noise rejection based upon the results of the calibration run can remove some regularly-occurring-event noise. (Note that "noise" is generated when a process other than S or R runs). Furthermore, only a fraction of the possible distinguishing code values is used to provide some detection of

Table II. Data Rates (baud) for Message Units (bits)

1	998	9	5952
2	1992	10	4940
3	2976	11	3608
4	3937	12	2354
5	4844	13	1414
6	5639	14	805
7	6205	15	456
8	6369	16	243

noise in transmission (i.e., the code works in the presence of noise to a degree determined primarily by the redundancy in the code).

Now, S sends a message M by consuming an amount of processor time which represents the coded version of M. R computes the amount of time which has passed since R last had control. It subtracts T_s from this. It now reconstructs the value of M corresponding to this time. Since the code is redundant, the computation may indicate a value not in the valid code set. To acknowledge M, R could give up the processor immediately (alternately, a subset of the code could be used to transmit positive acknowledgment). If the coded value is not a valid message, R acknowledges receipt negatively by consuming a particular amount of processor time before giving up the processor. This allows transmission in the presence of noise. S measures how much time has passed since giving up the processor. If it corresponds to correct receipt of M by R (e.g., T_s in the simplest case above), a new message is sent. If not, M is re-sent.

The scheme, if described correctly, transmits a message M only when S and R are the only ready processes on the processor for 1 cycle of the scheduler. More robust schemes are possible. The bandwidth of the scheme is related to the process switch time T_s, the processor speed, the resolution of the real-time clock which R and S use, and the variance of the process switch time and resulting possible unique code set size, and the amount of redundancy necessary to increase the probability of detecting noise to a sufficiently high level. For a worst-case example, assume that S and R are alone on the machine, there is no variance in the process switch time, there are no recurring events, and no redundancy is used. Assume that it is known that the computer can execute a specific number of instructions per second (which determines the resolution which the sender can use to send a code), and that the code used is simply a number of microseconds (implying the machine is fast enough to consume time in 1-microsecond units). A message is sent by breaking the message into small units, and consuming $2^{\text{message unit value}}$ microseconds. Simple coding theory and the measured value of T_s allow one to compute the maximum data rate. Sending 1 bit per T_s would allow a rate limited essentially by T_s. Sending more bits per T_s increases the efficiency at first, but the time it takes to send an additional bit in each message unit grows as $2^{\text{number of bits per unit}}$ grows, so a medium value must be chosen. For example, if the process switch time is 1 millisecond, and the code and timer resolution is 1 microsecond, one gets the data rates for the message unit sizes indicated in Table II. Obviously, these are higher than the rates one can obtain using redundant coding and operating in the presence of noise. Variance in the switch time will

rapidly lower the reliably distinguishable set of characters that can be transmitted. Regular noise can be filtered, but any noise, since it is generated by usage of the CPU, will also lower the bandwidth directly. Thus, the values presented are strict upper bounds for the channel described.

3.3 Formal Specifications

In this section the shared resource matrix methodology is applied to a formal specification of the example system. The formal specifications for the system are written in a variant of Ina Jo,[5] which is a nonprocedural assertion language that is an extension of first-order predicate calculus. The language assumes that the system is modeled as a state machine. The key elements of the language are types, constants, variables, definitions, initial conditions, a criterion, and transforms. The *criterion* is a conjunction of assertions that specify what a good state is. The criterion is often referred to as a state invariant since it must hold for all states, including the initial state. An Ina Jo language transform is a state transition function; it specifies what the values of the state variables will be after the state transition, relative to what their values were before the transition took place. A complete description of the Ina Jo language can be found in the *Ina Jo Reference Manual* [13].

Before giving the specification for the example system, a brief discussion of some of the Ina Jo notation is necessary. The following symbols are used for logical operations:

&	Logical AND
|	Logical OR
~	Logical NOT
→	Logical implication

In addition there is a conditional form

(if A then B else C),

where A is a predicate and B and C are well-formed terms.

The notation for set operations is

∈	is a member of
∪	set union
~~	set difference
$\{a, b\,..\,c\}$	the set consisting of elements $a, b, \ldots, c$
{set description}	the set described by set description.

The language also contains the following quantifier notation:

∀	for all
∃	there exists.

Two other special Ina Jo symbols that may be used are

N''	to indicate the new value of a variable (e.g., $N''v1$ is the new value of variable $v1$)
NC''	which indicates no change to the value of a variable.

[5] Ina Jo is a trademark of the System Development Corporation, a Burroughs Company.

The specification for the example system is shown in Figure 4. The eight transforms correspond to the eight operations of the English description.

```
TITLE Confinement
SPECIFICATION Confinement
LEVEL Top_Level
    TYPE
        Process,
        Processes = Set Of Process,
        File,
        Data
    TYPE
        Access = (read,write),   /* enumerated type */
        Accesses = Set Of Access,
        Security_Class,
        Security_Classes = Set Of Security_Class,
        Access_Right,
        Access_Rights = Set Of Access_Right
    CONSTANT
        Acc_Rights(Process):Access_Rights,
        Sec_Classes(File):Security_Classes,
        Class(Access_Right):Security_Class,
        Acc(Access_Right):Accesses
    CONSTANT
        OK_To(r:access,p:Process,f:File):Boolean =
           ∀s:Security_Class (
             s ∈ Sec_Classes(f) →
               ∃ a:Access_Right (
                  a ∈ Acc_Rights(p)
                & Class(a) = s
                & r ∈ Acc(a) ))
    VARIABLE
        Current_Process:Process,
        Locked_By(File):Process,
        Locked(File):Boolean,
        In_Use_Set(File):Processes,
        Value(File):Data,
        Buffer(Process):Data,
        Result(Process):Boolean

/* The INITIAL and CRITERION sections of the specification are
   used for the formal proof of the access control mechanism and
   are of no use to the covert channel analysis */

    INITIAL
        ∀f:File ( In_Use_Set(f)=Empty & ~Locked(f))

    CRITERION
        ∀p:Process,f:File (
             (p ∈ In_Use_Set(f) → OK_To(read,p,f))
           & (Locked(f) & Locked_By(f)=p → OK_To(write,p,f))
           & (Locked(f) → In_Use_Set(f)=Empty) )
```

Fig. 4. Ina Jo specification of example system.

```
TRANSFORM Write_File(f:File) External
   Effect
     ∀fl:File (
          N"Value(fl)=
           (if  fl=f
              & Locked(f)
              & Locked_By(f)=Current_Process
                    then  Buffer(Current_Process)
                    else  Value(fl)))

TRANSFORM Read_File(f:file) External
   Effect
     ∀pl:Process (
          N"Buffer(pl)=
           (if  pl=Current_Process
              & Current_Process ∈ In_Use_Set(f)
                  then  Value(f)
                  else  Buffer(pl) ))

TRANSFORM Lock_File(f:File) External
   Effect
     (if  OK_To(write,Current_Process,f)
        & ~Locked(f)
        & In_Use_Set(f)=Empty
            then  ∀fl:File (
              N"Locked(fl)=
                 (if fl=f
                     then  true
                     else  Locked(fl) )
            & N"Locked_By(fl) =
                 (if fl=f
                     then  Current_Process
                     else  Locked_By(fl) ))
            else  NC"(Locked,Locked_By) )

TRANSFORM Unlock_File(f:File) External
   Effect
     ∀fl:File (
          N"Locked(fl)=
             (if  fl=f
                & Locked_By(fl)=Current_Process
                    then  False
                    else  Locked(fl) ))

TRANSFORM Open_File(f:File) External
   Effect
     ∀fl:File (
          N"In_Use_Set(fl)=
             (if  fl=f
                & OK_To(read,Current_Process,f)
                & ~Locked(f)
                    then  In_Use_Set(fl) ∪ {Current_Process}
                    else  In_Use_Set(fl) ))

TRANSFORM Close_File(f:File) External
   Effect
     ∀fl:File (
          N"In_Use_Set(fl)=
```

Fig. 4. (continued)

```
                (if f1=f
                      then  In_Use_Set(f1) ~~ {Current_Process}
                      else  In_Use_Set(f1) ))


   TRANSFORM File_Locked(f:File) External
      Effect
        ∀pl:Process (
             N"Result(pl)=
                (if  pl=Current_Process
                   & OK_To(write,Current_Process,f)
                      then  Locked(f)
                      else  Result(pl) ))

   TRANSFORM File_Opened(f:File) External
      Effect
        ∀pl:Process (
             N"Result(pl)=
                (if  pl=Current_Process
                   & OK_To(write,Current_Process,f)
                      then  In_Use_Set(f) ≠ Empty
                      else  Result(pl)       ))

END Top_Level
END Confinement
```

Fig. 4. (continued)

3.4 Applying the Methodology to the Formal Specifications

When an Ina Jo specification is used, the variables are the attributes and the transforms are the primitives. Thus, Acc_Rights and Sec_Levels can be eliminated immediately, since they are declared to be constants in the Ina Jo specification. Also, since it is necessary to explicitly specify the result of the File_Locked and File_Opened transforms, there is an attribute, *result*, which was missing from the matrix generated for the English requirements. Therefore, the row headings of the matrix are Locked_By, Locked, In_Use_Set, Value, Buffer, Result, and Current_Process, and the column headings are the eight transform names.

To fill in the matrix one must determine which attributes are referenced and modified by each transform. Any variable that occurs in the effects section of a transform, preceded by the new value notation, is considered to be modified. All other attributes that are mentioned in the effects section, except those preceded by the no-change notation, are referenced. Consider the specification for the Write_File transform in Figure 4. Since the value attribute is preceded by N'', it may be modified by this transform. The attributes that occur in the effects section of the Write_File transform not preceded by N'' or NC'' are Locked, Locked_By, Current_Process, Buffer, and Value. Each of these is referenced by the transform. Thus, the column corresponding to the Write_File transform contains Rs in the Buffer, Locked_By, Locked, Value, and Current_Process rows and an M in the Value row.

RESOURCE ATTRIBUTE \ PRIMITIVE		WRITE FILE	READ FILE	LOCK FILE	UNLOCK FILE	OPEN FILE	CLOSE FILE	FILE LOCKED	FILE OPENED
PROCESS	BUFFER	R	R,M						
	RESULT							M	M
FILES	LOCKED BY	R	R	R,M	R	R	R	R	R
	LOCKED	R	R	R,M	R,M	R	R	R	R
	IN-USE SET	R	R	R	R	R,M	R,M	R	R
	VALUE	R,M	R						
CURRENT PROCESS		R	R	R	R	R	R	R	R

Fig. 5. Transitive closure of matrix for Ina Jo specification.

This process is repeated for each of the transforms and the transitive closure is computed. The resultant matrix is shown in Figure 5.

A problem may occur when the approach outlined above is used to determine which attributes are referenced in an Ina Jo specification. The problem arises because, when using the Ina Jo language, if a variable is to be changed under certain circumstances, but not others, all circumstances must be described explicitly. This is not enforced by the specification processor; therefore, the effect section of an Ina Jo transform may not be deterministic. For instance, a possible specification for the File_Locked transform is

```
∀p1:Process (
  N″ Result (p1) =
    (if p1 = Current_Process
      & OK_To(write, Current_Process, f)
        then Locked(f)))
```

Notice that this specification differs from the one that appears in Figure 4. The interpretation of this specification is that if the executing process has write access to file *f*, then the new value of the executing process's result attribute will be true if file *f* is locked and false if it is not locked. The problem is that it does not specify what the value of the result attribute will be if the executing process does not have write access to file *f*; nor does it specify what the new value of the result attribute for the other processes will be. That is, this specification is equivalent

to the following:

```
∀p1:Process (
  N″ Result(p1) =
    (if p1 = Current_Process
      & OK_To (write, Current_Process, f)
        then Locked(f)
        else N″ Result(p1)))
```

In the Ina Jo language the meaning of $N''var = N''var$ is that the variable *var* can assume *any* value in the new state. Thus, its new value can be the result of referencing any of the state variables. Therefore, when filling in the matrix for this type of specification, one must assume the worst case. That is, it is assumed that all state variables are referenced to determine the value of the result attribute. The column that corresponds to this transform would have an R in every row. Thus, whenever the effects section of a transform is nondeterministic, the user must assume that all shared attributes can be referenced.

The shared resource matrix generated from the Ina Jo specification is analyzed in the same manner as described for the English requirements matrix. Therefore, the discussion is not repeated here.

3.5 Implementation Code

In this section it is shown how the methodology is applied to implementation code. Each of the primitives is implemented as a Pascal procedure, and the attributes are the fields of the variables.

The procedure implementing the Write_File primitive might look as follows:

```
procedure writefile(fileid:filerange);
  begin
    if files[fileid].locked and
            (files[fileid].lockedby = currentprocess))
      then files[fileid].value := processes[currentprocess].buffer
  end;
```

To determine which attributes are modified, one need find only those attributes that appear on the left-hand side of an assignment statement (:=). In a complete implementation, however, these assignment statements are not only those that are explicit in the code for the operation. Consider an instruction that causes a page fault. This may result in an assignment to a system page table, indicating that the desired page is being swapped in or that another page is being swapped out. Thus, the possible side effects of each operation must also be considered. It should also be noted that these assignments are not only to variables in the software, but may also be the setting of some hardware register (e.g., a device register). For the *writefile* procedure the file's value field may be modified.

Finding which attributes are referenced by a procedure is more difficult. First, any attribute that appears on the right-hand side of an assignment statement may be referenced, since its value may be used to generate the value assigned. However, there are additional attributes which may be referenced to determine whether to make the assignment. These references are usually referred to as

implicit [14]. That is, any attribute whose value is used to determine which path to take in the program is referenced.

The attributes that are referenced implicitly by the *writefile* procedure are the *locked* and *lockedby* fields of files[*fileid*] and the *currentprocess*. In addition, the buffer is referenced directly.

After the direct and implicitly referenced attributes, as well as the modified attributes, are marked in the matrix, its transitive closure is generated in the same way as before. The shared resource matrix is now complete, and the analysis is performed as described in Section 3.2.

It should be mentioned that as the software life cycle progresses and more detail is added to the system design, the size of the matrix also grows. Therefore, the shared resource matrix for the implementation code is likely to be much larger than the matrix derived from the English requirements. The method of constructing and analyzing the matrix is, however, the same.

3.6 Other Phases of the Software Life Cycle

Although Sections 3.3 and 3.4 deal only with top-level specifications, the shared resource methodology may be applied to more detailed specifications in the same manner. The more detailed specification may introduce new attributes (e.g., the size of a file) and more transforms, and the transforms may have more parameters (e.g., offset in a file or buffer size). Therefore, the matrix will grow in size.

The shared resource matrix is also useful during the debugging and maintenance phases of the life cycle. If one wants to know which elements are affected by a particular attribute, it is only necessary to consult the matrix. For instance, before modifying a variable one can immediately determine which other attributes would be affected by the modification. Finally, if it is desirable to change the structure of some variable, one can determine from the matrix which procedures would be affected by the change.

As the system is modified, any changes in the attributes that are referenced or modified should be reflected in the shared resource matrix, and the changes to the matrix should be analyzed for possible storage and timing channels.

4. CONCLUSIONS

The shared resource matrix methodology has been successfully applied to the design of a secure network front-end [10]. This application has revealed a number of storage and timing channels. Of the channels discovered the worst-case bandwidth was 5000 bits per second, with a typical bandwidth of 20 bits per second. However, in practice, the bandwidth of these channels is much less, owing to the presence of noise and interference from other than the cooperating processes. As a result of the analysis the front end was redesigned to block or reduce the bandwidth of the channels discovered.

There are several advantages to using the shared resource attribute matrix to locate storage and timing channels, as opposed to searching for these channels in an ad hoc fashion. The first advantage is that by using the matrix, attributes that do not meet the preliminary criteria of being modified or referenced by a process are quickly discarded.

Another advantage is that, by presenting the shared resource information in graphical form, the information can be checked easily by those persons participating in the design, implementation, testing, and maintenance of the system, whether or not they are involved directly in the security analysis.

The matrix also serves as an excellent design tool. By indicating which attributes are affected by a primitive, design oversights that may have been left out of the preliminary design may be discovered. Also, if a primitive is to be changed, the attributes that may be affected are readily determined from the matrix.

Finally, since the process of generating the matrix is an iterative process, the matrix can be used throughout the software life cycle of the project as a design tool, as well as a security analysis tool. As the specifications become more detailed, more attributes and primitives are added to the matrix. Furthermore, since the methodology is not tied to a particular description form, it can be applied to a description whose constituent parts are described in different forms (e.g., English requirements and formal specifications). That is, part of the system may be implemented while other parts are only described by English requirements or formal specifications; but the methdology can be applied to the collection of all descriptions.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge Tom Aycock, Francis Chan, Tom Hinke, and John Scheid, who participated in the original development and application of this methodology to the secure network front-end design, and also Paul Eggert, Dino Mandrioli, Steve Bunch, and Gary Grossman, who reviewed earlier drafts of the paper and provided helpful comments.

REFERENCES

1. LAMPSON, B.W. A note on the confinement problem. *Commun. ACM 16*, 10 (Oct. 1973), 613–615.
2. WALKER, B.J. KEMMERER, R.A., AND POPEK, G.J. Specification and verification of the UCLA Unix Security Kernel. *Commun. ACM 23*, 2 (Feb. 1980), 118–131.
3. LIPNER, S.B. A comment on the confinement problem. In *Proc. 5th Symp. Operating Systems Principles.* (Austin, Tex., Nov. 19–21), ACM, New York, 1975.
4. MILLEN, J.K. Security kernel validation in practice. *Commun. ACM 19*, 5 (May 1976), 243–250.
5. SCHAEFER, M., GOLD, B., LINDE, R., AND SCHEID, J. Program confinement in KVM/370. In *Proc. 1977 Ann. ACM Conf.*, (Seattle, Wash., Oct.), ACM, New York, 1977, pp. 404–410.
6. AMES, S.R., AND MILLEN, J.K. Interface verification for a security kernel. In *System Reliability and Integrity*, vol. 2, Infotech State of the Art Rep., INFOTECH Int., Ltd., Maidenhead, Berkshire, UK, 1978.
7. KLINE, C.S. Data security: Security, protection, confinement, covert channels, validation. Ph.D dissertation, Computer Science Dept., Univ. of California, Los Angeles, 1980.
8. MILLEN, J.K., HUFF, G.A., AND GASSER, M. Flow table generator. MITRE Working Paper, WP-22554, The MITRE Corp., Bedford, Mass., Nov. 1979.
9. FEIERTAG, R.J. A technique for proving specifications are multilevel secure. CSL-109, SRI International, Menlo Park, Calif., Jan. 1980.
10. GROSSMAN, G.R. A practical executive for secure communications. In *Proc. 1982 Symp. Security and Privacy*, (Oakland, Calif., April 26–28, 1982). IEEE, New York, pp. 144–155.

11. BELL, D.E., AND LAPADULA, L.J. Secure computer systems. ESD-TR-73-278, vols. 1–3, The MITRE Corp., Bedford, Mass., June 1974.
12. KEMMERER, R.A. Shared resource matrix methodology: A practical approach to identifying covert channels. Rep. TRCS81-10, Computer Science Dept., Univ. of California, Santa Barbara, Nov. 1981.
13. LOCASSO, R., SCHEID, J., SCHORRE, V., AND EGGERT, P. *The Ina Jo Specification Language Reference Manual.* SDC Document TM-6889/000/01, System Development Corp., Santa Monica, Calif., Nov. 1980.
14. DENNING, D.E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 238–243.

Received September 1982; revised April 1983; accepted May 1983