Formal Methods and Automated Tool for Timing-Channel Identification in TCB Source Code†

Jingsha He
VDG Inc.
6009 Brookside Drive
Chevy Chase, MD 20815
U.S.A.

Virgil D. Gligor
Department of Electrical Engineering
University of Maryland
College Park, MD 20742
U.S.A.

Abstract. We characterize the properties of timing channels that are reflected in source code and present formal methods for the identification of these channels in source code of trusted computing bases (TCBs). Our study differs significantly from previous ones which focus on a high-level characterization of timing channels without leading to practical methods for their identification [11, 16]. We also discuss how to integrate the formal methods presented into the automated system that has been previously developed for storage-channel identification [9] to build an automated tool for timing-channel identification in TCB source code which, otherwise, is still carried out in an ad-hoc way due to the lack of general and practical methods. The presented methods, however, cannot be directly applied for detecting hardware channels that result from hardware system configurations.

1 Introduction

Covert channels present a serious threat to secure computer systems and networks. These channels provide an illicit means of leaking sensitive information from cooperative users who have legal access to the information to unauthorized users. Because the leakage of information is usually carried out through variables and system states that are not part of the object representation in the security model, information flows through these variables are not protected under any multilevel mandatory security policy. For example, the variable that stores the total number of available disk blocks can be used to represent sensitive information: the sending process can encode a signal by exhausting the disk space and the receiving process can detect the signal by requesting an allocation

† Also available as IBM Technical Report 85.0148, June 1992.

The work presented herein has been performed at IBM Corporation, 800 N. Frederick Avenue, Gaithersburg, MD 20879, USA under a contract to VDG Inc. The views expressed in this paper only reflect those of the authors and do not imply any product offering from IBM.

Y. Deswarte et al. (eds.), *Computer Security - ESORICS 92*

of some disk space. Information transfer is realized when the receiving process is returned an error message for its request or observes a different response time from its execution due to the unavailability of the disk space.

There are two types of covert channels: *storage channels* use shared global variables/data structures as the media for information transmission, whereas *timing channels* take advantage of different response times observed by the receiving process under the manipulation of the sending process [11, 13]. Both types of channels present an equal threat to secure systems and are required in [15] to be formally analyzed for a system to be evaluated at the B3 level and above. However, past work in this area has mostly centered on identifying storage channels, which has led to the development of several formal methods and automated tools [3, 5, 7, 9, 11, 12, 14, 16], and very little has been done in timing-channel analysis. The only relevant technique that can be immediately applied to timing-channel analysis is the interference approach [4, 7]. But its drawbacks are equally obvious [7]. Another method, the Shared Resource Matrix (SRM) Methodology [11], is only useful for deriving transitive closures of timing channels between processes but offers little help in identifying the primitive channels in the first place. To this date, timing-channel identification is still mostly conducted in an ad-hoc way due to the lack of general and practical methods.

Hu introduced the notion of "fuzzy time" as the basis for the development of a collection of mechanisms to block the use of timing channels [10]. While fuzzy time can reduce the effective bandwidth of timing channels by randomizing the response time of system events to users, it also destroys other necessary system synchronization mechanisms for legitimate user communication and for real-time control. Therefore, the use of fuzzy time in dealing with timing channels is limited to military and commercial applications where the use of timing channels is the only primary concern. To this date, identification and analysis still remains the only practical way of meeting the relevant requirements in [15] concerning the use of covert channels.

In this paper, we study the properties of timing channels and propose formal methods for systematic identification of these channels in TCB source code. We also discuss how to integrate these formal methods into the automated system that has been previously developed for storage-channel identification [9]. These formal methods cannot be directly used for detecting hardware timing channels, however. Nevertheless, it is still necessary and important to devise effective ways of identifying covert channels in TCB source code (i.e., software channels) although hardware channels usually have a higher bandwidth. This is because hardware channels are generally of a dynamic nature, i.e., they may not exist in every system that runs the same TCB but will also depend on specific hardware system configurations, whereas software channels are always of a static nature, i.e., they exist in every system that runs the same TCB. In addition, past experience indicates that most software channels have a substantially higher bandwidth than that specified in [15] for security evaluation and certifi-

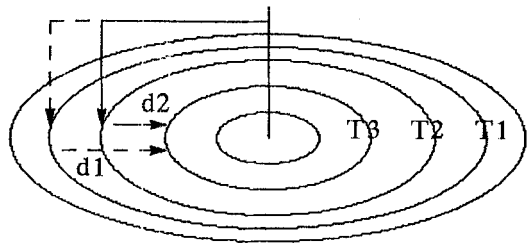
cation [9, 16] and the difference would only become greater with the advent of faster processors and architectural improvement.

This paper is organized as follows. In the next two sections, we characterize timing channels and present formal methods for identifying these channels based on the characterization. In Section 4, we describe the automated system that has been previously developed for storage–channel identification [9] and discuss the integration of the formal methods into the system to build an automated tool for timing–channel identification. In Section 5, we discuss hardware timing channels that cannot be determined through source–code analysis because of their dependence on hardware configurations and thus conclude that, unlike storage channels, source–code analysis alone may not reveal all the potential timing channels. Finally, we conclude this paper in Section 6.

2 Characterization of Timing Channels

The general definition of covert channels does not distinguish between the two types: *storage* and *timing*, but is presented in terms of system behaviors with respect to the multilevel mandatory policy in the security model and its interpretation in real implementations. That is, given a security model M and its interpretation I(M) in a system, a potential communication between two subjects (processes) in I(M) is covert if and only if such communication between the two corresponding subjects (processes) in M is illegal [16]. This definition also implies that covert channels that exist in source code are only potential channels and only a subset of these potential channels are exploitable as real channels. Thus, real–time scenarios must exist for a potential channel to be a real channel [9, 11, 16]. The distinction between a storage channel and a timing channel is made at this time when real–time scenarios to use it is constructed. If a scenario involves the altering and viewing of a shared global variable by the two communicating processes, the channel is a storage channel. If a scenario involves having the sending process modulate its use of some system resources in such a way that the response time of the receiving process can be affected, the channel is a timing channel [11, 16]. Therefore, it is possible that a covert channel can be both a storage channel and a timing channel.

While the scenario for the use of a storage channel indicates a property that can be directly used for the identification, that for the use of a timing channel provides little insight of how to characterize the static nature of timing channels and of how to reach a general solution for the identification. This is mainly because the words "altering" and "viewing" in characterizing storage channels can immediately find their semantics in source code but the word "modulate" in characterizing timing channels does not have any simple semantics to directly map into. Timing channels should be characterized in such a meaningful way that a formal method for the identification can be easily obtained. Therefore, the key to a satisfactory solution to this problem relies upon the proper interpretation of the word "modulate;" this along with the characterization of timing channels are offered below.



Sending process
1. Modulate the use: position disk head at T1 or T2;
2. Send a one: position disk head at T1;
3. Send a zero: position disk head at T2.

Receiving process
1. Receive data: position disk head at T3;
2. Receive a one: need a longer response time d1;
3. Receive a zero: need a shorter response time d2.

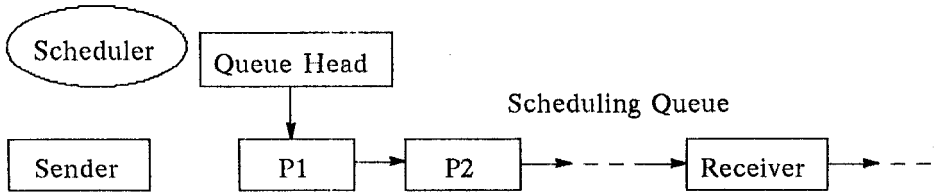
Fig. 2.1. A direct timing channel using the hard disk

2.1 Timing Channel Types

Timing channels can be characterized in various ways. In this paper, we characterize timing channels based on the number of processes that have to be involved in the communication through the channels. In general, timing channels can be classified into two types: *direct* and *indirect*. In a direct timing channel, the sending and the receiving processes are the only two processes that are needed to make use of the channel: the sending process modulates its use of a system resource so that the response time of the receiving process can be affected without the involvement of any other user processes in the system. An example of a direct timing channel is depicted in Fig. 2.1: the sending and the receiving processes share the use of a hard disk and the sending process modulates the use of the hard disk by positioning the disk head in different cylinders so that the response time of the receiving process is affected.

In contrast to a direct timing channel, the use of an indirect timing channel has to involve other processes in the system for the receiving process to observe different response times. An example of an indirect timing channel is depicted in Fig. 2.2: the sending process is able to affecting the execution of processes that are to be scheduled before the receiving process. Depending upon the execution of these processes, the response time of the receiving process can be different. The sending process can achieve this goal by exhausting the system resources that the processes in the scheduling queue (i.e., P1, P2, etc.) have to acquire before they start so that their execution has to be suspended due to the lack of necessary resources. This channel exists in TCB systems that use a simple FIFO scheduling scheme and becomes useless if the scheduler randomly selects the next process to occupy the CPU.

2.2 Minimum Requirements for Timing Channels



Sending process
1. Modulate the use: change response times of P1, P2, ...;
2. Send a one: make P1 or P2 or ... execute longer;
3. Send a zero: make P1 or P2 or ... execute shorter.

Receiving process
1. Receive data: execute on CPU;
2. Receive a one: a longer delay before execution;
3. Receive a zero: a shorter delay before execution.

Fig. 2.2. An indirect timing channel caused by process execution

Kemmerer gives a set of "minimum" conditions for the existence of a timing channel [11], which can be summarized as follows:

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

Same as in the case of a storage channel, condition (d) requires that a real–time scenario exist for a potential timing channel and conditions (a), (b) and (c) describe the inherent properties of the channel. Condition (c) is not necessary for a timing channel, however. A counter–example is the indirect timing channel in Fig. 2.2. In general, in an indirect timing channel, it is not necessary for the receiving process to be able to detect a change made to a shared resource by the sending process. It would be sufficient that the response time of the receiving process be affected in response to the change to the shared resource. Therefore, condition (c) cannot be a minimum condition for all types of timing channels, in particular not for indirect timing channels.

We now formulate a set of requirements as the minimum conditions for the existence of a timing channel. Here we like to emphasize two points. First, the sending process must be able to alter the system state in such a way that the response time of the receiving process can be affected. The part of the system state that is represented in source code consists of all the resources that can be shared by processes. Thus, it includes all global variables (those that do not belong to any process and retain their values from one process to another) and all user interface variables (those whose values can be set by the user when invoking a process). It is not required, however, that the receiving process share

access to the same resource as the sending process. That is, condition (a) is too excessive.

Second, the receiving process must be able to access a time reference. It is not necessary, however, that the sending process also have access to a time reference. This is because only the receiving process needs to have the knowledge of the response time to decode information. The way in which the sending process modulates the use of a shared resource does not depend on any factors related to time but only on what signal (i.e., one or zero) it intends to send to the receiving process. That is, condition (b) is overstated as a minimum requirement. The time reference can be a real-time clock in the system that is accessed by the receiving process or a real-time clock out of the system that the invoking user of the receiving process can read upon return from the execution.

We list the following four minimum conditions for the existence of a timing channel:

(a') The sending process must be able to alter the system state.
(b') The receiving process must have access to a time reference.
(c') The alteration of the system state by the sending process can affect the response time of the execution of the receiving process.
(d') There must be a synchronization mechanism for initiating the sending and the receiving processes and for ordering system events.

Since the above conditions are minimum, the absence of any would make it impossible for the sending and the receiving processes to transmit information. If condition (a') is not true, the sending process would have no way of affecting the execution of the receiving process. If condition (b') is not true, the receiving process would have no way of distinguishing between different response times or determining the order of the arrival of multiple system events [17] to decode the signals from the sending process. If condition (c') is not true, the information encoded by the sending process cannot be decoded by the receiving process. Finally, condition (d') determines whether a potential timing channel is a real channel. If there is no way of using a potential channel in real time, the channel cannot be a real channel, the same requirement as that for a real storage channel [9, 16]. Note that, when a static timing-channel identification is conducted in source code based on the above criteria, only conditions (a') and (c') are used to derive the solutions because condition (b') trivially holds in all systems nowadays and condition (d') can only be applied manually to determine the set of real channels out of the potential channels discovered statically. Note also that condition (d') may require the sending process also to access a time reference for synchronization purpose, which depends on specific synchronization mechanisms but is not an inherent property of timing channels, and thus is not considered as a minimum condition in this paper.

The two example timing channels in Fig. 2.1 and Fig. 2.2 satisfy all the above conditions. In particular, the sending process alters the system state by positioning the disk head at different cylinders in Fig. 2.1 and by affecting the execution of the processes in the scheduling queue in Fig. 2.2. The alterations to the

system state in both cases cause the receiving process to execute with different response times.

3 Identification of Potential Timing Channels

Timing-channel identification in source code can only reveal the set of potential channels [9, 11, 16][1]. The similarity of the characteristics of a timing channel to those of a storage channel indicates that methods and theories that have been developed for storage-channel analysis are applicable to timing-channel analysis as well. Therefore, the identification of timing channels should be conducted in TCB source code where all the potential software channels in final system implementations reside [9, 16]. We can model TCB source code as a tuple <V, E> where V is a set of variables and E is an ordered list of rules describing information flows among the variables in V. Shared system resources in source code are also expressed in the form of global variables/data structures that can be referenced in more than one procedure. Consequently, the part of the system state that is defined in terms of system resources and user interface variables can be uniformly defined in terms of variables only.

Definition 3.1. (System State) The system state in source code consists of (1) all the global variables that are shared by more than one process and (2) all the user interface variables whose values can be set by the user when invoking a process. Variables of the above types are called state variables. □

3.1 Direct Timing Channels

Definition 3.2. (Direct Timing Channels) A direct timing channel is a timing channel that satisfies the following two conditions:

(1) The channel consists of two processes P_s and P_r where P_s is the sending process and P_r the receiving process.

(2) The execution of P_s can affect the response time of the execution of P_r .□

Let us now identify all the possible behaviors of two arbitrary processes that can be exploited in a direct timing channel for information transmission. Let us assume that, for any two sequences of statements S_1 and S_2, the amount of time for executing S_1 is always different from that for executing S_2 unless S_1 and S_2 are semantically the same. Therefore, the execution of the different branches of an alternation statement always takes a different amount of time if the two statement sequences in it are not semantically the same. That is, the execution of the statement:

$$\text{if (C) then } S_1 \text{ else } S_2$$

takes a different amount of time when $C \neq 0$ from that when C=0. Similarly, the amount of time for executing an iteration statement:

---

[1] From now on in this paper, all discussions on timing-channel identification and analysis in source code refer to the identification of potential timing channels.

while (C) do S

will depend on the value of C. Consequently, all variables in the condition expression C of an alternation or iteration statement, whether global or local, can affect the execution time of the statement.

The determination of semantic equivalence of S_1 and S_2 is not straightforward, however. Strictly speaking, S_1 and S_2 are semantically equivalent if and only if they always take the same amount of time to execute. Thus, S_1:{x=1;y=0;} and S_2:{y=10;x=0;} are usually considered semantically equivalent. However, the above notion cannot be simply extended to statements of the same type, e.g., assignment statements are semantically equivalent. For example, S_1:{x=y;} may need a different amount of time to execute from S_2:{x=0;} depending on the amount of time to fetch y. In general, deciding the semantic equivalence of two arbitrary statement sequences is equivalent to the halting problem and is thus unsolvable in any static analysis. Consequently, we take the common practice of conservatism in security and assume that S_1 and S_2 in an alternation statement is always semantically different. The problem with this conservative assumption is the possibility of generating false flows, that is, those that do not exist when S_1 and S_2 are indeed semantically the same. This problem is unavoidable but these false flows will be singled out at the time of constructing real-time scenarios for the corresponding potential channels because there are none for them.

There are two ways in which the response time of the receiving process can be affected by the sending process. First, the sending process can set (or transform) the system state in such a way that the execution of the receiving process is affected directly by the system state. Second, the sending process can manipulate its own execution to control the time at which the receiving process starts to execute. In other words, the sending process controls *how* the receiving process executes in the first way and *when* the receiving process executes in the second way. We call the former a *tightly coupled timing channel* and the latter a *loosely coupled timing channel.*

3.1.1 Tightly Coupled Timing Channels

For a tightly coupled timing channel:

(1) the sending process is able to alter a global variable during its execution,
(2) the receiving process has the value of the same global variable appear in the condition expression of an alternation or iteration statement,
(3) the sending process sets the global variable to a value corresponding to the value of the signal (i.e., one or zero) it intends to send to the receiving process, and
(4) the receiving process detects the signal by observing the response time of its execution.

A tightly coupled timing channel is illustrated with the two example procedures in Fig. 3.1. To transmit one bit of information using this channel, the sending process sets the value of the signal it intends to send to the receiving process.

Sending process

```
t_sender(signal)
int signal;
{
    extern int gv;
    gv = signal;
}
```

Global variable

```
extern int gv;
```

Receiving process

```
t_receiver()
{
    extern int gv;
    int lv = gv;
    if (lv)
        delay(2);
    else delay(1);
}
```

Fig. 3.1. A tightly coupled timing channel

The receiving process observes the response time of its execution and detects a "one" when the response time corresponds to a two-unit-time delay and a "zero" when the response time corresponds to a one-unit-time delay. To transmit a string of "zeros" and "ones" from the sending process to the receiving process, the two procedures have to be executed repeatedly with one bit per invocation. For example, if the sending process wants to send "1101011101" to the receiving process, it invokes procedure *t_sender* ten times with the value of the signal being properly set on each invocation. The receiving process also invokes procedure *t_receiver* ten times to detect the response times of "2212122212" from its executions. Fig. 3.2 graphically shows the process of transmitting this string of bits from the sending process to the receiving process. Here we assume that the execution of the sending process always takes a one-unit-time delay and that process switching time is negligible.

The identification of a potential tightly coupled direct timing channel in any formal method can be described by the following algorithm:

Algorithm 3.1. (Identification of a Tightly Coupled Timing Channel)

(1) Select a global variable V;
(2) Determine if V can be altered in the program of a process (sending process);
(3) Determine if the value of V is used in the condition expression of an alternation or iteration statement in the program of another process (receiving process);

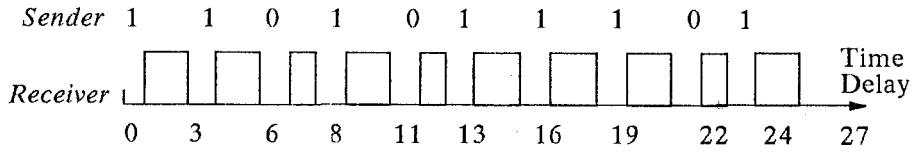


Fig. 3.2. Information transfer using a tightly couple timing channel

End.

It is easy to see that the example in Fig. 2.1 is a tightly coupled timing channel. It is also easy to see that a resource–exhaustion channel which is viewed as a classical storage channel [9, 16] is also a tightly coupled timing channel since different response times can occur during the execution of the receiving process depending on whether the shared resource is still available. Note that one way of handling a resource–exhaustion storage channel is to prevent the error message from being returned to the receiving process after the resource has been exhausted, which would eliminate the corresponding storage channel. However, the fact that the same channel is also a (tightly coupled) timing channel makes it possible to still leak sensitive information through exhausting the same resource although it is no longer usable as a storage channel.

3.1.2 Loosely Coupled Timing Channels

For a loosely coupled timing channel:

(1) the sending process is able to control the response time of its own execution, which requires that the value of a user interface variable appear in the condition expression of an alternation or iteration statement,
(2) the sending process can set the user interface variable in order to manipulate the response time of its execution, and
(3) the receiving process detects the signal from the sending process by observing the time it starts execution or the time it stops execution and responds.

A loosely coupled timing channel is illustrated with the two example procedures in Fig. 3.3. To transmit one bit of information using this channel, the sending process sets the value of the interface variable to control the amount of time it will spend on execution. The receiving process observes the time when it starts to execute and detects a "one" if it starts late and a "zero" if it starts early. Therefore, to send a string of "zeros" and "ones" from the sending process to the receiving process, the two procedures have to be repeatedly executed with one bit per invocation. For example, if the sending process wants to send "1101011101" to the receiving process, it invokes procedure *l_sender* ten times with the value of the signal being properly set on each invocation. The receiving

Sending process

```
l_sender(signal)
int signal;
{
   int lv = signal;
   if (lv)
      delay(2);
   else delay(1);
}
```

Receiving process

```
l_receiver()
{
   delay(1);
}
```

Fig. 3.3. A loosely coupled timing channel

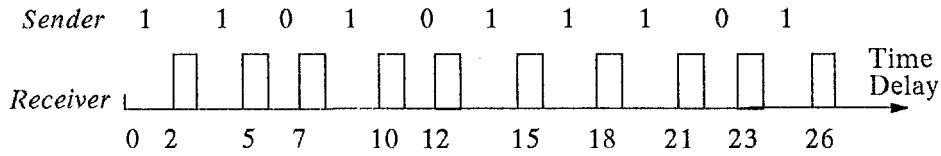


Fig. 3.4. Information tranfer using a loosely coupled timing channel

process also invokes procedure *l_receiver* ten times to detect its relative starting times of "2212122212". Fig. 3.4 graphically shows the process of transmitting this string of bits from the sending process to the receiving process. Here we assume that the execution of the receiving process always takes a one-unit-time delay and that process switching time is negligible.

It is clear that, in a loosely coupled direct timing channel, the sending process does not need to share any data with the receiving process. Moreover, the sending process does not have to be paired with any specific receiving process. Any process can be the receiving process to detect signals from the sending process. Therefore, it is sufficient to identify only the sending process in this type of channels that can potentially send or broadcast information to any other processes that wish to receive the information. The algorithm for detecting a timing channel of this type can be described as follows:

Algorithm 3.2. (Identification of a Loosely Coupled Timing Channel)

(1) Select a user interface variable V;
(2) Determine if the value of V is used in the condition expression of an alternation or iteration statement in the program of a process (sending process);
(3) Do nothing to any other process (receiving process);

End.

3.2 Indirect Timing Channels

Definition 3.3. (Indirect Timing Channels) An indirect timing channel is a timing channel that satisfies the following two conditions:

(1) The channel consists of n+2 processes P_s, P_1, P_2, ..., P_n, P_r, $n \geq 1$, in which (a) P_s and P_r are the sending and the receiving processes, respectively; (b) P_1, P_2, ..., P_n are other user processes in the system; and (c) the execution of the processes observes the same order as that in the above representation.
(2) The execution of the sending process P_s can affect the response time of one or more of the other n processes P_1, P_2, ..., P_n. □

Note that the environment for the use of an indirect timing channel is different from that for the use of a direct timing channel. It is generally believed that the introduction of other processes into a system can make a timing channel virtually useless due to the incurred noise level. We can see that this way of resolving

conventional timing channels, however, has the potential of introducing indirect timing channels. This is because, from Definition 3.3 (2), (a) P_S and a P_i, $1 \leq i \leq n$, i.e., the process whose response time can be directly affected by P_S, have a tightly coupled direct timing channel in which P_S and P_i are the sending and the receiving processes and (b) P_r can be thought of as one of the processes that is added to the system on purpose for generating noise to the direct timing channel between P_S and P_i. We thus have a simple rule for detecting an indirect timing channel:

Rule 3.1. (Existence of Indirect Timing Channels) For each tightly coupled direct timing channel, there always exists an indirect timing channel between the sending process of the direct timing channel and any process except the receiving process of the direct timing channel. □

The above rule indicates that the identification of indirect timing channels does not require additional effort beyond that of direct timing channels. Indirect timing channels can be discovered as a by–product of direct timing–channel identification.

3.3 Application in TCB Source Code

The analysis of TCB source code (along with hardware examination) is an integral part of covert–channel analysis. Since user processes have to invoke TCB primitives to gain access to system resources, an analysis of the entire TCB source code is mitigated to a static analysis of individual primitives. The determination of potential covert channels between processes, therefore, becomes a job of identifying potential covert channels between primitives. Consequently, one must consider two primitives and one shared global variable for tightly coupled direct timing channels and one primitive and one user interface variable for loosely coupled direct timing channels in the analysis.

The procedure of identifying a potential tightly coupled direct timing channel between primitives can be described as follows:

Procedure 3.1. (A Tightly Coupled Timing Channel)

(1) Select a global variable that is shared by primitives;
(2) Select a pair of primitives for analysis;
(3) Invoke Algorithm 3.1 with the shared global variable and with the two primitives as the sending and the receiving processes, respectively;

End.

The output of the identification can be represented in the form of a set of matrices with each one for the potential channels through one global variable in which primitives are labeled under the variable as being able to send or to receive information through the channel. An example matrix for global variable *file–>f_count* in Secure XENIX[2] (which represents the availability of an entry in

***** file->f_count *****		------ tightly coupled ------
Sender	creat	/* scenarios of use */
	creatsem	.
	fork	.
	open	.
	opensem	.
Receiver	creat	.
	creatsem	.
	dup	.
	exec	.
	fork	.
	open	
	opensem	/* scenarios of use */

Fig. 3.5. A tightly coupled direct timing channel in Secure XENIX

the system-wide file table) is shown in Fig. 3.5.[3] To determine the set of real channels, real-time scenarios of use must be constructed. For example, an examination of the use of the file table by primitive *fork* reveals that the way in which *fork* alters the variable *file->f_count* is not consistent with the scenario for sending a signal. Thus, primitive *fork* is only a potential sender (determined statically) but cannot be a real sender for this channel (determined in real time).

A similar procedure can be formulated for the identification of a potential loosely coupled direct timing channel between primitives:

Procedure 3.2. (A Loosely Coupled Timing Channel)

(1) Select a primitive for analysis;
(2) Select a user interface variable whose initial value can be set by the user upon the invocation of the selected primitive;
(3) Invoke Algorithm 3.2 with the user interface variable and with this primitive as the sending process;

End.

Each output matrix, however, only contains the list of potential senders because all primitives are potential receivers and need not be explicitly shown.

4 Automated Tool for Timing-Channel Identification

We have developed an automated system for the identification of potential covert storage channels in TCB source code and have successfully applied the sys-

---

[2] XENIX is a registered trademark of Microsoft, Inc.
Secure XENIX was developed by IBM Federal Systems Company. It is now marketed as Trusted XENIX by Trusted Information Systems, Inc.

[3] The matrix may not contain all primitives that can send and/or receive information through the timing channel.

tem to Secure XENIX [6, 9]. Our past experience shows that this system is able to identify all potential storage channels in TCB source code. We discuss in this section how to integrate the timing–channel identification algorithms presented in this paper into this system to make it able to identify both types of potential channels in TCB source code. The integrated system cannot detect timing channels resulting from hardware configurations, however, which we discuss in the next section.

4.1 The Automated Storage–Channel Identification System

The automated system takes the C language of a TCB implementation and generates a list of illegal information flows that can lead to covert storage channels in the TCB. Each such illegal flow is represented by a global variable, the identities of the two primitives that can alter and view the global variable, and the altering and the viewing flow paths and flow conditions that show how and under what condition the global variable is altered and viewed. If a real–time scenario exists for this flow to take place, it indicates a real covert channel in the TCB and must be handled properly.

This system consists of three components: the *Flow Generator* (FG), the *Flow Integrator* (FI) and the *Flow Analyzer* (FA), whose structure and the relationships among the components are depicted in Fig. 4.1. The function of the FG is to take C functions of TCB source code and translate each individual statement into a set of flow relations represented in a universal format. These flow relations are handed over to the FI for flow integration to derive global flows at the TCB user interface. These global flows are then analyzed in the FA to determine their legality with respect to the multilevel mandatory security policy implemented in the TCB. The final output of the entire system is the set of illegal flows that lead to potential covert storage channels. Finally, real–time scenarios are constructed for these potential channels to determine the set of real covert storage channels. The last step has to be carried out manually because the determination of real flows from a static analysis is equivalent to the halting problem [9] and is thus unsolvable.

This system has the following two distinctive features. First, the separation of flow generation from integration makes the FI and the FA general–purpose tools independent of any programming language. To use the tools for covert channel analysis in TCBs of a language other than C, one only needs to build a separate FG module that parses the language and supplies the set of flow relations. Second, the separation of flow integration from analysis makes the FI a general–purpose tool not solely for covert–channel identification. This is important because information–flow technique is equally applicable to analyzing other system properties, e.g. penetration and integrity. In addition to be able to identify all potential covert storage channels, this system does not generate false illegal flows that indicate nonexistent storage channels [8, 9], a serious problem that exists in all previous such automated tools [3, 5, 12, 14].

4.2 Integration of Timing–Channel Analysis into the Automated System

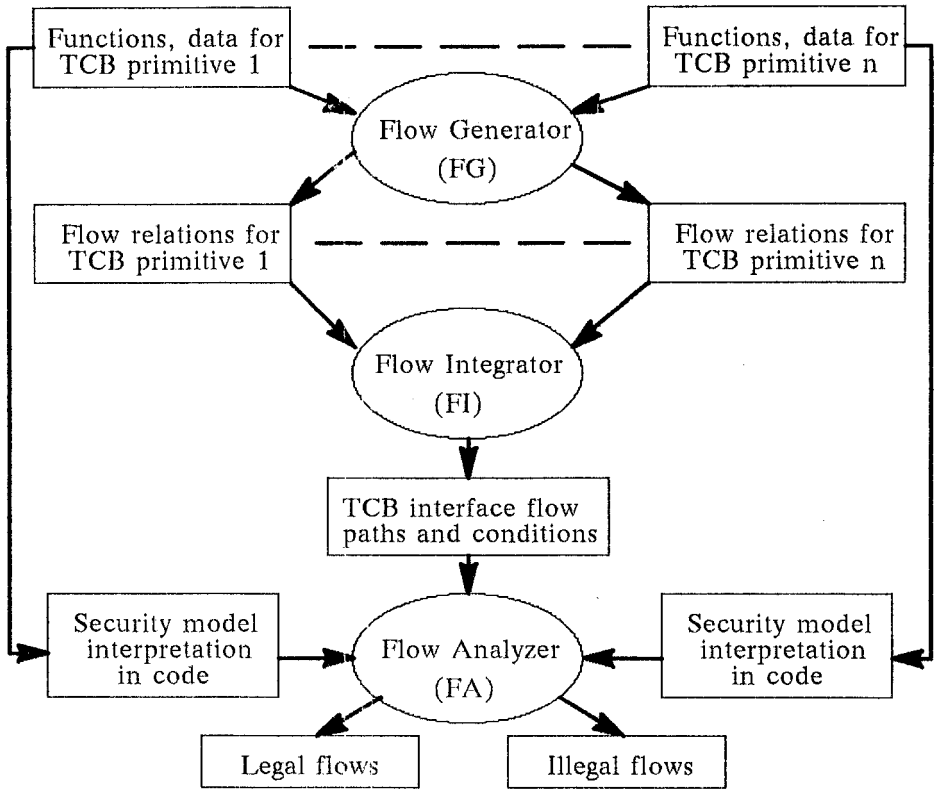


Fig. 4.1. Structure of the Automated Covert-Channel Analysis System

The FI in the automated system derives global flows based on the characteristics of storage channels. That is, it derives an altering flow in a TCB primitive to a global variable and a viewing flow in the same or a different TCB primitive from the same global variable to a user interface variable. Since timing channels are characterized in this paper in a similar way to storage channels, i.e., they are described in terms of altering and reading some system state (global variables and user interface variables), the formal methods can be easily incorporated into the FI for it to derive global flows that lead to potential timing channels. Specifically, to detect a global flow that leads to a tightly coupled direct timing channel, we use the same criteria to that in the FI for storage channels to determine the altering flow and implement the "viewing part" in the FI to determine if the same global variable exists in any condition expression of an alteration or iteration statement. Similarly, to identify a global flow that leads to a loosely coupled direct timing channel, we augment the FI to derive the altering flow by determining if a user interface variable appears in any condition expression of an alteration or iteration statement but need not do anything for the "viewing part". With all the functions necessary for deriving various kinds of global flows in place, e.g., deciding direct flows and inferred flows [5, 9, 16] and tracing

aliases among variables [9, 16], etc., the implementation of an automated tool for timing-channel identification requires no extra effort beyond the augmentation of the FI with a few additional criteria for deriving altering and viewing flows. Little change is needed in the FG and in the FA because (1) flow generation does not involve channel characteristics and (2) flow analysis employs a technique independent of any specific format of flows and, thus, can be applied directly for determining illegal flows leading to potential timing channels [8].

5 Hardware Timing-Channel Exceptions

Systematic covert-channel identification conducted statically is not complete unless all potential channels can be identified. It is shown in [16] that source-code analysis is sufficient to detect all potential storage channels. This claim cannot be applied to timing channels, however. This is because, while the shared global variable that is used as the medium of information transmission in a storage channel exists independently of hardware, the process response time that is used for encoding information in a timing channel can vary from system to system depending on system configurations.

We have shown that the existence of alternation and iteration statements in source code makes it possible for the existence of timing channels. These channels exist in each and every system that executes the source code independent of system structures. Hardware timing channels, on the other hand, may exist in some systems but may not exist in all systems. This dynamic nature of hardware timing channels makes them undeterminable in source-code analysis. Taking an assignment statement for example, the operands of the statement may already be in the main memory, may need to be fetched from the secondary storage, or may have to be keyed in from a user terminal. Depending on the whereabouts of the operands, the response time of executing the statement can be quite different. One can thus take advantage of the above scenario to create a timing channel through the execution of the two programs shown in Fig. 5.1. This timing channel is of dynamic nature because its existence depends on the hardware conditions (1) that the system must use a storage hierarchy offering different access times, (2) that user programs and data permanently reside in the secondary storage and the use of the main memory is only for performance purpose, (3) that memory management in the system uses the policy of demand on request, that is, programs and data are loaded into the main memory only when the system receives an explicit request, and (4) that the system does not immediately invalidate all the global data in the main memory upon the termination of a process but only when insufficient memory is left does the system selectively invalidate the appropriate area based on some predefined replacement policy. Hence, when the sending process wants to send a "zero", it loads *gv* into the main memory so that the receiving process will experience a shorter response time to access *gv*. Conversely, when the sending process wants to send a "one", it does not load *gv* into the main memory so that the receiving process will experience a longer response time to access *gv*. This channel would not exist should any of the above conditions not hold. Also, this channel is not always reliable. For example, the value of *gv* may not still be

Program for the sending process

```
sender(signal)
{
extern int gv, gu;
int i;
if (signal)
   i = gu;
else i = gv;
}
```

Program for the receiving process

```
receivier()
{
extern int gv;
printf('%d', gv);
}
```

Sending process
1. Modulate the use: load gv into main memory;
2. Send a one: do not load gv;
3. Send a zero: load gv.

Receiving process
1. Receive data: observe response time;
2. Receive a one: a longer response time;
3. Receive a zero: a shorter response time.

Fig. 5.1. A dynamic and system-depedent timing channel

residing in the main memory when the receiving process starts to execute in order to receive a "zero" because *gv* may have already been preempted due to requests for the main memory from some other processes in the system. Therefore, hardware timing channels have a certain level of noise determined by specific system structures and execution environments.

The technique presented in this paper is only applicable to source-code analysis and cannot be relied upon for detecting all types of potential timing channels. Hence, source-code analysis alone is not sufficient to determine all potential timing channels that may appear in final systems.

6 Conclusion

We characterized the behaviors of timing channels that are reflected in source code. This characterization differs significantly from previous ones [11, 16] and immediately leads to formal methods for timing-channel identification in TCB source code. Algorithms are presented and, coupled with the various techniques that have been previously developed for storage-channel identification, such as information-flow analysis [1, 2, 9, 11, 16] and variable aliasing [9, 16], timing-channel identification can be carried out in a systematic way in TCB source code. Because of the similar approach we take in characterizing timing channels to that in characterizing storage channels, automated tools that have been previously built for storage-channel identification [3, 9, 12, 14] can be augmented with minimal effort for timing-channel identification. In particular, we discussed how to integrate the algorithms into the automated system described in [9].

The shortcoming of source-code analysis is that it cannot reveal timing channels that result from hardware configurations. These channels, however, are generally difficult to use and are subject to a higher level of vulnerability during real-time information transmission. These channels can also be blocked or made virtually useless by changing system configurations and management policies.

Acknowledgment

The authors are grateful to Matthew Hecht and Janet Cugini of IBM, and Shyh-Wei Luan of VDG Inc. for reviewing early versions of this paper. The authors would also like to thank Tom Tamburo of IBM for his continuous support during this research work.

References

1. Andrews, G. R. and R. P. Reitman, "An Axiomatic Approach to Information Flow in Programs," *ACM Trans. Prog. Lang. Syst.*, Vol. 2, No. 1, Jan. 1980, pp. 56–76.
2. Denning, D. E., "A Lattice Model of Secure Information Flow," *Comm. ACM*, Vol. 19, No. 5, May 1976, pp. 236–243.
3. Eckmann, S. T., "Ina Flo: The FDM Flow Tool," in *Proc. 10th Nat'l Compt. Sec. Conf.*, NBS, Gaithersburg, MD, 1987, pp. 175–182.
4. Feiertag, R. J., "A Technique for Proving Specifications Are Multilevel Secure," *Computer Science Lab Report CSL–109*, SRI, Menlo Park, CA, 1980.
5. Gasser, M., *Building A Secure Computer System*, Van Nostrand Reinhold Company, New York, NY, 1988.
6. Gligor, V. D., C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W.-D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan, "Design and Implementation of Secure Xenix," *IEEE Trans. Software Engr.*, Vol. SE–13, No. 2, Feb. 1987, pp. 208–221.
7. Haigh, J. T., R. A. Kemmerer, J. McHugh, and W. D. Young, "An Experience Using Two Covert Channel Analysis Techniques on a Real System Design," *IEEE Trans. Software Engr.*, Vol. SE–13, No. 2, Feb. 1987, pp. 157–168.
8. He, J. and V. D. Gligor, "Information-Flow Analysis for Covert-Channel Identification in Multilevel Secure Operating Systems," in *Proc. Computer Security Foundations Workshop III*, Franconia, NH, June 1990.
9. He, J., *An Automated System for the Identification of Potential Covert Channels in Multilevel Secure Operating Systems*, Ph.D. Dissertation, University of Maryland, College Park, MD, Dec. 1990.
10. Hu, W.-M., "Reducing Timing Channels with Fuzzy Time," in *Proc. IEEE Symp. Research on Security and Privacy*, Oakland, CA, May 1991.

11. Kemmerer, R. A., "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels," *ACM Trans. Compt. Syst.*, Vol. 1, No. 3, Aug. 1983, pp. 256–277.

12. Kramer, S. M., "The Mitre Flow Table Generator – Volume 1," *M83–31 Volume 1*, Mitre Corporation, Bedford, MA, Jan. 1983.

13. Lampson, B. W., "A Note on the Confinement Problem," *Comm. ACM*, Vol. 16, No. 10, Oct. 1973, pp. 613–615.

14. McHugh, J. and D. I. Good, "An Information Flow Tool for Gypsy," in *Proc. IEEE Symp. Security and Privacy*, Oakland, CA, April 1985, pp. 46–48.

15. *Trusted Computer System Evaluation Criteria*, U. S. Dept. of Defense Standard DOD 5200.28–STD, Dec. 1985.

16. Tsai, C.–R., V. D. Gligor, and C. S. Chandersekaran, "On the Identification of Covert Storage Channels in Secure Systems," *IEEE Trans. Software Engr.*, Vol. 16, No. 6, June 1990, pp. 569–580.

17. Wray, J. C., "An Analysis of Covert Timing Channels," in *Proc. IEEE Symp. Research on Security and Privacy*, Oakland, CA, May 1991, pp. 2–7.