A Practical Approach To Identifying Storage and Timing Channels†

Richard A. Kemmerer

Department of Computer Science
University of California
Santa Barbara, California

Abstract

Recognizing and dealing with storage and timing channels when performing the security analysis of a computer system is an elusive task. Methods of discovering and dealing with these channels for the most part have been *ad hoc*, and those that are not are restricted to a particular specification language.

This paper outlines a practical methodology for discovering storage and timing channels that can be used through all phases of the software life cycle to increase the assurance that all channels have been identified. The methodology is presented and its application to three different descriptions (English, formal specification, and high order language implementation) are discussed.

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 from one subject to another. That is, one subject writes into a data object and another subject reads from the object; thus, information is transferred from one object to another. The channels are overt because the entity used to hold the information is a data object; i.e., it is an object that is normally viewed as a data container. Examples of objects used for overt channels are buffers, files, and i/o devices. 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. *

Overt channels are controlled by enforcing the access control policy of the system being designed and implemented. The access control 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 project [2]. Access control is not further addressed in this paper.

† This research has been supported in part by the National Science Foundation under Grant No. ECS81-06688.

* 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 as discussed in this paper include both storage and timing channels.

Recognizing and dealing with covert channels is more elusive. As stated above, objects used to hold the information being transferred are normally not viewed as data objects, but can often be manipulated maliciously to transfer information. There are examples of these channels and methods for blocking them [1,3,4,5,6]. However, methods to discover these covert channels for the most part have been *ad hoc*, and assurance that all storage and timing channels have been discovered has been lacking. Previous work on flow analysis [7,8] has located covert channels; however, both of these systems are tightly coupled to a restricted subset of a particular specification language. This paper presents a methodology that can be applied to a variety of system description forms and can increase the assurance that all storage and timing channels have been found. It is easily reviewed, it disregards resources that are not shared, and it is iterative as the design is refined or changed. In addition, it can be used throughout all phases of the software life cycle. The methodology, called the shared resource matrix methodology, uses a matrix to graphically represent shared resources that might permit a channel.

The next section introduces the methodology; its application to three different system descriptions is demonstrated in Section 3; in Section 4 its advantages are discussed; and in the last sections, experience with the methodology, future work and conclusions are presented.

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 of these resources 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. Examples of common shared resources are files and devices. After all shared resources have been enumerated it is necessary to further refine each shared resource by indicating its attributes. For instance, the attributes of a file might be security_level, locked/unlocked, owner, size, and value. This step is necessary since, although two processes may be able to view the same shared resource, they may each view a different attribute of that resource. For example, the first process may be able to determine only whether a shared file is locked, while the second process may be able to determine only the size of the file. The attributes of all shared resources are indicated in the row headings of the shared resource matrix.

Next, it is necessary to 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 for the shared resource matrix one must determine for each attribute of each shared resource (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 different approaches are discussed in Section 3 and a detailed application using the methodology is presented in [9].) As an example, the write_file primitive might reference the security_level, locked/unlocked, and owner attributes of a file object and modify the value attribute. Figure 2.1 shows the matrix filled in for the write_file primitive; "R" indicates reference and "M" indicates modify. 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 shared resource matrix generated is then used to determine whether any covert information channels exist using these resources.

Two types of channels are considered in a covert channel analysis: 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 a bit of information. Each type of channel is considered separately in the following paragraphs.

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 sending and receiving processes and for sequencing the events correctly.

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 storage 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.

The matrix is used to determine whether criteria a, b, and c are satisfied. If they are, then 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, b, and c can readily be identified and discarded.

An example of a storage channel is as follows: process 1 can lock file A (lock_file primitive), and process 2 can detect whether file A is locked (file_locked primitive). Now, if one can initialize and sequence process 1 and process 2, then process 1 can signal to process 2 ones and zeroes by locking and unlocking file A. The file lock is not considered a data object that can be read and written directly, but reading and writing can occur anyway.

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

a. Both the sending and receiving process must have access to the same attribute of a shared resource.

b. Both the sending and receiving process must have access to a time reference such as a real-time clock.

c. The sender must be capable of modulating the receiving process'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.

It is important to note that any time a processor is shared there is a shared attribute which is the response time of the cpu. A change in response time is detected by the receiving process by monitoring the shared resource attribute and using the clock to determine the amount of time for a change to occur. Again for a timing 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.

Again, the shared resource matrix is used to eliminate shared attributes that cannot be used as timing channels.

Once a storage or timing 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.

3. Application to Different System Descriptions

The intent of this section 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. First, its application to an English description of the system is considered, then to a formal specification, and finally to implementation code.

3.1 Applying the Methodology to English Requirements

3.1.1 Constructing the Matrix

The first thing to be done when applying the shared resource approach to the English requirements is to determine what are the objects of the system and what attributes they possess. Consider a system which consists of two types of objects: processes and files. The attributes of a process are ID, Access Rights, Buffer, and CurrentProcess. The attributes of a file are ID, Security Classes, Owner, Locked, In-Use Set, and Value.

After identifying the attributes, it is necessary to identify the operational primitives of the system. These are the operations used to change the state of the system (eg. the system calls if dealing with an operating system or kernel calls if dealing with a security kernel). Using this information the skeleton of the matrix can be constructed with the attributes as row headings and the primitives as column headings.

The skeletal matrix is now filled in by carefully determining for each attribute of each shared resource whether the primitive indicated by each column heading modifies or references the attribute. When working with English requirements key words such as "checks", "reads", "if", and "copy from" lead the user to find attributes that are referenced and key words such as "change", "set", "replace", and "copy to" lead the user to attributes that are modified. Consider the description of Write_File:

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

When one sees the keyword "if" then he knows that what follows probably indicates attributes whose values are referenced. Therefore, for this operation the file's Locked and Owner attributes, as well as the Current Process are referenced. The keyword "modify" alerts the user to look for what is modified and with what. For this operation the file's Value attribute is modified using the process's Buffer attribute. Therefore, the Value attribute of file is modified and the Buffer attribute of the process is referenced. Thus, the Buffer, Owner, Locked, and Current Process rows of the Write_File column contain Rs, the Value row contains an M, and the other rows of this column remain blank. This process is repeated for all of the primitives. An example matrix is shown in Figure 3.1.

Since the attributes referenced by one primitive may have been modified by another primitive that referenced additional 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 storage and timing channels one must assure that the modification to active_user doesn't 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 and 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 the matrix presented in Figure 3.1. Since Write_File references the file's Locked attribute there is an R in the Locked row of this column. However, if the Locked row is scanned one sees that the Locked attribute is modified by the Lock_File primitive. Therefore, it is necessary to see what attributes were referenced to make this modification. This is done by checking which rows of the Lock_File column contain Rs. A quick check reveals that the attributes Access Rights, Security Classes, Locked, In-Use Set, and Current Process are referenced. Of these five 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. Thus, the Write_File column now contains Rs in rows Access Rights, Buffer, Security Classes, Owner, Locked, In-Use Set, and Current Process.

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.* This completes the first step of the shared resource matrix approach.

3.1.2 Analyzing the Matrix

Now that the shared resource matrix is complete it is necessary to use it to locate potential storage and timing channels. 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 matrix of Figure 3.1 the only attributes that need to be considered are Owner, Locked, In-Use Set, Buffer and Value.

For discussion purposes assume that the access policy is defined as follows. For a process to read information from a file each of the security classes in 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. For a process to write information to a file each of the security classes in the file's security class set must exist in the access rights set of the process with either write or read/write access. If this condition is satisfied the process is said to have "write access" for the file.†

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, if the sending process (that is, the process that modifies the attribute) is required to have write access, then a storage channel exists only if the receiving process (that is, the process that references the attribute) is not required to

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

† The security model presented here is not the Bell-LaPadula security model [10]; however, both the *-property and the simple security condition can be represented using the proposed model.

have read access. That is, it is not necessary to consider cases where the the access control mechanism requires the sending process to have write access and the receiving process to have read access, because if they satisfy these requirements the sender can write the file and the receiver can read the file. Thus, no storage channel is needed to communicate.

The analysis discussed above deals only with storage channels. To provide an example of a timing channel, assume 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 an 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 3.2. 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 storage channel. Analyzing this attribute for a storage channel one discovers that any process can modify Current Process and any process can reference this attribute. However, the only information that the executing process can glean from this attribute is that it 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.

After locating the storage and timing channels 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. Example storage and timing channels complete with baud rate estimates are presented in [9].

3.2 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 Ina Jo*, a non-procedural 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, criterion, and transforms. The criterion is a conjunction of assertions that specify what is a good state. 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 [11].

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 conditional form

(A => B <> C)	if A then B else C.

The notation for set operations is:

<:	is a member of
||	set union
~~	set difference
S"(a,b..c)	the set consisting of elements a,b..and c.

The language also contains the following quantifier notation:

A"	for all
E"	there exists

Two other special Ina Jo symbols that are used in the example specification are:

N"	to indicate the new value of a variable (eg. N"v1 is new value of variable v1)
NC"	which indicates no change to a variable.

A partial system specification is shown in Figure 3.3. The transforms correspond to the operations of the English description.

3.2.1 Applying the Methodology to the Formal Specifications

As was the case with the English requirements the user first must identify the attributes that can be modulated and the primitives of the system. When using an Ina Jo specification the variables are the attributes and the transforms are the primitives. Notice that 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 that was missing from the matrix generated for the English requirements. Therefore, the row headings of the skeletal matrix are Owner, 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. When working with an Ina Jo specification 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. 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, Owner, 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, Owner, Locked, Value, and Current_Process rows and an M in the Value row. This process is repeated for each of the transforms.

Entries must then be added to the matrix by computing its transitive closure (with respect to references).

* Ina Jo is a trademark of System Development Corporation.

The same process that was used for the matrix constructed from the English requirements is applied again. The resultant matrix is shown in Figure 3.4.

A problem may occur when using the approach outlined above 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:

```
A"p1:Process (
  N"Result(p1) =
   ( p1 = Current_Process
    & OK_To(write,Current_Process,f)
      => Locked(f)))
```

Notice that this specification differs from the one that appears in Figure 3.3. 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. However, it does not specify what the value of the Result attribute will be if the executing process doesn't 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 specification.

```
A"p1:Process (
  N"Result(p1) =
   ( p1 = Current_Process
    & OK_To(write,Current_Process,f)
      => Locked(f)
      <> 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. Notice how this differs from the value for the File_Locked column in Figure 3.4. 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 was described for the English requirements matrix.

3.3 Implementation Code

In this section the methodology is applied to implementation code. Figure 3.5 is an incomplete Pascal-like implementation of the example system. Each of the primitives is implemented as a Pascal procedure, and the attributes are the fields of the variables.

To determine which attributes are modified one need find only those attributes that appear on the left-hand side of an assignment statement (:=). Thus, for the writefile primitive the Buffer attribute 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 that may be referenced to determine whether to make the assignment. These references are usually referred to as conditional references. That is, any attribute whose value is used to determine which path to take in the program is referenced. For instance, if the program contains the following piece of code:

```
if attribute1=cons
    then attribute2:=attribute3
```

then attribute1 is referenced since the new value of attribute2 is dependent on the value of attribute1. That is, if attribute1 is equal to cons then the new value of attribute2 is assigned the value of attribute3 and if attribute1 doesn't equal cons then the value attribute2 is unchanged.

The attributes that are referenced conditionally by the writefile procedure are the Locked and Owner fields of the datafile type and the Currentprocess. In addition, the Value field of the datafile type is referenced directly.

After the direct and conditionally 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 3.1.2.

3.4 Other Phases of the Software Life Cycle

Although Section 3.2 dealt with only 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 (eg. the size of a file), more transforms, and transforms may have more parameters (eg. offset in a file or buffer 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 all that is necessary is to consult the matrix. For instance, if one wants to modify a variable it can immediately be determined 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 what 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 the introduction of possible storage and timing channels.

4. Advantages of The Methodology

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

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 (eg. English requirements and formal specification). That is, part of the system may be implemented while other parts have only English requirements or formal specifications describing them, but the methodology can be applied to the collection of descriptions.

5. Experience and Future Work

The shared resource matrix methodology has been successfully applied to the design of a secure network front end [12]. 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. As a result the front end was redesigned to block or reduce the bandwidth of these channels.

At present the matrix construction and analysis has mostly been performed manually. However, much of the work of generating the matrix could be automated. Currently the generation of the transitive closure of the matrix has been automated. This process is not dependent on the form of the system description; therefore, mechanizing the process did not restrict the versatility of the approach.

6. Conclusions

This paper has attempted to present a practical methodology for discovering storage and timing channels that can be used throughout the software life cycle. A complete example used to illustrate the use of the methodology on an English requirement, a formal specification, and implementation code can be found in [9].

It is hoped that the work presented here will help in the understanding of covert channels and more importantly how they may be enumerated.

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. Also Paul Eggert, Dino Mandrioli, Steve Bunch, and Gary Grossman who reviewed earlier drafts of the paper and provided helpful comments.

References

[1] B.W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, Vol. 16, pp. 613-615, Oct. 1973.

[2] B.J. Walker, R.A. Kemmerer, and G.J. Popek, "Specification and Verification of the UCLA Unix Security Kernel," *Communications of the ACM*, Vol. 23, pp. 118-131, Feb. 1980.

[3] S.B. Lipner, "A Comment on the Confinement Problem," *Proceedings of the Fifth Symposium on Operating Systems Principles*, The University of Texas at Austin, Nov. 1975.

[4] J.K. Millen, "Security Kernel Validation in Practice," *Communications of the ACM*, Vol. 19, pp. 243-250, May 1976.

[5] M. Schaefer, B. Gold, R. Linde, and J. Scheid, "Program Confinement in KVM/370," *Proceedings of the 1977 Annual Conference ACM*, Seattle, Washington, pp. 404-410, Oct 1977.

[6] C.S. Kline, "Data Security: Security, Protection, Confinement, Covert Channels, Validation," Ph.D. Dissertation, Computer Science Department, UCLA, Los Angeles, California, 1980.

[7] J.K. Millen, G.A. Huff, and M. Gasser, "Flow Table Generator," Mitre Working Paper, WP-22554, The Mitre Corporation, Bedford, Massachusetts, Nov. 1979.

[8] R.J. Feiertag, "A Technique for Proving Specifications are Multilevel Secure," CSL-109, SRI International, Menlo Park, California, Jan 1980.

[9] R.A. Kemmerer, "Shared Resource Matrix Methodology: A Practical Approach To Identifying Covert Channels," Report TRCS81-10, Computer Science Department, University of California, Santa Barbara, Nov. 1981.

[10] D.E. Bell and L.J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Vol I-III, The Mitre Corporation, Bedford, Massachusetts, Jun. 1974

[11] R. Locasso, J. Scheid, V. Schorre, and P. Eggert, "The Ina Jo Specification Language Reference Manual," SDC document TM-6889/000/01, System Development Corporation, Santa Monica, California, Nov. 1980.

[12] G.R. Grossman, "A Practical Executive for Secure Communications," DTI working paper, Digital Technology Incorporated, Champaign, Illinois, Sep 1981.

RESOURCE ATTRIBUTES \ PRIMITIVES		WRITE FILE	READ FILE	LOCK FILE	FILE LOCKED
FILE	SECURITY-LEVEL	R			
	LOCKED/UNLOCKED	R			
	OWNER	R			
	SIZE				
	VALUE	M			
DEVICE					

Figure 2.1

Example of a Shared Resource Matrix Filled-in for the WRITE_FILE Primitive

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
	OWNER	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		

Figure 3.1

Resource Matrix Filled-in from the English Description

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	
	OWNER	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

Figure 3.2

Transitive Closure of Matrix for English Description with Timing Example Added

```
$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 =
    A"s:Security_Class (
     s <: Sec_Classes(f) ->
       E"a.Access_Right (
          a <· Acc_Rights(p)
         & Class(a) = s
         & r <  Acc(a) ))

 VARIABLE
   Current_Process.Process,
   Owner(File):Process,
   Locked(File):Boolean,
   In_Use_Set(File):Processes,
   Value(File).Data,
   Buffer(Process):Data,
   Result(Process).Boolean

 TRANSFORM Write_File(f.File) External
  Effect
  A"f1:File (
    N"Value(f1)=
      (  f1 = f
      & Locked(f)
      & Owner(f) = Current_Process
        => Buffer(Current_Process)
        <> Value(f1)))

          .
END Top_Level
END Confinement
```

Figure 3.3

Ina Jo Specification of Example System

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	OWNER	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

Figure 3.4

Transitive Closure of Matrix for Ina Jo Specification

```
program toysystem (input,output);

const numproc = 30;
 numfiles = 5;
 filesize = 255;

type access = (read,write);
 securitylevel = (uncl,conf,secret,topsecret);
 procrange = 1..numproc;
 filerange = 1..numfiles;

 datafile =
  record
    owner: procrange;
    locked: boolean;
    inuseset: set of procrange;
    securitylevels. set of securitylevel;
    value: array[0..filesize] of integer
  end;

 accessright =
  record
    level: securitylevel;
    acc: access
  end;

 process =
  record
    accrights: set of accessright,
    buffer: array[0..filesize] of integer
  end;

var files: array[1..numfiles] of datafile;
 processes: array[1..numproc] of process;
 currentprocess: procrange;

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

procedure readfile (fileid:filerange);
     .
     .
begin
     .
     .
end.
```

Figure 3.5

Pascal-like Implementation Of Example System