

Auditing the Use of Covert Storage Channels in Secure Systems

Shiuh-Pyng W. Shieh and Virgil D. Gligor

Department of Electrical Engineering
University of Maryland
College Park, MD 20742

Abstract

In this paper we define requirements for auditing covert storage channels, and illustrate some fundamental problems which appear in most computer systems. We argue that audit subsystems designed to minimally satisfy the TCSEC requirements [1, 2] are unable to detect many instances of covert channel use, and hence require major design and implementation changes before they are able to detect all use of covert storage channels. Finally, we present the design of a Secure Xenix(*) tool for covert-channel audit that has been in operation since July 1989. Results of experiments indicate that the tool is able to detect all use of covert storage channels without raising false alarms.

1. Introduction

The idea of monitoring known security weaknesses of a computer system as a security counter-measure can be traced back to the Rand Corporation studies of computer security in the late 1960's and early 1970's [4]. Explicit requirements for monitoring weaknesses of otherwise secure systems have only been imposed more recently for evaluations of commercially available systems. In particular, the ability "to audit identified events that may be used in the exploitation of covert storage channels" has been an explicit requirement for computer systems in the security classes B2 - A1 of the TCSEC [1, 2]. However, in spite of this requirement and the significance of covert channels to system security and integrity, neither the notion of auditing storage channels has been defined precisely nor any tools for such auditing have been provided for any multi-level secure systems to date.

In this paper we define requirements for auditing covert storage channels, and illustrate several fundamental problems associated with such auditing in all computer systems.

(*) Xenix is a trademark of Microsoft Inc. Secure Xenix is a product of IBM Corp. Unix(R) is registered trademark of the AT&T Laboratories.

We argue that audit subsystems designed to minimally satisfy the TCSEC requirements [1, 2] are unable to detect many instances of covert-channel use, and hence require major design and implementation changes before they are able to detect all use of covert storage channels. Finally, we present solutions to the problems illustrated herein, and discuss their implementation in Secure Xenix(*) [3]. A storage-channel audit tool for Secure Xenix has been in operation since July 1989, and experiments with it show that it has been able to detect all use of storage channels without raising false alarms.

This paper is organized as follows. In Section 2, the requirements for auditing the use of covert storage channels are presented. In Section 3, some fundamental problems of auditing covert storage channels are discussed and illustrated. In Section 4, the important design and implementation issues that guide the development of an auditing tool for covert storage channels in Secure Xenix are presented. Sections 5 and 6 contain the conclusions and acknowledgments, and section 7 lists cited references.

2. Requirements of Covert-Channel Auditing

Covert-channel analysis identifies *individual* covert storage channels by triples $\langle \text{variable}, \text{PA}_i, \text{PV}_j \rangle$. The *variable* represents an internal system variable, or attribute, that can be modulated for illicit information transfer; $\text{PA}_i(\text{variable} \leftarrow 011)$, and $\text{PV}_j(\text{variable} \rightarrow 011)$, is a Trusted Computing Base (TCB) primitive that can alter, and view respectively, the *variable* [6, 8]. A covert-channel *type* is identified by the single variable or attribute that is modulated by various TCB primitives, and is represented by $\langle \text{variable}, \{\text{PA}_i\}, \{\text{PV}_j\} \rangle$. The distinction between an individual covert channel and a covert-channel type appears because a channel variable, or attribute, can be viewed and altered by more than one pair of PA_i, PV_j primitives.

The identification of individual covert channels and of channel types is made after covert-channel analysis is completed, and can be summarized as sets of $\langle \text{variable}, \text{PA}_i,$

PVj> triples as shown in Figure 1. The set of triples <variable, PAi, PVj> of Figure 1 represents only a subset of the Secure Xenix covert channels. This subset is used to illustrate the auditing problems discussed in this paper. Primitives **creat**, **link**, **open**, and so on, in the left-most column of Figure 1 are TCB primitives. Inode-table, inode-space, and so on, in the top row of Figure 1 are covert-channel variables. Note that upper-case letters "A" and "V" of Figure 1 are used to represent altering and viewing primitives of real covert channels whereas lower-case letters "a" and "v" are used to denote primitives which provide flows that cannot be used to create covert channels. For example, an individual inode-table channel can be represented by <inode-table, **creat**, **creat**>, and an inode-table channel type can be represented by <inode-table, {**creat**, **link**, **open**, **close**, **sdget**, **exec**, **unlink**}, {**creat**, **link**, **open**, **sdget**>.

Covert-channel auditing requires that sufficient data be recorded in audit trails to enable the *unambiguous* identification of: (1) individual covert-channel use; (2) multiple covert-channel use; and (3) transmitters and receivers of individual channels or of channel types (i.e., unambiguous identification of covert channel users). Furthermore, covert-channel auditing requires that discovery of covert-channel use must be *certain* (i.e., covert-channel auditing must not be circumventable), and that false detection of covert-channel use must be avoided. Circumvention of covert-channel auditing is undesirable because it allows the leakage of information to remain undetected. False detection of covert-channel use is also undesirable because it may make it impossible to distinguish between innocuous user activity and covert-channel use. Discovery of *actual leakage bandwidth* is possible and desirable once covert-channel use has been determined by audit-trail analysis. However, note that in general it is impossible to discover the actual information being leaked through covert channels by analyzing audit trails because a user can encrypt it before leakage. To an auditor, the information leaked through a covert channel is merely a meaningless sequence of events representing either zeros or ones. Also, it is impossible to distinguish between the leakage of useful information and noise merely by inspecting audit trails. However, note that whenever constant streams of events representing either zeros or ones are the only recorded patterns, these streams can be unambiguously classified as noise. Information transfer can only be achieved through the transmission of zero/one varieties.

TCB primitives	channel variable					
	Inode Table	Inode Space	File Inter-lock	File Table	No.Free Inodes	No.Free Blocks
creat	AV	AV	aV	AV	A	A
link	AV	av	a		a	a
open	AV	AV	aV	AV	A	A
close	A	a	a	A	a	a
sdget	AV	A	av		A	A
exec	A	a	A	A	a	a
unlink	A	A	aV		A	A
fork		a	a	aV	a	a
chsize		a	aV		a	A
waitsem			a			
ustat					V	V

Legend: A(V) = Altering (Viewing) primitive of a real covert channel
a (v) = Altering (Viewing) primitive flow that cannot cause a covert channel

Figure 1. Examples of Covert-Channel Analysis Results

3. Fundamental Problems of Auditing Covert Storage Channels

The problems of covert-channel auditing presented in this section are illustrated using variables and primitives of Unix(R) systems. However, these problems are fundamental in the sense that they are shared by most operating systems. These problems include: (1) determination of which covert-channel variable (type) is used, (2) separation of transmitters from receivers, (3) determination of whether a TCB primitive alters or views a channel variable, and (4) circumvention of covert-channel audit.

In general, these problems appear because a single TCB primitive may both alter and view a covert-channel variable, depending upon the argument values of that primitive and upon the system state; because a single TCB primitive may alter or view simultaneously (i.e., in one invocation) variables of different covert channels with or without error return; and because multiple TCB primitives may alter or view a single covert-channel variable. For example, the inode-table variable of Figure 1 can be both altered and viewed by each of the TCB primitives **creat**, **link**, **open**,

and `sdget` individually; and only altered but not viewed by `close`, `exec`, and `unlink` (**).

Simultaneous altering and viewing of channel variables can be achieved as follows:

- Simultaneous variable alteration – A primitive can alter multiple variables during a single TCB-primitive invocation depending on the flow of control within the primitive. For example, the `disk-inode-space`, `number-of-free-inodes`, `inode-table`, and `file-table` variables of Figure 1 can all be altered by a single invocation of `creat`;
- Simultaneous variable viewing – A primitive can view multiple variables during a single TCB-primitive invocation depending on the flow of control within the primitive. For example, whenever `creat` returns a file-table error it helps the invoker process view the inode table, disk-inode space, file interlock, and file table (with its error return) during a single invocation. Thus, regardless of whether a TCB-primitive invocation returns an error, that invocation can be used to view multiple variables.
- Simultaneous variable viewing and alteration – A primitive can alter a variable and view another variable during a single TCB-primitive invocation (with or without error returns). For example, `creat` alters the `disk-inode-space` variable (no error return) and views the `file-table` variable while returning an error uniquely associated with the `file-table`;

The set of variables altered by a primitive can be divided into several simultaneous-altering subsets in such a way that two variables belonging to two different subsets can be altered by the same primitive but only during different invocations; i.e., the primitive can alter either the first variable or the second variable in the same invocation. For example, `creat` can exclusively alter either the `disk-inode-space` variable or the `disk-block-space` variable in the same invocation. [This type of alteration is called the *exclusive alteration* in examples below]

(**) Note that the `link` primitive can only increment the `ip->i_nlink` (`disk-inode-space`) variable by one, if the count of the variable is greater than zero. The primitive `link` fails if `ip->i_nlink` variable is equal to zero. This type of alteration cannot allocate entries in the `disk-inode` space, because allocating an entry means that the variable is incremented from zero to one. Thus, `link` cannot be used as an altering primitive. Similarly, `link` cannot view whether inode space is exhausted and, therefore, it cannot be used as a viewing primitive. However, `close` can only decrement the `ip->i_count` (`inode-table`) variable by one, if it is greater than zero. In addition, `close` cannot increment `ip->i_count` variable. Thus `close` can be an altering primitive but not a viewing primitive of the `inode-table` channel.

3.1. An Example

The following example is used to illustrate the first three of the covert-channel audit problems mentioned at the beginning of this section.

Three processes P1, P2, and P3 (shown in Figure 3) run on a multi-level secure system with P1 running at the highest security level, P2 running at some security level between the highest and the lowest levels, and P3 running at the lowest level. Three variables, namely the `file-table` variable (`ft`), the `number-of-free-inodes` variable (`nfi`), and the `number-of-free-blocks` variable (`nfb`) are used for covert transmissions of information among the three processes shown of Figure 2. Each node of the graph represents a process, each edge represents the transmission of a single bit of information (0 or 1) and each set of rectangles represents an invocation of a primitive and its possible results. For example, the invocation of the primitive `ustat` by process P3, which is denoted by P3: PV <ustat; u.u_error=0> in Figure 2, helps process P3 view four transmissions in different system states.

The effects of primitives `creat` and `ustat` in the system state of Figure 2 can be summarized as follows:

- (1) P1: PA <creat; u.u_error = 0/23>
=> PA(ft ← 1) or PA(nfb ← 0/1)
- (2) P1: PA <close; u.u_error = 0>
=> PA(ft ← 0)
- (3) P2: PA/PV <creat; u.u_error = 23>
=> PV(ft → 1) or PA(nfi ← 0/1)
- (4) P2: PA/PV <creat; u.u_error = 0>
=> PV(ft → 0) or PA(nfi ← 0/1)
- (5) P3: PV <ustat; u.u_error = 0>
=> PV(nfb → 0/1) or PV(nfi → 0/1)

Statement (1) denotes the case when a process P1 invokes the primitive `creat` with `no-error/file-table-error(23)` return. Depending upon the system state, process P1 either modulates (alters or does not alter) `ft` and sends one bit of information '1', or modulates `nfb` and sends one bit of information '1' or '0'. Statement (3) denotes the case when a process P2 invokes the primitive `creat` and receives error 23, which is associated with `ft` uniquely. Process P2 either views `ft` and receives one bit of information '1' or modulates `nfi` and sends one bit of information '1' or '0'. The same convention applies to statements (2), (4) and (5).

Primitive `creat` can either create a new ordinary file or truncate an existing file depending on current system state. If the file exists, the length of the file is truncated to zero. As a result, the total number of free blocks is incremented, but the total number of free inodes remains unchanged. If the file does not exist, a new file is created. As a result, the

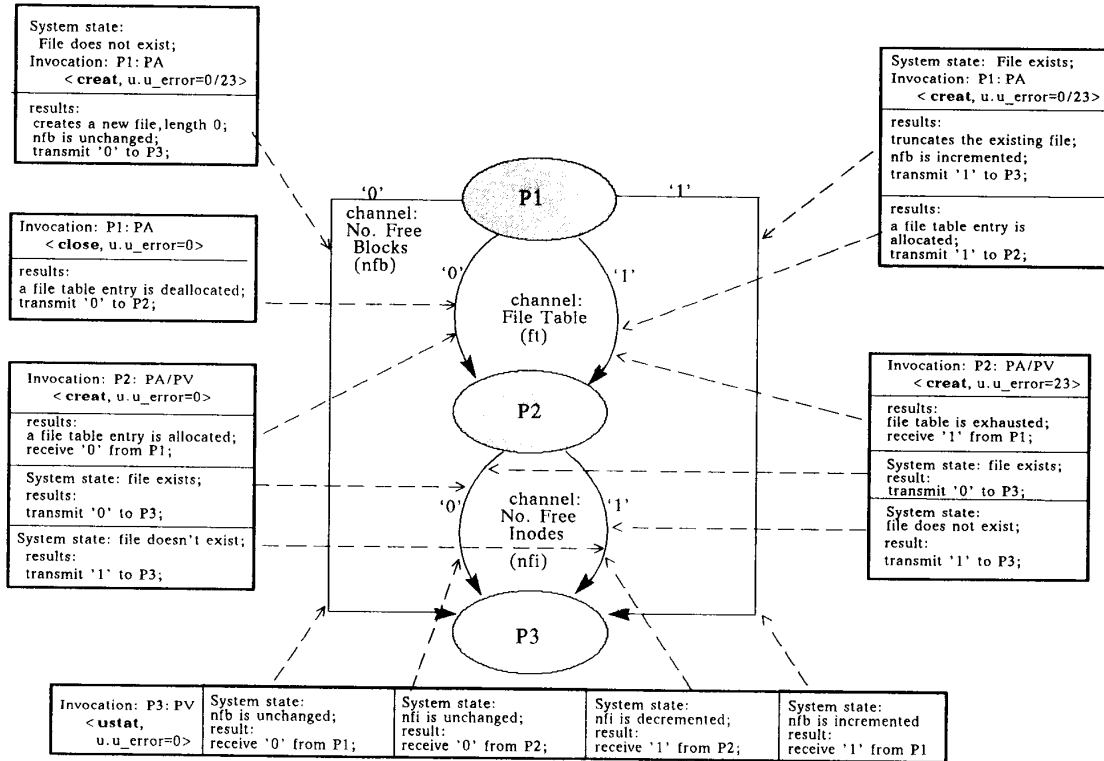


Figure 2. An Example of the Use of Multiple Channels by Three Processes.

total number of free inodes is decremented by one, but the total number of free blocks remains unchanged. In addition, if **creat** invocation returns no-error, a new file-table entry will be allocated. Otherwise, the file table remains unchanged.

3.2. Transmitter and Receiver Ambiguity

Ambiguity in identifying whether a process is a *transmitter* or a *receiver* arises from the simultaneous alteration and viewing of several covert-channel variables by a single primitive. For example, the following sequence of primitive executions could appear on an audit trail:

```

System state: file exists
:
P1: < creat, u.u_error=0 >
:
System state: file does not exist
:
P2: < creat, u.u_error=23 >
:
:
P3: < ustat, u.u_error=0 >
:

```

In this audit trail sequence, P1 truncates (**creat**) a file with no-error return because the file exists. Primitive **creat** can be the PA primitive of both the file-table channel and the number-of-free-blocks channel (see Figure 2). Process P1 can covertly transmit information to both P2 and P3 simultaneously through these two channels by invoking **creat** once. Thus, we have a *receiver ambiguity* (or an altering ambiguity) problem. In the meantime, process P2 can create a new file and can receive a file-table error return. However, **creat** can still alter the number of free inodes even if its invoking process receives a file-table error return. Therefore, **creat** can be both the PV primitive for the file-table channel and the PA primitive for the number-of-free-inodes channel. Thus, we also have a *transmitter/receiver ambiguity* (or an altering/viewing ambiguity) problem. At the same time, P3 can invoke **ustat** to check the file system statistics. Primitive **ustat** can be the PV primitive for both the number-of-free-blocks channel and the number-of-free-inodes channel. Thus, we have a *transmitter ambiguity* (or a viewing ambiguity) problem. These ambiguities cannot be resolved by using recommended audit mechanisms and audit-record formats [1, 2]

because these formats only include the TCB primitive identifier (action on an object) and its outcome, but not the actual covert-channel variable used.

3.3. Determination of the Covert-channel Variable Used

The simultaneous altering, viewing, and altering/viewing of multiple covert-channel variables by single primitive invocations (discussed in the example of Section 3.1 above) also make it impossible to identify unambiguously the covert channel being used. The audit record format recommended by [1, 2] only includes the TCB primitive identifier (action on an object) and its outcome, but not the identity of the altered or viewed variables.

3.4. Determination of Whether A TCB Primitive Alters or Views A Channel Variable

The use of TCB primitives for covert-channel transmission of information can be both system-state (environment) and call-argument dependent. Without knowledge of the system state or of the argument values passed to these primitives, the auditor cannot distinguish whether a TCB primitive, and which could potentially alter or view a channel variable, actually altered or viewed that variable. Thus, the auditor cannot determine whether a TCB primitive has actually altered (or viewed) a channel variable. The recording of all variables that are actually used (altered or viewed) by a primitive appears to be indispensable for covert-channel auditing. Thus, satisfying minimally the current TCSEC audit requirements and implementing the suggested audit-record format is insufficient for auditing covert-storage channels because the recording of channel variables is not required.

The covert-channel examples of Figure 2 illustrate both system-state and argument dependencies. For example, the primitive `creat` can alter (i.e., decrement) the total number of free inodes (`nfi`) only if the object to be created does not exist. If the object exists, `creat` has no effect on `nfi`. In addition, primitive `creat` can be used to alter (i.e., increment) the total number of free blocks (`nfb`) in the system if the file being created currently exists. That is, if the file exists, `creat` truncates the file, and as a result increments `nfb`. Otherwise, `creat` has no effect on `nfb`. (The disk-block-space channel is also affected by this condition.) Furthermore, the alteration of disk-block-space channel, and of the `nfi` and `nfb` channels by primitive `creat` is determined by the file system specified in the argument of the `creat` invocation.

The example of Figure 2 can also be used to illustrate combined state and argument dependencies. Consider again

the channel that modulates the `nfb` and the disk-block-space channel. Primitive `chsize` can be used to alter these channel variables (i.e., deallocate memory and increase the total number of free blocks) only if the file on which it is applied exists, and only if its argument indicates file shrinkage. When used to expand the size of an existing file, primitive `chsize` does not alter the channel variables but merely changes the `ip->i_size` field of the inode.

Other examples of argument dependency and combined state and argument dependencies, which are unrelated to those of Figure 2 can be found. For example, the primitive `semget(key, nsems, semflg)` can affect the semaphore-identifier channel and the semaphore-map exhaustion channel. Within this primitive, if `key` is equal to `IPC_CREAT`, a semaphore identifier, its associated semaphore data structure, and a set containing `nsems` semaphores are created for `key`. In contrast, if argument `key` is not equal to `IPC_CREAT`, nothing is created.

Furthermore, if argument `key` does not already have a semaphore identifier associated with it, and if the boolean expression (`semflg & IPC_CREAT`) is true, a `semget` call creates for argument `key` a semaphore identifier, its associated data structure, and the set containing `nsems` semaphores. If argument `key` already has a semaphore identifier associated with it, a new semaphore structure is not created.

3.5. Circumvention of Covert-Channel Audit

Covert-channel audit can be circumvented for at least two reasons: first, the use of covert channels can be disguised; second, the granularity of recorded time is insufficient to determine the ordering of event occurrences. These two problems are discussed below.

• Disguised Use of Covert Storage Channels

The discovery of covert-channel use can be difficult whenever a single TCB primitive views/alters multiple covert-channel variables. Because of lack of knowledge of control flow within a primitive, the actual use of a single variable cannot be discerned even when all variables that could be viewed are known and even when the `error` outcomes of a `PVj` primitive can be unambiguously associated with single channel variables. For example, a user can infer a `no-error` outcome of a `PVj` primitive on a given variable from an `error` outcome of the same `PVj` primitive on another variable. This enables users to transmit constant strings of either zeros or ones in order to disguise the use of a single channel as noise received on multiple channels.

To illustrate the notion of disguised use of covert channels, consider the flow of control within primitive `creat` shown in

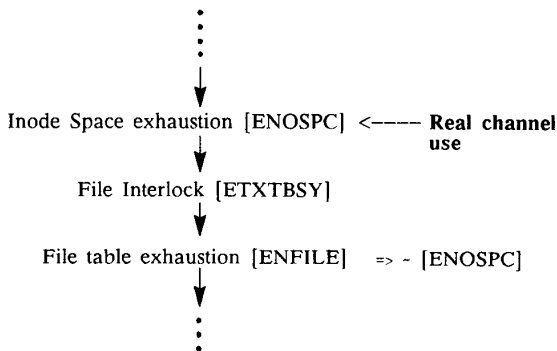


Figure 3. Fragment of Control Flow And Error Returns of the `creat` Primitive

Figure 3. This flow reveals the sequence in which different error values are reported to the user. By examining this sequence, a user can infer that certain error values have *not* been set and, therefore, that certain channel variables have *not* been modified as a result of the current primitive invocation. In this example, whenever exception `ENFILE` is reported by the file-table exhaustion channel, the user can infer that neither the `ETXTBSY` nor the `ENOSPC` error conditions prevail. Therefore, if this user wants to circumvent covert-channel audit whenever he uses the inode-space channel, he can use the primitives of the file-table channel for some, or all, of the no-`ENOSPC` values of the inode-space channel. This is shown in Figure 4. Note that the audit trail will contain records that represent constant strings of both the same inode-space and the same file-table events. These constant strings can be represented either by 1's (as in Figure 4) or 0's.

To enable auditors to discover such disguised uses of channels, the auditor tools would have to maintain tables of inferred no-error returns for each TCB primitive such as those for the primitives shown in Figure 1 above. These tables could be constructed after a complete analysis of the flow of control through TCB source code. Since the audit mechanisms recommended in [1, 2] do not include the use of such tables, tools cannot be built based on these mechanisms to detect the disguised use of covert channels.

• **Insufficient Granularity of Recorded Time**

The granularity of time recorded in audit records may be insufficient for covert-channel auditing in distributed systems. For example, in Secure Xenix (and in other systems) the granularity of the recorded time is that of a second. However, kernel calls may only use 1% of a second. In a centralized computer system, the granularity of recorded time may not be very important since all the occurrences of

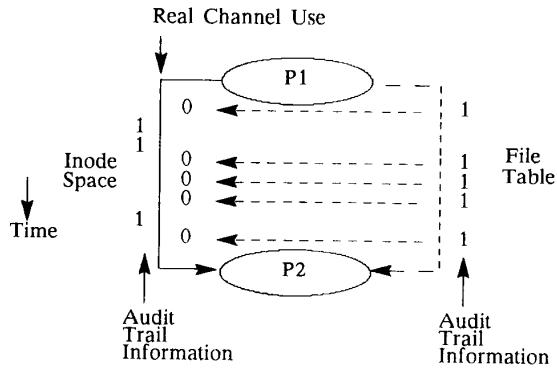


Figure 4. Disguised Use of the Inode-Space Channel (using both the inode-space and file-table channels)

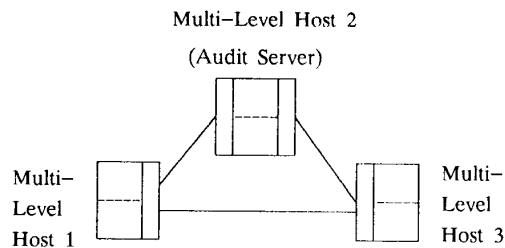


Figure 5. Identification of Single-Channel Use in Secure Distributed Systems

events are sequential. Thus, the occurrence of covert-channel events within a given interval can be determined even if multiple events appear to have happened at the same time.

However, time granularity is important for covert-channel auditing in distributed systems. For example, a connection between two hosts of a multi-level secure network may be established as shown in Figure 5. Suppose all the hosts use the TCP/IP protocols to communicate with each other. A transmitter process could exhaust all the ports on its host, and then could acquire or release a port to send one bit of information. A second process of the same host could request a port for a new connection. At the same time, a third process of another host could request a new port for a new connection with the host of the transmitter process. Therefore, the transmitter process could use the port-exhaustion channel to send information to either the process on its own host or the process on another host. In order to identify the transmitter and the receivers of this channel, the audit server would have to collect all the audit trails from every host of the network. However, without fine granularity of recorded time, the network audit server could not

determine the ordering of events. Thus, the audit server could not discover the use of this covert channel in the distributed-system environment. [The solution to this and other problems of covert-channel auditing in distributed system are beyond the scope of this paper and, therefore, will be addressed elsewhere.]

4. Design and Implementation of an Audit Tool for Covert Storage Channels

Key to the design of covert-channel auditing is the determination of what events should be recorded by the auditing mechanisms and what data should be maintained by the auditing tools to ensure that all covert-channel use can be discovered. The definition of individual channels and channel types would suggest that recording all events including pairs of <PA_i, variable> and <PV_j, variable> is necessary and sufficient for covert-channel auditing. However, the recording of such events is fraught with both practical and fundamental difficulties. Audit record formats and mechanisms currently used in practice include only process and user identifiers, object identifiers, process and object security levels, type of event (e.g., primitive identifier), and event outcome (i.e., success or failure); viz., [1, 2]. Fields for recording covert-channel variables are not included in existent audit-record formats; consequently, TCB code for audit-event collection is not specifically designed to capture the use of covert-channel variables. Thus, redesign of audit-collection mechanisms is necessary for covert-channel auditing.

4.1. Audit-Collection Mechanism

The audit-collection mechanism must resolve the fundamental problems of covert-channel auditing discussed above:

- (1) Determination of which covert channel variable (type) is used;
- (2) Separation of transmitters from receivers;
- (3) Determination of whether a TCB primitive alters or views a channel variable;
- (4) Circumvention of covert channel audit.

Problem (1) is resolved by having the TCB set a special error flag, [*u.u_aud_err*], which identifies uniquely a covert channel variable. This error flag can be used by the audit-analysis tool to identify the individual covert channels and covert-channel types. Problems (2), (3), and (4) are resolved by recording audit vectors which unambiguously determine the variables that are altered or viewed (and the corresponding user identifiers).

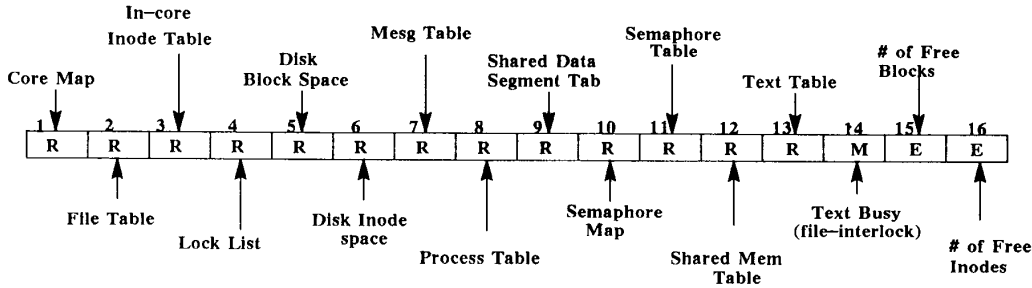
In resource-exhaustion channels it is sometimes possible to identify <PV_j, variable> pairs from recorded primitive

identifiers and event outcomes. For example, whenever the event outcome is an *error* that can be unambiguously associated with a channel variable, the auditor can infer that the recorded primitive identifier represents PV_j. However, whenever the event outcome is *no-error* and if the viewing primitive, PV_j, is also an altering primitive, PA_i, the auditor cannot tell whether the recorded primitive identifier is for a PV_j or a PA_i-type of primitive; nor can he tell whether the recorded primitive identifier represents a PA_i type of primitive or an innocuous TCB primitive whenever primitive PV_j differs from primitive PA_i. Thus, the recording of channel variables appears to be necessary for all *no-error* outcomes of primitives associated with covert channels.

Furthermore, we argued above in section 3.4 that variable alteration and viewing are system-state and argument-dependent operations. Therefore, the altering and viewing of a variable cannot be determined solely by the outcome of a primitive invocation. Also, since a primitive can both alter and view multiple channels simultaneously, all the covert-channel variables that are altered or viewed during execution have to be recorded in the audit trails. For these reasons, we introduce two types of vectors in the audit trails, namely, the *altering vector* and *viewing vector*.

Audit vectors include the altering vector and the viewing vector. In Secure Xenix, each vector is sixteen bytes long and each byte has a variable associated with it (see Fig. 6). Each vector monitors thirteen resource-exhaustion channel variables (represented by R), two event-count channel variables (represented by E) and one policy conflict channel variable (represented by M) [8, 9]. Only those variables that are system-state and argument dependent are included in the vector. Also recorded are process, user and group identifiers, outcomes, process and object security levels, arguments, pathname, device number and inode number. The device number is used to identify which file system's variable is altered or viewed. Both the device number and inode number are useful for detecting file-interlock (text busy) channels.

We also argued in section 3.2 that the unambiguous identification of transmitters and receivers is necessary. Knowledge of variable alteration by a primitive is required for transmitter identification. Although during the execution of a primitive, a variable may be incremented or decremented (i.e., allocated or deallocated), only the final result (i.e., difference between the number of elements allocated and the number of elements deallocated) after the exit of a system call really matters. Therefore, the altering vector must keep track of all the changes made to these covert-channel variables during the execution of a primitive. Whenever a primitive tries to increment a variable, whether successfully



R/M/E denote Resource-exhaustion, MAC policy conflict, and Event count channels [8].

Fig. 6. Byte Map of an Audit Vector

or not, the variable is viewed. However, a primitive that decrements a variable cannot view that variable. Therefore, the increment and decrement operation on a variable abide by the following rules:

- Initialization: $Alt_vector(X) = 0$;
- If m elements of resource X are allocated, then $Alt_vector(X) = Alt_vector(X) + m$;
- If n elements of resource X are deallocated, then $Alt_vector(X) = Alt_vector(X) - n$;
- Initialization: $View_vector(X) = 0$;
- If one element of resource X is allocated successfully/unsuccessfully, then $View_vector(X) = VIEWED$; where $VIEWED$ is a constant:

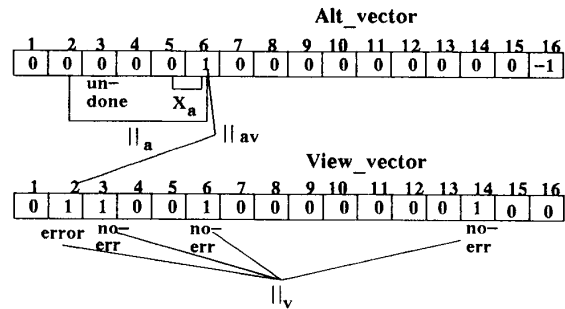
Figure 7 illustrates the contents of altering and viewing vectors. For primitive `creat`, the creation of a new file with file-table-exhaustion error return `u.u_aud_err=2` causes the following changes of the system state:

- alter and view disk-inode-space (variable 6);
- alter total-number-of-free-inodes (`nfi`, variable 16);
- view file-table (`ft`, variable 2) with associated error return;
- view in-core-inode-table (variable 3);
- view file-interlock (variable 14);
- the alteration of in-core-inode-table (variable 3) is undone because of the error return.

We also illustrate in Figure 7 the relations of simultaneous or exclusive altering and viewing that hold among the covert-channel variables of Secure Xenix. For example, variable 5 and 6 can only be exclusively altered while variable 2 and 6 can be simultaneously altered.

Additional information is needed for identification of covert-channel use, besides the identification of channel variables and their use. For example, if more than one file system is used in a system, the device (file system) number

`creat, new file,`
`u.u_aud_err=2 -> (file-table exhaustion)`



Legend: **Xa**: alter exclusively
||v: view simultaneously
undone: alteration is undone when getting error return

Figure 7. A Sample of an Altering Vector and a Viewing Vector for the Primitive `creat`

must be recorded in the audit trails to identify the disk-block-space channel, the disk-inode-space channel, the number-of-free-inodes channel, number-of-free-blocks channel, and so on. To identify the file-interlock channel, the inode number is also needed.

4.2. Audit-Analysis Tool

After sufficient data is collected by the audit-collection mechanism, the design of an audit-analysis tool becomes possible. Our audit-analysis tool reads the audit trail generated by the audit-collection mechanism and discovers covert-channel transmitters and receivers. The decisions made by the tool for identifying individual transmitters and receivers, handling composite events, finding patterns of transmitter-receiver association, preventing false alarms, and filtering noise, are presented below.

(a) *User Identity*

The identification of transmitters and receivers for covert-channel auditing should not rely on process identifiers. This is because a process could create and communicate with a large number of child processes in Unix(R)-like systems. Therefore, to allow for the legal flow of information between parent and child processes at the same security level, the audit tool must identify process families. The largest family of processes that can communicate legally, usually consists of processes with the same UID which run at the same security level. Therefore, the audit-analysis tool must identify transmitters and receivers with user, rather than process, identities.

(b) *Handling Composite Events*

Composite events, such as those of the remove-directory primitive `rmdir` in Unix(R), may consist of more than ten primitive calls. Also audit-trail analysis of these primitives individually may not be useful for detecting covert-channel use. For example, individual primitives used by `rmdir` cannot be used as viewing primitive, `PVi`, for detecting use of the upgraded-directory channel [9]. Although the trusted program `rmdir` as a whole can be used as a viewing primitive, `PVi`, the following problem arises. Every time `rmdir` is called, it may invoke different combinations of primitives depending upon the current system state. Hence, any audit-analysis tool needs to know each combination of composite events to produce correct results.

To keep track of composite events on a per-process basis, we introduced a *process profile* in the audit-analysis tool. A new process profile is created whenever a process is created and is removed when the process is destroyed. The process profile contains the user identifier, process security level, the device and inode numbers of the running program, and object security level. The user identifiers are used to discover transmitters and receivers. The device and inode numbers identify the running program. The subject and object security levels help determine whether the object is an upgraded directory. Thus, the process profile provides sufficient information to enable the tool to reconstruct a history of the process.

(c) *Patterns of Transmitter-Receiver Collusion*

An association (i.e., collusion) pattern for transmitters and receivers represents a particular type of composite event. This composite event consists of two primitives: the altering primitive and the viewing primitive. The task is to select associated altering and viewing primitive pairs from a large volume of audit records. To accomplish this task, the *viewer* and *alterer profiles* are created and maintained by the audit tool. A viewer profile is created only when a user

views a specific channel variable at a certain security level with an specific error return. The newly created viewer profile continues to record the user's activities associated with that channel variable. When a user process at a higher or incomparable security level alters the same channel variable as that of the viewer profile, an alterer profile associated with the channel variable is created because the existence of a viewer profile implies potential use of a covert channel. The alterer profile records the number of elements allocated or deallocated between two viewings of the channel variable. Once an alterer profile and viewer profile co-exist and channel-variable alteration occurs before channel-variable viewing, a combined *alter-viewer profile* is created which corresponds to each pair of alterer and viewer profiles. Thus, the alterer-viewer profile can track the activities of the alterer and viewer in a selective manner. Even though every user who alters (views) a covert-channel variable is a potential alterer (viewer), it is impractical to assume the use of a covert channel just because we can find an alterer and a viewer. Without the specific ordering of events, covert channels cannot be used. Event ordering and frequency are the two important factors which differentiate illegal from legal use of covert-channel primitives. Illegal use of covert-channel primitives require that certain patterns appears with high frequency and that the altering primitives must be followed by viewing primitives at a lower or incomparable security level. The audit-analysis tool is able to audit multiple users at different security levels using multiple covert-channel variables which can potentially leak information.

(d) *0/1 Variation, Frequency, and Audit-Trail Noise*

Resource-exhaustion channels and event-count channels can be noisy. Covert-channel noise can be caused by unconfined processes [5, 9] that run concurrently with transmitters and receivers within the system and alter or view each covert-channel variable. Unconfined processes are neither alterer nor viewer processes. The alterations and viewings of channel variables performed by unconfined processes can cause noise. As a result of this noise, many new alterer-viewer profiles could be created and could lead to false detection of covert channel use. To minimize false detection of covert channel use, the noise must be removed during audit-trail analysis.

High frequency of 0/1 variation between alterer and viewer processes suggests use of a covert channel. In contrast, an unconfined process may cause noise by invoking a primitive to alter or view the same channel variable and getting the same return (error/no-error). Thus, little or no 0/1 variation between alterer and viewer processes can classified as noise. We conclude that an alterer-viewer pair of

users invoking primitives with high frequency and 0/1 variation is involved in covert-channel use, whereas users invoking primitives with low 0/1 variation and frequency merely produce audit-trail noise (with respect to covert channel use) with high probability. The counting of the 0/1 variation reduces the false detection of covert-channel use.

4.3. Maximum Transmission Rate Between Two Users

Finding the maximum transmission rate between two users through a single covert-channel variable is inadequate for covert channel analysis, because each primitive may view more than one covert-channel variable at the same time. This means that a primitive may receive multiple bits through multiple channel variables. Furthermore, no-error returns can be inferred from error returns (viz., example of section 3.5).

As an example of simultaneous use of multiple covert channel variables, suppose that a single primitive can view channel variables $V1$, $V2$ and $V3$. Also suppose that the flow of control within this primitive is sequential; $V1$ is accessed before $V2$ and $V2$ is accessed before $V3$. The use of the three channels can be represented with a four-state graph, as shown in Figure 8. In this graph, nodes represent the states of the three variables; an edge indicates a state transition after an invocation of a viewing primitive, and the values associated each edge are coded message and elapsed time of this transition. Thus, the elapsed time T_{ij} equals to $2T_{cs} + T_{aij} + T_{vij} + T_{envij}$, where T_{cs} is the context switch time; T_{aij} is the elapse time of altering primitive; T_{vij} is the elapse time of a viewing primitive; T_{envij} is the environment setup time; i and j vary from 1 to 4. For example, during a transition from state (N, N, N) to state (N, F, X) of Figure 8, a receiver receives the coded message $01x$ with elapsed time $T_{1,4}$.

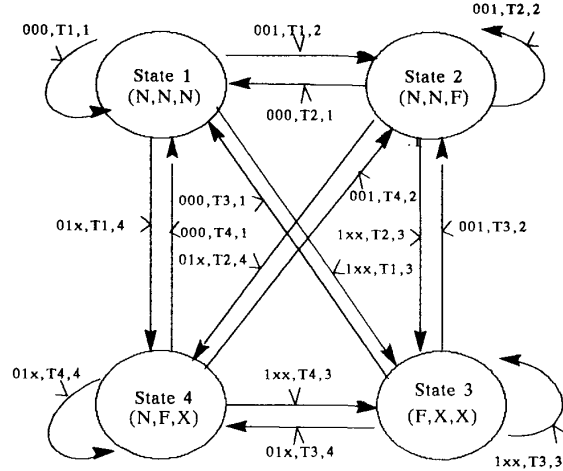
A method of finding maximum transmission rate of a n -state graph is described in [10] and an example of two-state graph is shown in reference [7]. The maximum transmission rate through a channel is defined as the channel capacity:

$$C = \lim_{t \rightarrow \infty} (\log_2 N_i(t)) / t \quad (\text{eq 1})$$

where $N_i(t)$ is the number of possible symbol sequences of total time t beginning at state i . In general, $N_i(t)$ obeys the difference equation :

$$N_i(t) = \sum_j N_j(t - T_{ij}), \quad (\text{eq 2})$$

where i and j vary from 1 to n and $T_{ij} = 2T_{cs} + T_{aij} + T_{vij} + T_{envij}$ is the time taken by a transition from state i to j . To determine the capacity of a channel, it is only necessary



Legend: (N,F,X) : $V1$ is non-full;
 $V2$ is full;
 $V3$ don't care;
 $01x, T_{4,4}$: $V1$ viewed with no-error return;
 $V2$ viewed with error return;
 $V3$ not viewed(don't care);

Fig. 8. A four-state graph for three combined covert channels using variables $V1$, $V2$, $V3$

to find the asymptotic upper limit of

$$N_i(t) = A_i x^t.$$

Substituting this solution in (eq 2) we obtain the system of equations:

$$A_i x^t = \sum_j A_j x^{t-T_{ij}}$$

This system of equations can be expressed in matrix form as $(P - I)A = 0$, where P is a matrix of negative powers of x . Since $P - I$ is singular, its determinant $\text{Det}(P - I) = 0$. Thus, (eq 1) implies:

$$C = \lim_{t \rightarrow \infty} (\log_2 A_i x^t) / t = \log_2 x$$

The channel capacity C is calculated from the largest root of the system of equations.

As the number of states n increases, the size of $n \times n$ matrix $(P - I)$ increases by the order of n^2 . Thus, solving the n simultaneous equation becomes a complex task. To find a simple solution, the following assumptions is made: $\forall i, j, i, j=1, \dots, n, T_{aij} = T_a, T_{vij} = T_v$ and $T_{envij} = T_{env}$. Let $T = 2T_{cs} + T_a + T_v + T_{env}$, thus $T_{ij} = 2T_{cs} + T_a + T_v + T_{env} = T$. Therefore, we have

$$\text{Det}(P - I) = \begin{vmatrix} x^{-T} & -I & x^{-T} & \dots & x^{-T} \\ x^{-T} & x^{-T} & -I & \dots & \vdots \\ x^{-T} & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x^{-T} & \dots & \dots & x^{-T} & -I \end{vmatrix}$$

$$= (-1)^{n-1} (n x^{-T} - I) = 0$$

$$\Rightarrow C = \log_2 x = \log_2 n / T$$

Because $n = N+1$, where N is the number of variables viewed by a primitive in which the flow of control is sequential, the maximum transmission rate of combined channels is:

$$C = \log_2(N+1) / (2Tcs+Ta+Tv+Tenv).$$

The transmission rate of the three combined channels of Figure 8 is twice that of a single channel. The original message 00, 01, 10 and 11 can be coded as the message 000, 001, 01x and 1xx.

Intuitively, We conclude that the maximum transmission rate is affected significantly by two factors. First, the time for altering and viewing a channel variable should be minimized. Second, the altering (viewing) primitive should alter (view) as many variables as possible. The actual transmission rate could also be discovered by our audit analysis tool, even if multiple channels are used, because the audit vectors record all the variables altered and viewed during the invocation of a primitive.

5. Conclusions

Fundamental problems of covert-channel auditing have been defined in this paper and illustrated with Secure Xenix examples. These problems appear in any multi-level secure system and illustrate the complexity of detecting user conclusion (e.g., information leakage) by audit-trail analysis. Research in this area shows that audit systems implemented to minimally satisfy current TCSEC audit standards and formats are unable to detect use of covert storage channels in many instances. Covert-channel auditing requires major changes to the design and implementation of the existent audit mechanisms.

6. Acknowledgments

This research was supported by IBM, Systems Integration Division under contract 481382B-WF at the University of Maryland, College Park. We are grateful to Janet Cugini of IBM for her review of earlier versions of this paper. We would also like to thank Gary Kennedy, Joe Seppy, Tom Tamburo, Matthew Hecht and Ed Gordy for their continued support and encouragement.

7. References

- [1] *Trusted Computer System Evaluation Criteria*, DoD STD-5200.28, December 1985.
- [2] *A Guide to Understanding Audit in Trusted Systems*, NCSC-TG-001 Version 2, June 1988.
- [3] Gligor, V.D., Chandrasekaran, C.S., Chapman, S., Dotterer, L., Hecht, M.S., Jiang, W.-D., "Design and Implementation of Secure XENIX," *IEEE Transactions on Software Engineering*, Vol. 13, No. 2, pp. 208-221, February 1987.
- [4] Hollingsworth, D., "Enhancing computer system security," Rand Corp. Paper P-5064, 1973.
- [5] Huskamp, J. C., "Covert Communication Channels in Timesharing Systems," Technical Report UCB-CS-78-02, Ph.D. Thesis, University of California, Berkeley, California, (1978).
- [6] Kemmerer, R. A., "Shared Resource Matrix Methodology: An approach to Identifying Storage and Timing Channels," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, August 1983, pp. 256-277.
- [7] Millen, J. K., "Finite-State Noiseless Covert Channels," Proc. of the IEEE Workshop on Computer Security Foundations, Franconia, NH., June 1989, 81-86.
- [8] Tsai, C. R., V. D. Gligor, and C. S. Chandrasekaran, "A Formal Method for the Identification of Covert Storage Channels in Source Code," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, (April 1987), pp. 74-86 (to appear in the *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990).
- [9] Tsai, C. R., Gligor, V. D., "A Bandwidth Computation Model for Covert Storage Channels and Its Applications," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, (April 1988), pp. 108-121.
- [10] Shannon, C. E., Weaver, W., *The Mathematical Theory of Communication*, The University of Illinois Press, Urbana, Illinois, 1964.