# A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later

Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California Santa Barbara
kemm@cs.ucsb.edu

## Abstract

*Secure computer systems use both mandatory and discretionary access controls to restrict the flow of information through legitimate communication channels such as files, shared memory and process signals. Unfortunately, in practice one finds that computer systems are built such that users are not limited to communicating only through the intended communication channels. As a result, a well-founded concern of security-conscious system designers is the potential exploitation of system storage locations and timing facilities to provide unforeseen communication channels to users. These illegitimate channels are known as covert storage and timing channels.*

*Prior to the presentation of this paper twenty years ago the covert channel analysis that took place was mostly ad hoc. Methods for discovering and dealing with these channels were mostly informal, and the formal methods were restricted to a particular specification language. This paper presents 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. In the original paper the methodology was presented and applied to an example system having three different descriptions: English, formal specification, and high-order language implementation. In this paper only the English requirements are considered. However, the paper also presents how the methodology has evolved and the influence it had on other work.*

**Keywords:** 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 users. The channels, such as buffers, files, and I/0 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. This definition, which was presented in the original paper, differs from that introduced by Lampson in his original note on the confinement problem [13], because the covert channels discussed in this paper include both storage and timing channels.

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 realizes the stated access control policy. Access control is not addressed further 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 information and an unauthorized subject. Although there is concern that a user at a high security level may use a covert channel to signal or leak information to a user at a lower level, the major threat from a covert channel is its potential to be employed by a Trojan horse.

There are many examples of these channels and methods for blocking them [13, 14, 17, 19, 1, 12]. However, methods for discovering these channels have for the most part been *ad hoc*, giving little assurance that all storage and tim-

ing channels have indeed been discovered. The most systematic of these methods validates a specification for a multilevel, secure version of Multics [1], where 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 [18, 4] has also located storage and timing channels; however, these systems, like [1], were tightly coupled to a restricted subset of a particular specification language.

This paper presents the Shared Resource Matrix methodology, which is an approach that can be applied to a variety of system description forms and 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 applies the methodology to an example system, and Section 4 discusses experience with the methodology. Finally, the last section discusses what has happened to the Shared Resource Matrix methodology in the last twenty years.

## 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 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, one process may be able to determine only whether a shared file is locked, while another process may only view the size of the file. Attributes of all shared resources are indicated in row headings of the Shared Resource Matrix. Table 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 presents the details of applying the approach to English requirements.). 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 mechanism 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.

| Resource Attributes | 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 | | | | | | |
| **File** | | | | | | | | |
| 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 Proc** | R | R | R | R | R | R | | |

**Table 1. Resource matrix filled in from English system description**

(b) The receiving process 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 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 [5]; 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.

## 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 defined similarly.

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 Matrix 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 Table 1.

The attributes referenced by one primitive may have been modified by another primitive that 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 a 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.

For instance, consider the column for Write-File in Table 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. The transitive closure matrix for the example system is shown in Table 2.

Although the matrix construction has been performed manually, much of the generation could he automated. A prime candidate for automation is the generation 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. In fact, a program for generating the transitive closure of a matrix is presented in [8].

#### 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

| Resource Attributes | 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 | | | | | | |
| **File** | | | | | | | | |
| 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 Proc** | R | R | R | R | R | R | R | R |

**Table 2. Transitive Closure of the matrix for English description**

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 potential 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 set whose maximum size was less than the number of processes that were allowed read access, then the set could

| Attribute Modulated | Primitive Sensing Change | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | WRITE FILE | READ FILE | LOCK FILE | UNLOCK FILE | OPEN FILE | CLOSE FILE | FILE LOCKED | FILE OPENED |
| 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

**Table 3. Summary of Matrix Analysis**

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 [9]. Table 3 contains a summary of this storage channel analysis. Two attributes that could be used as potential storage channels were 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.

### 3.3. 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 Table 4. 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 invocation at the next 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. T 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 on 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. [1] Furthermore, only a fraction of the possible distinguishing code values is used to provide some detection of 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 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

[1] It should be noted that "noise" is generated whenever a process other than S or R runs.

| Resource Attributes | Primitive | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | WRITE FILE | READ FILE | LOCK FILE | UNLOCK FILE | OPEN FILE | CLOSE FILE | FILE LOCKED | FILE OPENED | PROC SLEEP |
| **Process** | | | | | | | | | |
| ID | | | | | | | | | |
| Access Rights | R | R | R | R | R | R | R | R | |
| Buffer | R | R,M | | | | | | | |
| **File** | | | | | | | | | |
| 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 Proc** | R | R | R | R | R | R | R | R | R,M |
| **System Clock** | R | R | R | R | R | R | R | R | R |

**Table 4. Transitive Closure of the matrix for English description with timing example added**

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 in the simplest case above), then a new message is sent. If not, then M is resent. For more details on this timing channel see [9].

### 3.4. Other Phases of the Software Life Cycle

In the original paper the Shared Resource Matrix methodology was applied to a formal specification of the example system and to implementation code. The formal specifications for the system were written in a variant of Ina Jo, 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. 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 [15]. When an Ina Jo specification is used, the state variables are the attributes and the transforms are the primitives. In the original paper it was shown how to determine which attributes were referenced and/or modified for each transform.

When dealing with implementation code the primitives are the procedures and the attributes are the fields of the variables. Again the original paper showed how to identify what variable fields were referenced or modified in each section of code.

Although the original paper dealt only with top-level formal specifications, the Shared Resource Matrix methodology can be applied to more detailed specifications in the same manner. The more detailed specification could introduce new attributes (e.g., the size of a file) and more transforms, and the transforms could have more parameters (e.g., offset in a file or buffer size). Therefore, the matrix would 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 what 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. Results

The Shared Resource Matrix methodology was successfully applied to the design of a secure network front-end [5]. This application 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 was 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 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 methodology can be applied to the collection of all descriptions.

## 5. The Last Twenty Years

In the mid 1980s there was an increasing interest in covert channel analysis. One of the reasons was that covert channel analysis was a part of the evaluation criteria used by the National Computer Security Center to classify secure systems. That is, the Trusted Computer System Evaluation Criteria [3] states that for a class B2 system "The system developer shall conduct a thorough search for covert channels and make a determination (either by actual measurement or by engineering estimation) of the maximum bandwidth of each identified channel."

During this time period some researchers began applying the non-interference approach, developed by Goguen and Meseguer [7], to systems. With this approach a subject S1 is said to be *noninterfering* with a subject S2 if no action taken by S1 can have an effect on the view S2 has of the system.

In 1986 both the Shared Resource Matrix (SRM) approach and the noninterference approach were applied to a high level design for a real system – the Honeywell Secure Ada Target (SAT). The SAT was intended to meet or exceed all of the requirements for A1 certification. A formal model of the SAT was expressed in the GYPSY formal specification language. Both techniques were applied to the Gypsy abstract model of the SAT. In this case, values for the initial Shared Resource Matrix were provided by the Gypsy flow analyzer, which analyzed the specification and/or code. Each operation in the matrix corresponded to a Gypsy function or procedure, and the flow analyzer determined what components of the program's security state were read and/or modified by each operation. For the noninterference approach the failed proofs of the unwinding theorems lead the analyst to the flows to consider, but, like the Shared Resource Matrix approach, it too did not aid the analyst in the actual analysis. With both approaches the analyst had to come up with the signaling sequences and determine whether they could be used as covert channels. Both methods were successful in detecting covert channels. A detailed discussion of the application of both techniques and the nature of the covert channels discovered can be found in [6].

The severity of a covert channel threat has been traditionally measured in terms of the channel's bandwidth (i.e., the number of bits signaled per second). The higher the bandwidth, the greater the potential for serious compromise. In the original application of the SRM to a secure network front-end the channel with the worst case bandwidth was 5000 bits per second. However, as operating systems were ported to faster hardware architectures, the bandwidths of their covert channels increaseed significantly. In 1990 the First Workshop on Covert Channel Analysis was held in Los Angeles [2]. At this workshop timing channels with estimated bandwidths in the megabits per second range were demonstrated on symmetric multi-processing architectures. Channels will only continue to get faster.

As mentioned above, the SRM approach and the noninterference approach were used to determine what attributes might be used for signaling information. Neither approach produced the sequence of operations necessary to covertly signal the information. In 1991 Kemmerer and Porras introduced tree data structures to model the flow of information from one shared attribute to another [10]. These trees were called Covert Flow Trees (CFTs). CFTs are used to perform systematic searches for operation sequences that allow information to be relayed through attributes and that are eventually detected by a listening process. When traversed, the paths of a CFT yield a comprehensive list of operation sequences that support communication via a particular resource attribute. These operation sequences are then analyzed and either discharged as benign or determined to be covert communication channels. That is, the analyst with his/her experience is still the one that makes the determination.

In 1993 McHugh introduced three extensions that added

more precision to the SRM approach: operation splitting, guard expansion, and user flows. Operation splitting factors the independent information flows that occur within the same operation into separate operations. Thus, a single operation in the system may appear as two or more smaller operations in the matrix. The guard expressions take advantage of the fact that conditional flows only take one branch at a time. User flows make the flows between a user's private resources and the system's shared resources explicit in the matrix. The details of McHugh's extension and examples of their use can be found in [16].

In 1996 Data General Corporation decided that they wanted their trusted DG/UX operating system to undergo a B2 evaluation. DG/UX is a full commercial-strength UNIX system with many features and support for a wide range of devices. The part of the system that was to be analyzed for covert channels was on the order of 800,000 lines of C code. The kernel was structured so that each of the elements of the system state was under the control of a single subsystem. That is, these elements could only be referenced or modified by functions of the controlling subsystem; thus, each subsystem could be thought of as an abstract object.

In order to make the covert channel analysis task for the Trusted DG/UX kernel more manageable and, in particular, to deal with the Ratings Maintenance Program (RAMP), a modular approach that takes advantage of the subsystem architecture was developed. The approach used leveraged off of the subsystem architecture of the DG/UX kernel. First, an SRM analysis was performed on each of the subsystems that contained an exported function directly invoked from one of the system calls. These subsystems were called "peer subsystems." The information from the SRMs for all of the peer subsystems was then used to build a kernel-wide SRM.

There are two major advantages to this modular approach to covert channel analysis. The first is that the covert channel analysis can be decomposed into a number of separable tasks, which can be distributed among many developers. The second is that the system can be incrementally reanalyzed as it changes over time. The details of this work can be found in [11].

## References


[1] S. Ames and J. Millen. Interface verification for a security kernel. Technical report, Infotech International, Ltd., Maidenhead, Berkshire, UK, 1978.

[2] *Minutes of the First Workshop on Covert Channel Analysis*. IEEE Cipher, July 1990.

[3] Department of defense trusted computer system evaluation criteria. DoD standard, DOD 5200.28-STD, December 1983.

[4] R. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International, Menlo Park, CA, January 1980.

[5] G. Grossman. A practical executive for secure communications. In *Proceedings 1982 Symposium on Security and Privacy*, pages 144–155. IEEE, April 1982.

[6] J. Haigh, R. Kemmerer, J. McHugh, and W. Young. An experience using two covert channel analysis techniques on a real system design. *IEEE Transactions on Software Engineering*, SE-13(2), February 1987.

[7] J.Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 Symposium on Security and Privacy*, pages 11–20. IEEE, April 1982.

[8] R. Kemmerer. Shared resource matrix methodology: A practical approach to identifying covert channels. Technical Report TRCS81-10, UCSB Computer Science Department, November 1981.

[9] R. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3), August 1983.

[10] R. Kemmerer and P. Porras. Covert flow trees: A visual approach to analyzing covert storage channels. *IEEE Transactions on Software Engineering*, SE-17(11):1166–1185, November 1991.

[11] R. Kemmerer and T. Taylor. A modular covert channel analysis methodology for trusted DG/UX. In *Proceedings of the Twelfth Annual Computer Security Applications Conference*, pages 224–235, December 1996.

[12] C. Kline. *Data security: Security, protection, confinement, covert channels*. PhD thesis, UCLA, 1980.

[13] B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(1):613–615, 1973.

[14] S. Lipner. A comment on the confinement problem. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, November 1975.

[15] R. Locasso, J. Scheid, V. Schorre, and P. Eggert. The Ina Jo specification language reference manual. Technical Report TM-6889/000/01, System Development Corp., Santa Monica, CA, November 1980.

[16] J. McHugh. Covert channel analysis, handbook for the computer security certification of trusted systems. Naval Research Laboratory Report, March 1993.

[17] J. Millen. Security kernel validation in practice. *Communications of the ACM*, 19:243–250, May 1976.

[18] J. Millen and G. Huff. Flow table generator. Technical Report WP- 22554, The MITRE Corp., Bedford, MA, November 1979.

[19] M. Schaefer, M. Gold, B. Linde, and J. Scheid. Program confinement in KVM/370. In *Proceedings of the 1977 Annual ACM Conference*, pages 404–410. ACM, October 1977.