available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A formal framework for real-time information flow analysis

Joon Son[*], *Jim Alves-Foss*

Center for Secure and Dependable Systems, University of Idaho, P.O. Box 441008, Moscow, ID 83844-1008, USA

ARTICLE INFO

Article history:
Received 22 October 2007
Received in revised form
13 January 2009
Accepted 15 January 2009

Keywords:
Real-time information flow
Covert-timing channel free security policies
Specification of security policies

ABSTRACT

We view Multi-Level Secure (MLS) real-time systems as systems in which MLS real-time tasks are scheduled and execute, according to a scheduling algorithm employed by the system. From this perspective, we develop a general trace-based framework that can carry out a covert-timing channel analysis of a real-time system. In addition, we propose a set of covert-timing channel free policies: If a system satisfies one of our proposed security policies, we demonstrated that the system can achieve a certain level of real-time information flow security. Finally, we compare the relative strength of the proposed covert-timing channel free security policies and analyze whether each security policy can be regarded as a property (a set of execution sequences).

1. Introduction

When timing information is added to the specification of a Multi-Level Secure (MLS) system, it may reveal new information flow: if a classified entity, *High*, can modify the response time of an unclassified entity, *Low*, and *Low* can reliably deduce that the change in its response time is caused by *High*, *High* can use this as a mean to transmit illicit information to *Low*. This timing based information flow leakage is often called a covert-timing channel. As illustrated in Fig. 1, a covert (timing) channel has a transmission cycle which consists of a sender–receiver synchronization (S–R) period, a transmission period and a feedback period. During the S–R period, a sender (*High*) will notify a receiver (*Low*) that it is ready to transmit new information. However, the S–R period may not be necessary if the sender and the receiver have some prior agreement (e.g. every *t* units of time, new information is transmitted). In this paper, we assume there exists a prior agreement between the sender and receiver. Therefore, unless stated otherwise, it is assumed that there is no sender–receiver synchronization (S–R) period. If we assume there is an unreliable communication channel between a sender and a receiver, a feedback period must exist to establish reliable communication. Without feedback, the sender is unaware if the receiver has observed the intended information and does not know when to start a new transmission.

Numerous papers (Agat, 2000; Cabuk et al., 2004; Hu, 1991; Janeri et al., 1995; Moskowitz and Kang, 1994; Shah et al., 2006; Son and Alves-Foss, 2006a,b; Gray, 1993) have presented covert-timing channel (real-time information flow) analysis for various MLS systems. We are specifically interested in real-time systems (Liu, 2000) where time plays a fundamental role and thus timing information must be included in system specifications. Previously, formal frameworks such as timed process algebra (Focardi et al., 2003; Lowe, 2004) or timed automata (Barbuti and Tesei, 2003), were used to model timed behaviors of real-time systems and to specify security policies for preventing illicit information leakage through covert-timing channels. For the analysis of information flow security in non-real-time systems, a trace-based possibilistic formal framework or model has been quite popular (Mantel, 2000; McLean, 1996; Zakinthinos and Lee, 1997). In the trace-based

* *Corresponding author.*
E-mail address: joonson@gmail.com (J. Son).
0167-4048/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2009.01.005

possibilistic model, the non-timed behavior of a system is modeled by a set of traces, where every trace represents a possible execution sequence of the system and nondeterministic behavior is modeled by the possibility of different execution sequences. In the possibilistic model, system specifications do not require probabilities with which the different possible execution sequence will occur. In this paper, we present a new trace-based possibilistic framework which allows us to:

1) formally specify traces (histories) of *timed* actions of real-time tasks running under a real-time scheduling algorithm.
2) define security policies which specify how real-time tasks should behave against covert-timing attacks.
3) formally show, if a system running under a real-time scheduling algorithm satisfies the proposed security policies, that the system can achieve a high level of real-time information flow security. This satisfactory relation between a system specification and a security policy can reveal that the schedule of tasks generated by a real-time scheduler could leak information through a covert-timing channel.
4) compare the proposed security policies in terms of their relative strength and specification characteristics.

2. Our approach to a formal framework

Designating the level of abstraction to specify timed behaviors of a real-time system is crucial in our approach. A real-time system should function as follows (see Section 3 for more details): when multiple tasks arrive at a scheduler for execution, the way in which scheduling priorities are assigned to tasks (if a scheduling algorithm is priority based) and the way in which a task is scheduled to execute on the CPU are determined by the type of a scheduling algorithm in use and the (timing) constraints imposed on task executions. Thus, we model a real-time system as a set of tasks which execute according to a scheduling algorithm to meet their timing constraints. In this paper, we assume that a set of tasks consists of tasks with different security levels, e.g. a high-level (classified) task $\mathbf{T}_H$, a low-level (unclassified) task $\mathbf{T}_L$, and a third party[1] (trusted) task $\mathbf{T}_N$ (Fig. 2). The system model we employ throughout this paper is a trace or history-based model, i.e. the timed behavior of a system is modeled by a set of traces where every trace represents a possible timed execution sequence of a real-time task running in the system.

In order to accurately define real-time information flow from *High* to *Low*, we first need to specify what *Low* can observe and what it can reliably deduce from its observation, as well as how *High* can affect *Low*'s observation. As shown in Fig. 2, the model assumes *Low* is able to assess the response time (the time between the submission of its task to a system and the output event notifying that it is completed) of a low-level task; however, we assume that there is a limit to how accurately *Low* can record time. Our assumption is that *Low* cannot measure the time between any two occurrences of context switch. When a high-level task $\mathbf{T}_H$ preempts a low-

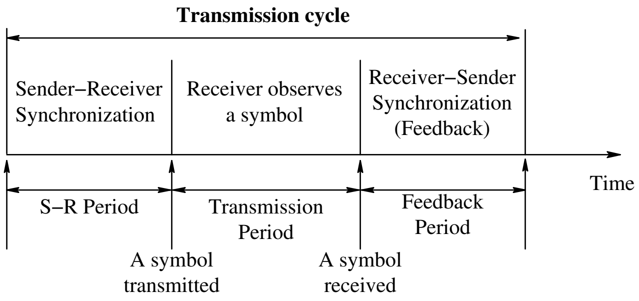

Fig. 1 – Transmission cycle of a covert-timing channel.

level task $\mathbf{T}_L$ while $\mathbf{T}_L$ is running, or $\mathbf{T}_H$ locks a resource shared by $\mathbf{T}_L$, thereby blocking an execution of $\mathbf{T}_L$, the response time of $\mathbf{T}_L$ is affected. The delay or change of the response time of $\mathbf{T}_L$ could be used as means of transmitting illicit information. Note that a third party (trusted) task $\mathbf{T}_N$ can also affect the response time of $\mathbf{T}_L$ in the same way that $\mathbf{T}_H$ affects $\mathbf{T}_L$. In our formal framework, the existence of covert-timing channels means that, upon observing the response time of a low-level task, *Low* can reliably deduce a sequence of timed actions or traces performed by a high-level task.

3. Real-time systems

3.1. *Abstract model of real-time systems*

We assume that a real-time system consists of the following components.

- A set of computational real-time tasks that compete for shared resources. These real-time tasks are typically assumed to be software processes or threads.
- A run-time scheduler (or dispatcher) that controls which task is executing at any given moment.
- A set of shared resources used by real-time tasks. These shared resources may include software variables, data bus, CPU, mutual exclusion control, variables, and memory.

3.2. *Lifecycle of a real-time task*

In a real-time system, which supports the execution of concurrent tasks on a single processor, a task can be defined as being in at least six different unique states:

- Running state – a task with highest priority enters this state as it starts executing on the processor.

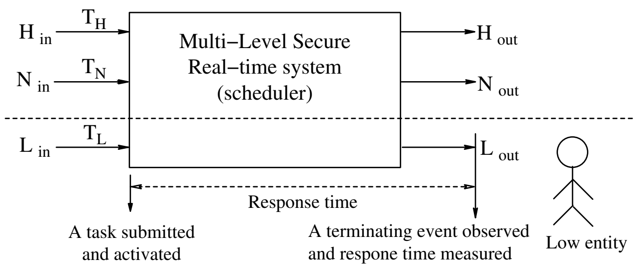
H in
T_H
N in
T_N
L in
T_L
Multi–Level Secure Real–time system (scheduler)
H out
N out
L out
Response time
A task submitted and activated
A terminating event observed and respone time measured
Low entity

Fig. 2 – A system model – task arrival & termination.

[1] The third party task $\mathbf{T}_N$ could be considered as a composite of tasks running in a system other than $\mathbf{T}_H$ and $\mathbf{T}_L$.

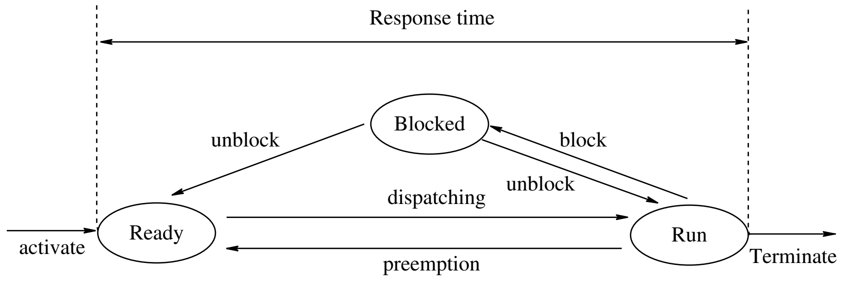

Fig. 3 – Minimum state transition diagram of a task.

- Ready state – a task in the ready state is ready to run but cannot because a higher priority task is executing.
- Blocked state – a task enters the blocked state if it has requested a busy (unavailable) resource or has delayed itself for some duration.
- Null state – a task has not been submitted to a scheduler.
- Activation state – when a task in the null state is submitted to a scheduler for execution, the task becomes activated.
- Termination state – when a task finishes its computation, it terminates; the task state changes from the termination state to the null state.

A task moves from one state to another according to the state transition diagram shown in Fig. 3. When a real-time task arrives at a run-time scheduler for execution, the task becomes activated. The task is placed in a ready queue and turns to the ready state. The queue is ordered according to the particular run-time scheduling algorithm in force. The task at the ready queue is released for execution and enters the running state. While the task is in the running state, it can be preempted by the arrival of another task with a higher priority. Once preempted, it moves back to the ready state. The running task can also move to the blocked state when blocking on a shared resource occurs or the task delays itself for some duration. A blocked task remains blocked until the blocked resource becomes available or an event (signal) the task is waiting for occurs. When the task is no longer blocked, it moves to the ready state, or it moves directly to the running state by preempting the currently running task if it is the highest priority task. Note that Fig. 3 shows only the minimum state transition diagram[2] of a real-time task.

We can represent the state of a task using six different atomic propositions (boolean variables). Let $\mathbf{T}_{id}$ be a task with the unique identifier *id*. Then, the different atomic propositions are $\varnothing(\mathbf{T}_{id})$, $act(\mathbf{T}_{id})$, $ready(\mathbf{T}_{id})$, $block(\mathbf{T}_{id})$, $run(\mathbf{T}_{id})$ and $term(\mathbf{T}_{id})$, which describe $\mathbf{T}_{id}$ being null ($\mathbf{T}_{id}$ has not been submitted to a scheduler), activated, ready, blocked, running, and terminated, respectively.

4. Timed Kripke Structures (TKS)

In this paper, we choose to use the Timed Kripke Structure (*TKS*) (Laroussinie et al., 2002, 2006; Logothetis and Schneider, 2001a,b) to describe timed behaviors of real-time tasks running under a scheduling algorithm for the following reasons:

- Since *TKS* is a simple extended version of a state transition system where each transition is labeled by an integer number which denotes the time required to move from one state to another, it is easy to specify a timing constraint between two consecutive actions performed by a task. From *TKS*, it is also easy to obtain the execution traces (histories) of real-time tasks.
- With *TKS*, we can easily specify actions performed simultaneously[3] by (independent) real-time tasks, e.g. simultaneous arrival of independent tasks at a scheduler.
- Formal verification (model checking) of timed properties of a system over *TKS* is relatively efficient.[4]

TKS is an extended version of Kripke Structure (*KS*). Formally it is defined as follows, where $\mathbf{N}$ is the set of non-negative integers.

Definition 1. Timed Kripke Structure *of a system is a tuple* $\mathcal{M} = (\mathcal{AP}, \mathcal{S}, \mathcal{S}_{init}, \mathcal{R}, \mathcal{L})$, where:

- $\mathcal{AP}$ is a finite set of atomic propositions,
- $\mathcal{S}$ is a finite set of states,
- $\mathcal{S}_{init} \subseteq \mathcal{S}$ is a set of initial states,
- $\mathcal{R} \subseteq \mathcal{S} \times \mathbf{N} \times \mathcal{S}$ is a set of transitions. A transition $(s_i, d_i, s_{i+1}) \in \mathcal{R}$ is denoted $s_i \xrightarrow{d_i} s_{i+1}$, by where d_i is the duration of execution in the state s_i.
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{AP}}$ is a labeling function; for any state s, $\mathcal{L}(s) \subseteq \mathcal{AP}$ is a set of atomic propositions that hold on s.

The difference of *TKS* from *KS* is that in *TKS*, $\mathcal{R} \subseteq \mathcal{S} \times \mathbf{N} \times \mathcal{S}$ is a finite set of *transitions* labeled by a non-negative integer, called the *duration* of the transition, whereas *KS* has no duration.

A (computation) *path* in a *TKS* is an infinite sequence of states $s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \cdots$ such that $(s_i, d_i, s_{i+1}) \in \mathcal{R}$ for all $i \geq 0$. A finite sequence of transitions $s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_0} s_2 \ldots s_n$ in *TKS* is called a *finite* computation path.

[2] Many commercial real-time systems (kernels) define additional states such as suspended, pended, delayed, etc.

[3] An alternative formalism that could be used is a timed process algebra equipped with a true concurrency semantics (Fidge and Zic, 1995) which can represent true parallelism.

[4] Model checking timed properties over Timed Automata (Alur et al., 1993) easily suffers from the state explosion problem. However, model checking of a wide range of timed properties over *TKS* can be done in polynomial time.

4.1. Timed computation tree

In *KS*, a computation path is formed by designating state s_R as initial and then unwinding the *KS* into an infinite tree with the designated state s_R as the root. Thus, for any *KS* and state $s_R \in \mathcal{S}$, there is an infinite computation tree with root labeled s_R such that (s_i, s_{i+1}) is an arc in the tree if and only if $(s_i, s_{i+1}) \in \mathcal{R}$ in *KS*.

The same concept is applied to *TKS* to construct a Timed Computation Tree. The Timed Computation Tree can be constructed by unfolding *TKS* into an infinite tree. Similarly, for any *TKS* and state $s_R \in \mathcal{S}$, there is an infinite computation tree with root labeled s_R such that (s_i, d_i, s_{i+1}) is an arc in the tree if and only if $(s_i, d_i, s_{i+1}) \in \mathcal{R}$ in *TKS*. The Timed Computation Tree is a basis used for defining real-time information flow requirements in Section 6.

5. A system model

5.1. Labeling states to model timed behaviors of a task

It is important that a state s_i of *TKS* should be properly and consistently labeled to model timed behaviors of a task. We assume that a real-time system has a set $\Gamma_{\mathcal{A}}$ of tasks which constantly change their task states according to a run-time scheduling algorithm $\mathcal{A}$. The task state of each task $\mathbf{T}_{id}$ running in a system is described by an atomic proposition $P_{id} \in \mathcal{AP}_{id}$, where

$$\mathcal{AP}_{id} = \{\varnothing(\mathbf{T}_{id}), act(\mathbf{T}_{id}), ready(\mathbf{T}_{id}), block(\mathbf{T}_{id}), run(\mathbf{T}_{id}), term(\mathbf{T}_{id})\}$$

For all states $s_i \in \mathcal{S}$, $\mathcal{L}(s_i)$ contains the proposition $P_{id} \in \mathcal{AP}_{id}$ for each task $\mathbf{T}_{id}$; $\mathcal{L}(s_i)$ has the following form if there are *n* number of tasks running concurrently under scheduling algorithm $\mathcal{A}$, i.e. $\Gamma_{\mathcal{A}} = \{\mathbf{T}_{id} | id \in \{1, \ldots, n\}\}$ and $\mathcal{AP} = \cup_{id \in \{1,\ldots,n\}} \mathcal{AP}_{id}$:

$$\mathcal{L}(s_i) = \{P_{id} | id \in \{1, \ldots, n\} \wedge P_{id} \in \mathcal{AP}_{id}\}$$

A state s_i of *TKS* progresses to the next state s_{i+1} if one or more tasks $\mathbf{T}_{id}$ in $\Gamma_{\mathcal{A}}$ changes their task states, i.e. $P_{id} \in \mathcal{L}(s_i) \neq P'_{id} \in \mathcal{L}(s_{i+1})$; for the remaining tasks with no change in their task states, their atomic propositions at the current state s_{i+1} remain the same as at the previous state s_i.

Let s_d be a state with the duration of $d > 0$ units of time and s_e be a state with zero duration ($d = 0$). We call an atomic proposition P_{id} an *action* of a task $\mathbf{T}_{id}$ if $P_{id} \in \mathcal{L}(s_d)$. An atomic proposition P_{id} is called an *event* of a task $\mathbf{T}_{id}$ if $P_{id} \in \mathcal{L}(s_e)$. An execution step of a task is represented as an interleaving sequence of *actions* and *events*. We denote an action $P_{id} \in \mathcal{L}(s_d)$ of $\mathbf{T}_{id}$ which requires the task to take $d > 0$ units of time by P_{id}^d. For example, $run(\mathbf{T}_{id})^d$ is used to represent the execution of $\mathbf{T}_{id}$ for *d* units of time.

An event of a task is used to describe the following incidents:

- The event of task activation: a task is activated and placed in a ready queue. There are two cases to consider. The first case is that a task is activated immediately after the system starts (at time zero). The second case is that a task arrives and is activated while other tasks are running. Let $s_i \in \mathcal{S}$, $i > 0$, and $s_0 \in \mathcal{S}_{init}$. The first case is expressed as $s_0 \xrightarrow{0} s_1 \cdots$, where $act(\mathbf{T}_{id}) \in \mathcal{L}(s_0)$, and $ready(\mathbf{T}_{id}) \in \mathcal{L}(s_1)$. The second case is denoted by $\cdots \to s_i \xrightarrow{d} s_{i+1} \xrightarrow{0} s_{i+2} \cdots$, where $d > 0$, $\varnothing(\mathbf{T}_{id}) \in \mathcal{L}(s_i)$, $act(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+1})$, and $ready(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+2})$.
- The event of task termination: task termination is expressed as $\cdots \to s_i \xrightarrow{d} s_{i+1} \xrightarrow{0} s_{i+2} \cdots$, where $d > 0$, $run(\mathbf{T}_{id}) \in \mathcal{L}(s_i)$, $term(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+1})$ and $\varnothing(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+2})$.
- The event of an immediate release[5] (execution) of a task $\mathbf{T}_{id}$ after it is activated: this immediate release of $\mathbf{T}_{id}$ is expressed in *TKS* as follows: $s_i \xrightarrow{0} s_{i+1} \xrightarrow{0} s_{i+2} \cdots$, where $i \geq 0$, $act(\mathbf{T}_{id}) \in \mathcal{L}(s_i)$, $ready(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+1})$, and $run(\mathbf{T}_{id}) \in \mathcal{L}(s_{i+2})$.

In summary, $run(\mathbf{T}_{id})$ and $block(\mathbf{T}_{id})$ are always used to represent an action of a task, and $act(\mathbf{T}_{id})$ and $term(\mathbf{T}_{id})$ always denote an event. The atomic proposition $ready(\mathbf{T}_{id})$ is used as an action to describe a delay before execution or as an event to describe the immediate release (execution) of a task. The atomic proposition $\varnothing(\mathbf{T}_{id})$ is used as an action to denote the passage of time during which a task is in the null state or as an event to denote a task being null in a state $s_e \in \mathcal{S}$.

Example 2. Assume $\Gamma_{\mathcal{A}} = \{\mathbf{T}_N, \mathbf{T}_H, \mathbf{T}_L\}$. The set $\Gamma_{\mathcal{A}}$ of tasks is assumed to share no resource between them (no task can be blocked on a busy resource but can delay itself for some duration) and executes according to the timing diagram of Fig. 4. There are a few things worth noting in the timing diagram: at time 5, $\mathbf{T}_H$ delays itself for three units of time and $\mathbf{T}_L$ in the ready queue experiences a delay of one unit of time before execution. At time 6, $\mathbf{T}_L$ begins executing. At time 8, $\mathbf{T}_H$ preempts $\mathbf{T}_L$ and begins executing, and $\mathbf{T}_L$ moves back to the ready queue. We construct the corresponding TKS as shown in Fig. 5. Each state of the TKS is labeled as described above.

5.2. Real-time trace

We define a (real-time) history of a system as:

Definition 3. For a computation path $\pi = s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} s_2 \cdots s_n$ in TKS, a history $\tau(\pi)$ is an ordered sequence of sets of atomic propositions, $i \geq 0$, where the set $\mathcal{L}(s_i)$ holds in state s_i for d_i units of time. Thus, we denote a history $\tau(\pi)$ by $\tau(\pi) = \mathcal{L}(s_0)^{d_0} \cdot \mathcal{L}(s_1)^{d_1} \cdot \ldots \mathcal{L}(s_i)^{d_i} \cdot \ldots \mathcal{L}(s_n)^{d_n}$.

A history $\tau(\pi)$ describes timing behaviors of a set $\Gamma_{\mathcal{A}}$ of real-time tasks. For convenience when $d_i = 0$, we omit d_i from the history. To obtain a real-time behavior of an individual task $\mathbf{T}_{id}$ from a history, we introduce a restriction operator $|_{id}$:

Definition 4. Let $\tau(\pi) = \mathcal{L}(s_0)^{d_0} \cdot \mathcal{L}(s_1)^{d_1} \cdot \ldots \mathcal{L}(s_i)^{d_i} \cdot \ldots \mathcal{L}(s_n)^{d_n}$. We define $\tau(\pi)|_{id}$ as an ordered sequence of propositions $\rho_i \in \mathcal{AP}_{id}$ with a superscript d_i, $i \geq 0$, formed by extracting ρ_i from each $\mathcal{L}(s_i)$; thus, $\tau(\pi)|_{id} = \rho_0^{d_0} \cdot \rho_1^{d_1} \cdot \ldots \rho_n^{d_n}$, where $\rho_i \in \mathcal{AP}_{id}$ and $\rho_i \in \mathcal{L}(s_i)$. We call $\tau(\pi)|_{id}$ a trace of an individual task $\mathbf{T}_{id}$. The same consecutive propositions listed in a trace can be simplified as: $\rho_i^{d_0} \cdot \ldots \cdot \rho_{i+k}^{d_k} = \rho_i^{d_0 + \cdots + d_k}$, where $\rho_i = \rho_{i+1} \ldots = \rho_{i+k}$.

[5] A task $\mathbf{T}_i$ is activated when $\mathbf{T}_i$ arrives at the scheduler. The scheduler makes the task $\mathbf{T}_i$ runnable (place it in the running state) by *releasing* it (if there are no tasks of higher scheduling priority than $\mathbf{T}_i$ in a priority-driven real-time system). Thus, a task starts to execute after it is released.

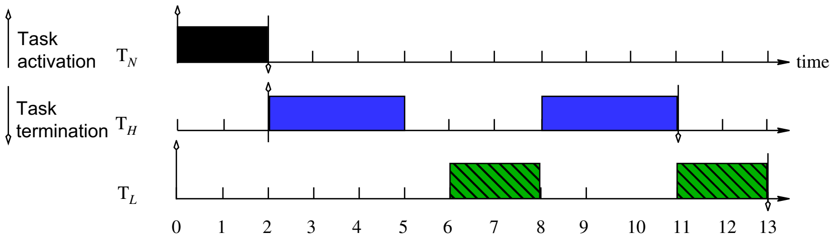

Fig. 4 – Timing diagram.

Definition 5. $\mathcal{D}(\tau(\pi)|_{id})$ denotes the cumulative duration of the trace $\tau_\pi|_{id}$ of an individual task $\mathbf{T}_{id}$:

$$\mathcal{D}(\tau(\pi)|_{id}) = \sum_{i=0}^{n} d_i, \text{ where } \tau(\pi)|_{id} = \rho_0^{d_0}\ldots\rho_n^{d_n}$$

Example 6. Let π_1 be the computation path shown in Fig. 5. Then, the history $\tau(\pi_1)$ is:

$$\tau(\pi_1) = \{act(\mathbf{T}_N), \varnothing(\mathbf{T}_H), act(\mathbf{T}_L)\}\cdot\{ready(\mathbf{T}_N), \varnothing(\mathbf{T}_H), ready(\mathbf{T}_L)\}\cdot \{run(\mathbf{T}_N), \varnothing(\mathbf{T}_H), ready(\mathbf{T}_L)\}^2\cdot\{term(\mathbf{T}_N), act(\mathbf{T}_H), ready(\mathbf{T}_L)\}\cdot \{\varnothing(\mathbf{T}_N), ready(\mathbf{T}_H), ready(\mathbf{T}_L)\}\cdot\{\varnothing(\mathbf{T}_N), run(\mathbf{T}_H), ready(\mathbf{T}_L)\}^3\cdot \{\varnothing(\mathbf{T}_N), block(\mathbf{T}_H), ready(\mathbf{T}_L)\}^1\ldots\{\varnothing(\mathbf{T}_N), \varnothing(\mathbf{T}_H), term(\mathbf{T}_L)\}$$

The trace of $\mathbf{T}_L$ is:

$$\begin{aligned}\tau(\pi_1)|_L = {} & act(\mathbf{T}_L)\cdot ready(\mathbf{T}_L)\cdot ready(\mathbf{T}_L)^2\cdot ready(\mathbf{T}_L)\cdot ready(\mathbf{T}_L) \\ & \cdot ready(\mathbf{T}_L)^3\cdot ready(\mathbf{T}_L)^1\cdot run(\mathbf{T}_L)^2\cdot ready(\mathbf{T}_L)^3\cdot ready(\mathbf{T}_L) \\ & \cdot run(\mathbf{T}_L)^2\cdot term(\mathbf{T}_L) \qquad = act(\mathbf{T}_L)\cdot ready(\mathbf{T}_L)^{0+0+2+0+0+3+1} \\ & \cdot run(\mathbf{T}_L)^2\cdot ready(\mathbf{T}_L)^{3+0}\cdot run(\mathbf{T}_L)^2\cdot term(\mathbf{T}_L) \\ = {} & act(\mathbf{T}_L)\cdot ready(\mathbf{T}_L)^6\cdot run(\mathbf{T}_L)^2\cdot ready(\mathbf{T}_L)^3\cdot run(\mathbf{T}_L)^2 \\ & \cdot term(\mathbf{T}_L)\end{aligned}$$

Furthermore, $\mathcal{D}(\tau(\pi_1)|_L) = 0+6+2+3+2+0 = 13$. Note that, in this example, $\mathcal{D}(\tau(\pi_1)|_L)$ is the response time of $\mathbf{T}_L$ since $\tau(\pi_1)|_L$ represents a sequence of events and actions occurring during the time between $act(\mathbf{T}_L)$ and $term(\mathbf{T}_L)$.

We introduce a function *RUN* which takes a trace $\tau(\pi)|_{id}$ of $\mathbf{T}_{id}$ as an input and returns an execution trace of $\mathbf{T}_{id}$ revealing only $act(\mathbf{T}_{id})$ and $run(\mathbf{T}_{id})$ from $\tau(\pi)|_{id}$. *High* uses the execution trace $RUN(\tau(\pi)|_{id})$ as the means of transmitting illicit information to *Low*.

Definition 7. Given a trace $\tau(\pi)|_{id}$, the execution trace $RUN(\tau(\pi)|_{id})$ is obtained by removing all the propositions except $act(\mathbf{T}_{id})$ and $run(\mathbf{T}_{id})$ from $\tau(\pi)|_{id}$ and adding a notation $\downarrow_{n_i}$ right after $act(\mathbf{T}_{id})$ to denote the activation time of $\mathbf{T}_{id}$. Thus, $RUN(\tau(\pi)|_{id})$ has the following form:

$$RUN(\tau(\pi)|_{id}) = \begin{cases} <>_{run}, \text{ if no } run(\mathbf{T}_{id}) \text{ is enabled in } \tau(\pi)|_{id}; \\ act(\mathbf{T}_{id})\downarrow_{n_1}\cdot run(\mathbf{T}_{id})^{k_1}\cdot act(\mathbf{T}_{id})\downarrow_{n_2}\cdot run(\mathbf{T}_{id})^{k_2}\cdot \\ \cdots .act(\mathbf{T}_{id})\downarrow_{n_i}\cdot run(\mathbf{T}_{id})^{k_i}\cdot\ldots, \text{ otherwise.}\end{cases}$$

$\downarrow_{n_i}$ used in $act(\mathbf{T}_{id})\downarrow_{n_i}$ denotes the time n_i of the i^{th} activation of $\mathbf{T}_{id}$. If $\mathbf{T}_{id}$ is a periodic task with period p, $n_{i+1} = n_i + p$. k_i used in $run(\mathbf{T}_{id})^{k_i}$ denotes the total execution (*run*) time of $\mathbf{T}_{id}$ during the i^{th} activation interval. We call $RUN(\tau(\pi)|_{id})$ the *run* trace of $\mathbf{T}_{id}$.

Example 8. Let $\tau(\pi_1)|_L$ be the trace of $\mathbf{T}_L$ we used in Example 6, i.e.
$\tau(\pi_1)|_L = act(\mathbf{T}_L).ready(\mathbf{T}_L)^6.run(\mathbf{T}_L)^2.ready(\mathbf{T}_L)^3.run(\mathbf{T}_L)^2.term(\mathbf{T}_L)$.
Note that $\mathbf{T}_L$ is activated at time 0 and has only one activation interval (not a periodic task). In addition, during the interval, the total execution time of $\mathbf{T}_{id}$ is 4 units of time since there are two *runs* ($run(\mathbf{T}_L)^2$ and $run(\mathbf{T}_L)^2$) in $\tau(\pi_1)|_L$. Then, the run trace of $\mathbf{T}_L$ is:

$$\begin{aligned}RUN(\tau(\pi)|_{id}) &= act(\mathbf{T}_{id})\downarrow_0\cdot run(\mathbf{T}_{id})^2\cdot run(\mathbf{T}_{id})^2 \\ &= act(\mathbf{T}_{id})\downarrow_0\cdot run(\mathbf{T}_{id})^4\end{aligned}$$

Definition 9. Let $RUN(\tau(\pi)|_{id}) = act(\mathbf{T}_{id})\downarrow_{n_1}\cdot run(\mathbf{T}_{id})^{k_1}\cdot act(\mathbf{T}_{id})\downarrow_{n_2}\cdot run(\mathbf{T}_{id})^{k_2}\ldots$ and $RUN(\tau(\pi')|_{id}) = act(\mathbf{T}_{id})\downarrow_{n'_1}\cdot run(\mathbf{T}_{id})^{k'_1}\cdot act(\mathbf{T}_{id})\downarrow_{n'_2}\cdot run(\mathbf{T}_{id})^{k'_2}\ldots$ Any two run traces $RUN(\tau(\pi)|_{id})$ and $RUN(\tau(\pi')|_{id})$ are equivalent if $\forall i \in \{1, 2, \ldots\}.\ n_i = n'_i \wedge k_i = k'_i$.

6. Real-time information flow security policies

In the previous section, we provide the general model which can describe the timed behaviors of the set $\Gamma_{\mathcal{A}}$ of concurrently running tasks. In this section, to specify real-time information flow security policies, we restrict (simplify) our assumption in such a way that the set $\Gamma_{\mathcal{A}}$ consists only of three tasks with different security levels. Our restricted assumption is that $\Gamma_{\mathcal{A}} = \{\mathbf{T}_H, \mathbf{T}_L, \mathbf{T}_N\}$, where $\mathbf{T}_H$, $\mathbf{T}_L$, and $\mathbf{T}_N$ denote a high-level task, a low-level task, and a third party task, respectively. A tuple $\mathcal{M}$ of *TKS* is used to model all possible timed behaviors of the set $\Gamma_{\mathcal{A}} = \{\mathbf{T}_H, \mathbf{T}_L, \mathbf{T}_N\}$ of real-time tasks running under a scheduling algorithm $\mathcal{A}$, i.e. $\mathcal{M} = (\mathcal{AP}, \mathcal{S}, \mathcal{S}_{init}, \mathcal{R}, \mathcal{L})$, where, for every state $s \in \mathcal{S}$, $\mathcal{L}(s) = \{P_N \in \mathcal{AP}_N, P_H \in \mathcal{AP}_H, P_L \in \mathcal{AP}_L\}$ and $\mathcal{AP} = \cup_{id \in \{H, L, N\}} \mathcal{AP}_{id}$.

6.1. *Finite Timed Computation Tree (TCT)*

For real-time information flow analysis, which is assumed to have a zero sender–receiver (S–R) period, it is important to trace the sequence of state transitions made during a transmission period (Fig. 3). In our context, the transmission period means the time between the activation and the termination of $\mathbf{T}_L$. We assume that, during the transmission period, $\mathbf{T}_L$ always terminates (in order for *Low* to observe a response time). Thus, in our model, the timed behavior of a real-time system during the transmission period means a collection of finite computation paths π (each path π consists of an interleave of actions

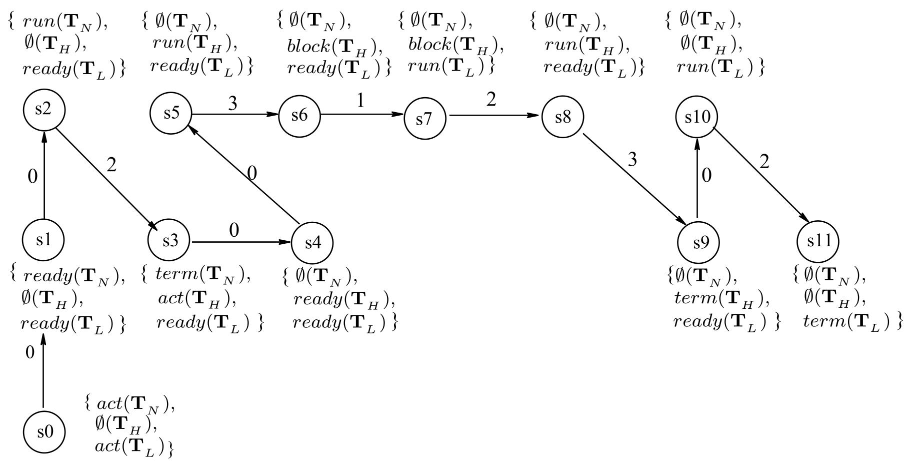
{ run(T_N), ∅(T_H), ready(T_L)}
{ ∅(T_N), run(T_H), ready(T_L)}
{ ∅(T_N), block(T_H), ready(T_L)}
{ ∅(T_N), block(T_H), run(T_L)}
{ ∅(T_N), run(T_H), ready(T_L)}
{ ∅(T_N), ∅(T_H), run(T_L)}
s2
s5
3
s6
1
s7
2
s8
s10
0
2
0
3
0
2
s1
s3
0
s4
s9
s11
{ ready(T_N), ∅(T_H), ready(T_L)}
{ term(T_N), act(T_H), ready(T_L)}
{ ∅(T_N), ready(T_H), ready(T_L)}
{∅(T_N), term(T_H), ready(T_L)}
{ ∅(T_N), ∅(T_H), term(T_L)}
0
s0
{ act(T_N), ∅(T_H), act(T_L)}

Fig. 5 – *TKS* with labels.

and events of the set $\Gamma_{\mathcal{A}}$ of tasks) occurring during the time between $\mathbf{T}_L$'s activation and termination. The computation path π is formed by designating state[6] $s_I \in \mathcal{S}$ as initial and then unwinding *TKS*$\mathcal{M}$ into a finite tree $\omega_{\mathcal{M}}$ with the designated state s_I as the root and a set F of multiple states as leaf nodes, where $act(\mathbf{T}_L) \in \mathcal{L}(s_I)$ and $F = \{s_f | s_f \in \mathcal{S} \wedge term(\mathbf{T}_L) \in \mathcal{L}(s_f)\}$. We call $\omega_{\mathcal{M}}$ the finite Timed Computation Tree (*TCT*) of *TKS*$\mathcal{M}$. $\omega_{\mathcal{M}}$ consists of one or more computation paths. The computation path $\pi = s_I \overset{d_I}{\rightarrow} \cdots s_f$ of $\omega_{\mathcal{M}}$ is denoted by $\pi[s_I \ldots s_f]$, where $act(\mathbf{T}_L) \in \mathcal{L}(s_I)$ and $s_f \in F$. Therefore, the finite Timed Computation Tree $\omega_{\mathcal{M}}$ represents all possible sequences of actions and events of a set $\Gamma_{\mathcal{A}}$ of tasks occurring during the transmission period. The activation time of $\mathbf{T}_L$ is a reference point for the first activation time n_1 of $\mathbf{T}_H$. If $\mathbf{T}_H$ is activated before or at the same time as $\mathbf{T}_L$'s activation time, we use $n_1 = 0$ (the run trace of $\mathbf{T}_H$ starts with $act(\mathbf{T}_H)\downarrow_0$). If $\mathbf{T}_H$ is activated m units of time after $\mathbf{T}_L$'s activation time, $n_1 = m$ (the run trace of $\mathbf{T}_H$ starts with $act(\mathbf{T}_H)\downarrow_m$).

6.2. *Notation*

To specify real-time information flow policies, we use the following notations:

- ρ denotes a finite computation path π from root s_I to a leaf node $s_f \in F$ of $\omega_{\mathcal{M}}$, i.e. $\rho = \pi[s_I \ldots s_f]$. We call ρ a complete path in $\omega_{\mathcal{M}}$. In addition, $PATHS_{\omega_{\mathcal{M}}}$ denotes a set of all possible complete paths in $\omega_{\mathcal{M}}$, i.e. $PATHS_{\omega_{\mathcal{M}}} = \{\rho | \rho = \pi[s_I \ldots s_f],\ act(\mathbf{T}_{id}) \in \mathcal{L}(s_I),\ s_f \in F\}$.
- $\tau(\rho)$ represents the history of a complete path ρ.
- $R_{\omega_{\mathcal{M}}} = \{\mathcal{D}(\tau(\rho)|_L) | \rho \in PATHS_{\omega_{\mathcal{M}}}\}$ is a set of all possible response times of $\mathbf{T}_L$ which *Low* can observe.

We need to define all possible execution behaviors (run traces) of $\mathbf{T}_H$. Additionally, we need a way to express which run trace(s) of $\mathbf{T}_H$ is/are responsible for a particular response time $t \in R_{\omega_{\mathcal{M}}}$.

[6] For simplicity, we assume that *TKS* $\mathcal{M}$ has a single state $S_I \in S$ where $act(\mathbf{T}_L)$ holds; as a result, only one Timed Computation Tree with root S_I is constructed for analysis. If there are multiple states on which $act(\mathbf{T}_L)$ holds, multiple Timed Computation Trees should be constructed for analysis.

Definition 10. The timed behavior set $RT^H_{\omega_{\mathcal{M}}}$ of $\mathbf{T}_H$ is defined as a set which contains all possible run traces of complete paths taken by $\mathbf{T}_H$ in $\omega_{\mathcal{M}}$, i.e. $RT^H_{\omega_{\mathcal{M}}} = \{RUN(\tau(\rho)|_H) | \rho \in PATHS_{\omega_{\mathcal{M}}}\}$. $RT^H_{\omega_{\mathcal{M}}}$ can be thought of as a set of all possible timed behaviors of $\mathbf{T}_H$ which can affect the response time of $\mathbf{T}_L$. The response time specific behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ of $\mathbf{T}_H$, $RT^H_{\omega_{\mathcal{M}}}(t) \subseteq RT^H_{\omega_{\mathcal{M}}}$, is a set of run traces which is responsible for the response time $t \in R_{\omega_{\mathcal{M}}}$ of $\mathbf{T}_L$, i.e. $RT^H_{\omega_{\mathcal{M}}}(t) = \{RUN(\tau(\rho)|_H) | \forall \rho \in PATHS_{\omega_{\mathcal{M}}} \cdot \mathcal{D}(\tau(\rho)|_L) = t\}$.

Example 11. Assume a set $\Gamma_{\mathcal{A}} = \{\mathbf{T}_H, \mathbf{T}_L, \mathbf{T}_N\}$ of tasks concurrently runs on a single processor and the finite Timed Computation Tree $\omega_{\mathcal{M}}$ shown in Fig. 6 represents all possible sequences of actions and events of a set $\Gamma_{\mathcal{A}}$ of tasks occurring during the transmission period of $\mathbf{T}_L$. For better readability, we omit the states with zero duration except initial state s_I and terminal states $s_f \in F$. In addition, the state s_d with the duration d are labeled by only $run(\mathbf{T}_{id})$ to indicate $\mathbf{T}_{id}$ executes for d units of time at s_d. The terminal states s_f are labeled by only $term(\mathbf{T}_L)$ to indicate that $\mathbf{T}_L$ terminates at s_f. The finite Timed Computation Tree $\omega_{\mathcal{M}}$ has the following characteristics:

– s0 as the initial state s_I and a set F of leaf nodes, i.e. $F = \{$s7, s8, s9, s10$\}$.

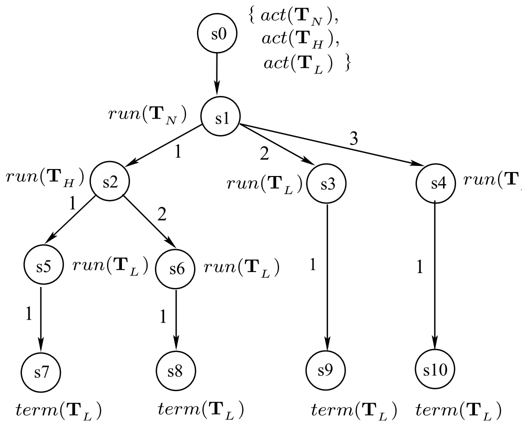

Fig. 6 – Example: a finite Timed Computation Tree.

- $PATHS_{\omega_{\mathcal{M}}} = \{\rho_1 = \pi[s0\ldots s7],\ \rho_2 = \pi[s0\ldots s8],\ \rho_3 = \pi[s0\ldots s9],\ \rho_4 = \pi[s0\ldots s10]\}$.
- Since $\mathcal{D}(\tau(\rho_1|_L)) = \mathcal{D}(\tau(\rho_3|_L)) = 3$ and $\mathcal{D}(\tau(\rho_2|_L)) = \mathcal{D}(\tau(\rho_4|_L)) = 4$, $R_{\omega_{\mathcal{M}}} = \{3, 4\}$.
- In path ρ_1, $\mathbf{T}_H$ is activated at time 0 and executes for 1 unit of time. Thus, $RUN(\tau(\rho_1)|_H) = act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^1$. Similarly, $RUN(\tau(\rho_2)|_H) = act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^2$ and $RUN(\tau(\rho_3)|_H) = RUN(\tau(\rho_4)|_H) = <>_{run}$
- $RT^H_{\omega_{\mathcal{M}}} = \{<>_{run},\ act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^1,\ act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^2\}$
- $RT^H_{\omega_{\mathcal{M}}}(3) = \{<>_{run},\ act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^1\}$ and $RT^H_{\omega_{\mathcal{M}}}(4) = \{<>_{run},\ act(\mathbf{T}_H)\downarrow_0 \cdot run(\mathbf{T}_H)^2\}$.

6.3. *Timing channel free policies*

In this section, we propose four Timing Channel Free (*TCF*) policies. If a finite Timed Computation Tree $\omega_{\mathcal{M}}$ of *TKS* $\mathcal{M}$ has the characteristics which can satisfy the *TCF* policy, we show that the system is free from covert-timing channels (to a certain degree). Formally, we use $\omega_{\mathcal{M}} \| = \mathcal{P}$ to denote that the characteristics of $\omega_{\mathcal{M}}$ satisfies the *TCF* policy $\mathcal{P}$. We provide the formal definitions of the *TCF* policies as follows.

6.3.1. *Weak timing channel free policy*

To satisfy the Weak Timing Channel Free (*WTCF*) policy, for every complete path with a response time $t \in R_{\omega_{\mathcal{M}}}$, which has one or more high-level actions *run*(**T**) enabled, there must exist a complete path with the same response time *t* which does not have any high-level action enabled. The formal definition of the *WTCF* policy is:

Definition 12. Weak Timing Channel Free (WTCF) policy

$$\forall t \in R_{\omega_{\mathcal{M}}} \cdot WTCF\left(RT^H_{\omega_{\mathcal{M}}}(t)\right),\ \text{where}\ WTCF(T) \equiv \exists r \in T \cdot r = <>_{run}$$

Lemma 13. *If* $\omega_{\mathcal{M}} \| = WTCF$*, upon observing a response time* $t \in R_{\omega_{\mathcal{M}}}$*, Low cannot reliably deduce occurrences of any run trace* $r \in RT^H_{\omega_{\mathcal{M}}}(t)$ *of* $\mathbf{T}_H$.

Proof. We assume $\omega_{\mathcal{M}} \| = WTCF$. Using Definition 12, for every response time *t* of $\mathbf{T}_L$, the response time specific timed behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ has at least one element ($<>_{run}$) which does not have any high-level action $run(\mathbf{T}_H)$ enabled. Therefore, *Low* cannot reliably deduce occurrences of any run trace $r \in RT^H_{\omega_{\mathcal{M}}}(t)$ of $\mathbf{T}_H$ since the response time *t* could have been observed without an intervention of timed actions of a high-level task. □

There still exists real-time information flow in a real-time system which satisfies *WTCF* policy. Although *Low* cannot reliably deduce occurrences of a run trace of $\mathbf{T}_H$, when *Low* observes a response time *t*, it can still deduce which run trace of $\mathbf{T}_H$ cannot be responsible for the response time *t*. This type of a covert channel is called a negative channel (Mantel, 2000; McCullough, 1988; Son and Alves-Foss, 2006a). To prevent *Low* from deducing nonoccurrences of run traces of $\mathbf{T}_H$, we need a stronger security policy than *WTCF*.

6.3.2. *Strong timing channel free policy*

We propose the Strong Timing Channel Free (*STCF*) policy to prevent *Low* from deducing not only occurrences, but also nonoccurrences of any run trace of $\mathbf{T}_H$. To prevent *Low* from deducing nonoccurrences of run traces of $\mathbf{T}_H$, for every response time *t*, the response specific behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ must include all run traces that a high-level task can possibly execute. The formal definition of *STCF* policy is:

Definition 14. Strong Timing Channel Free (STCF) policy

$$\forall t \in R_{\omega_{\mathcal{M}}} \cdot STCF\left(RT^H_{\omega_{\mathcal{M}}}(t)\right),\ \text{where}\ STCF(T) \equiv WTCF(T) \wedge NON(T),$$
$$\text{where}\ NON(T) \equiv RT^H_{\omega_{\mathcal{M}}} = T$$

Theorem 15. *If* $\omega_{\mathcal{M}} \| = STCF$*, there is no real-time information flow from High to Low in a real-time system* $\mathcal{M}$.

Proof. Assume $\omega_{\mathcal{M}} \| = STCF$. From Definition 14, $\omega_{\mathcal{M}} \| = WTCF$, which prevents *Low* from reliably deducing occurrences of any run trace *r* of $\mathbf{T}_H$ (Lemma 13). By definition, $\omega_{\mathcal{M}}$ also satisfies *NON*(*T*), where $T = RT^H_{\omega_{\mathcal{M}}}(t)$. *NON*(*T*) requires that the response time specific behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ must be equal to the set of all run traces which $\mathbf{T}_H$ can possibly produce (any measured value of the response time *t* will not rule out any possible run trace of $\mathbf{T}_H$ since $RT^H_{\omega_{\mathcal{M}}}(t) = RT^H_{\omega_{\mathcal{M}}}$ for every *t*). These two predicates together (*WTCF* and *NON*) prevent Low from reliably deducing anything about the run traces of $\mathbf{T}_H$ upon observing a response time. □

Example 16. The finite Timed Computation Tree $\omega_{\mathcal{M}}$ shown in Fig. 6 of Example 11 satisfies the *WTCF* policy because the response time specific timed behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ includes $<>_{run}$ for every response time t. However, the finite computation tree $\omega_{\mathcal{M}}$ does not satisfy the *STCF* policy because $RT^H_{\omega_{\mathcal{M}}}(t) \neq RT^H_{\omega_{\mathcal{M}}}$ for every response time *t*.

6.3.3. *Rigid timing channel free policy*

Definition 17. Rigid Timing Channel Free (RTCF) policy

$$|R_{\omega_{\mathcal{M}}}| = 1 \wedge STCF\left(RT^H_{\omega_{\mathcal{M}}}(t)\right),\ \text{where}\ t \in R_{\omega_{\mathcal{M}}}$$

The difference between *STCF* and *RTCF* is that *RTCF* requires that *Low* should always observe the same response time[7] ($|R_{\omega_{\mathcal{M}}}| = 1$), while *STCF* does not.

The definition of the *RTCF* policy, which is a special case of *STCF*, may be useful to convert a system satisfying only *WTCF* to a system satisfying *RTCF*; it is easier to design and verify a system satisfying *RTCF* than *STCF* (see Section 7.2). In addition, from Shannon's information theoretic view point (Cover and Thomas, 1991; Weaver and Shannon, 1963), the quantity (capacity) of real-time information flow from $\mathbf{T}_H$ to $\mathbf{T}_L$ over a covert channel for an *RTCF* system is zero no matter what probability distribution the channel has since *Low*, as an information receiver, always observes the same value (response time).

[7] Formally, $|R_{\omega_{\mathcal{M}}}| = 1 \Leftrightarrow \forall_{\rho_1} \forall_{\rho_2} \in PATHS_{\omega_{\mathcal{M}}} \cdot \mathcal{D}(\tau(\rho_1)|_L) = \mathcal{D}(\tau(\rho_2)|_L)$.

6.3.4. *Non-preemptive timing channel free policy*

The Non-preemptive Timing Channel Free (*NTCF*) policy prevents the existence of covert-timing channels for both real-time and non-real-time systems. It simply says that a low-level task $\mathbf{T}_L$ must not be preempted by a high-level task $\mathbf{T}_H$. The formal definition of *NTCF* is:

Definition 18. Non-preemption Timing Channel Free (NTCF) policy

$\forall r \in RT^H_{\omega_\mathcal{M}} \cdot r = <>_{run}$

In Appendix, we demonstrate a few examples of how our formal framework can be applied to more realistic problems.

7. Comparing real-time information flow policies

7.1. *Relative strength of real-time information flow policies*

In this section, we compare Timing Channel Free (*TCF*) policies to evaluate the relative strength of each policy. We first define what it means to compare *TCF* policies. Assume we have two *TCF* policies $\mathcal{P}_1$ and $\mathcal{P}_2$. To compare two policies, $\mathcal{P}_1$ and $\mathcal{P}_2$, we evaluate $\mathcal{P}_1 \Rightarrow \mathcal{P}_2$ and $\mathcal{P}_2 \Rightarrow \mathcal{P}_1$. If the first statement is true, $\mathcal{P}_1$ is at least as strong as $\mathcal{P}_2$, and vice versa. If both are true the policies are of equal strength. If neither is true, they are not comparable.

Theorem 19. *The hierarchical strength of Timing Channel Free requirements follows the partial ordering shown in Fig. 7.*

Proof. We first prove $NTCF \Rightarrow STCF$ and $STCF \not\Rightarrow NTCF$: ($NTCF \Rightarrow STCF$) Let $\omega_\mathcal{M}$ be a finite Timed Computation Tree model of $TKS\mathcal{M}_1$. If $\omega_{\mathcal{M}_1} \| = NTCF$, meaning that no $run(\mathbf{T}_H)$ is performed during an execution of $\mathbf{T}_H$, the set $RT^H_{\omega_\mathcal{M}}$ has a single element $<>_{run}$. Since $RT^H_{\omega_\mathcal{M}}(t) \subseteq RT^H_{\omega_\mathcal{M}}$, $RT^H_{\omega_\mathcal{M}}(t) = RT^H_{\omega_\mathcal{M}} = \{<>_{run}\}$ for all response times $t \in R_{\omega_\mathcal{M}}$; this condition satisfies the predicate *NON* of *STCF*. In addition, for every response time t, the response specific behavior set $RT^H_{\omega_\mathcal{M}}(t)$ contains the element $<>_{run}$, which satisfies the predicate *WTCF*. Therefore, $\omega_{\mathcal{M}_1} \| = NTCF$ implies that $\omega_{\mathcal{M}_1}$ satisfies both predicates *WTCF* and *NON* of *STCF*.

($STCF \not\Rightarrow NTCF$) Let $\omega_{\mathcal{M}_2}$ be a finite Timed Computation Tree model of $TKS\mathcal{M}_2$. If $\omega_{\mathcal{M}_2} \| = STCF$, any run trace $r \in RT^H_{\omega_\mathcal{M}}$ of $\mathbf{T}_H$ could have the running action $run(\mathbf{T}_H)$ enabled in it (meaning that $\mathbf{T}_H$ preempts $\mathbf{T}_L$ while $\mathbf{T}_L$ is running). This violates the *NTCF* policy.

Next, we prove $NTCF \not\Leftrightarrow RTCF$: while *NTCF* requires that $\mathbf{T}_L$ should not be preempted by $\mathbf{T}_H$, a system satisfying *RTCF* may allow the preemption. Thus, $RTCF \not\Rightarrow NTCF$. While *RTCF* requires that *Low* should only observe a single response time, a system satisfying *NTCF* may produce multiple response times (due to interference caused by timed action of $\mathbf{T}_N$) observable by *Low*. Thus, $NTCF \not\Rightarrow RTCF$.

We do not include the proofs showing the hierarchical strength of Timing Channel Free policies between *WTCF*, *STCF*, and *RTCF* since the careful observation of each definition can easily reveal the relative strength; the definition of *RTCF* is a more restrictive form of *STCF* and the definition of *STCF* includes one of *WTCF*. □

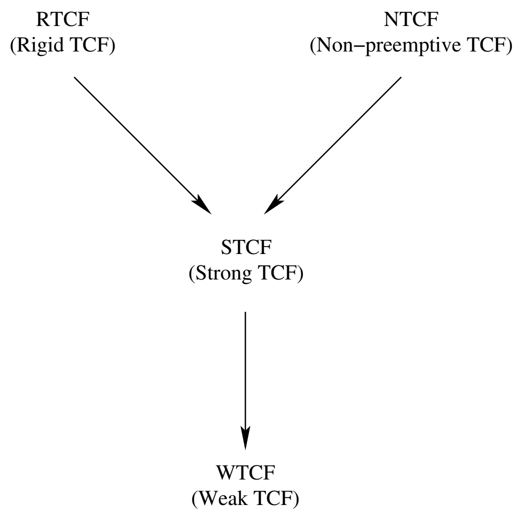

Fig. 7 – Partial ordering of Timing Channel Free requirements.

7.2. *Specifying security policies*

The Alpern–Schneider's framework (Alpern and Schneider, 1985) has been the dominant model in the specification of programs: a system is identified with the set $\prod$ of possible execution sequences π (an execution sequence π can be thought of as a history of a single computation path in TKS) that a system can produce and a property is regarded as the set of execution sequences. A property is often called a property of traces (Mantel, 2000). A system satisfies a property of traces if the system's set of execution sequences is a subset of the property's set. A set of execution sequences is called a *property* of traces if set membership is solely determined by each element and not by other members of the set. This implies that a security policy or a requirement $\mathcal{P}$ is a property if the security policy or the requirement $\mathcal{P}$ can be specified by a predicate $\widehat{\mathcal{P}}$ of the form (Schneider, 2000):

$$\mathcal{P}\left(\prod\right) \equiv \forall \pi \in \prod \cdot \widehat{\mathcal{P}}(\pi) \tag{1}$$

The above equation (1) reads as: the security policy or the requirement $\mathcal{P}$ is a property of traces if every execution sequence π in the set $\prod$ satisfies the predicate $\widehat{\mathcal{P}}$ (form of the security policy) on an individual execution sequence. From the equation (1), it is obvious that, given $\mathcal{P}(\prod)$, $\mathcal{P}(\prod')$ must also hold for any subset $\prod'$ of $\prod$.

Lemma 20. *The NTCF policy is a property of traces.*

Proof. Let $\omega_\mathcal{M}$ satisfy *NTCF* and π be an individual execution sequence (computation path), i.e. $\pi \in PATHS_{\omega_\mathcal{M}}$. To prove *NTCF* is a property of traces, we must demonstrate that we can construct a predicate $\widehat{\mathcal{P}}$ defined on a single execution sequence π such that $\forall \pi \in PATHS_{\omega_\mathcal{M}} \cdot \widehat{\mathcal{P}}(\pi)$.

The *NTCF* policy specifies that a low-level task $\mathbf{T}_L$ must not be preempted by a high-level task $\mathbf{T}_H$ during the transmission period (the time between $\mathbf{T}_L$' activation and termination).

Thus, the predicate $\widehat{\mathcal{P}}$ can be constructed to establish if any individual execution path π should not include any high-level action *run*($\mathbf{T}_H$) enabled. Therefore, *NTCF* is a property of traces. □

In general, an information flow security policy is not a property of traces unless we restrict its definition. It is generally known that information flow security policies are expressed as a closure condition on a trace set $\prod$ of possible execution sequences π (Alur et al., 2006; McLean, 1996). A closure condition is often called a property of a trace set. For example, the simplest form of a closure condition on a trace set can be expressed as: $\pi \in \prod \Rightarrow f(\pi) \in \prod$ for some function *f* which returns an execution sequence for an input π. This implies the following: given an information flow security policy $\mathcal{P}$, two sets of executions $\prod$ and $\prod'$, and $\prod' \subset \prod$, $\mathcal{P}(\prod) \nRightarrow \mathcal{P}\prod'$.

WTCF, *STCF* and *RTCF* policies have the following characteristics:

Lemma 21. *The WTCF, STCF and RTCF policies are not properties of traces; each policy is a property of a trace set.*

Proof. We use a proof by contradiction to show that the *WTCF*, *STCF* and *RTCF* policies are not properties of traces. Let us assume that *WTCF* is a property $\mathcal{P}$ of traces and a *TCT* model $\omega_{\mathcal{M}}$ satisfies $\mathcal{P}$. Thus, the following must hold: $\mathcal{P}(PATHS_{\omega_{\mathcal{M}}}) \equiv \forall \pi \in PATHS_{\omega_{\mathcal{M}}} \cdot \widehat{\mathcal{P}}(\pi)$. Further, for any subset $\prod'$ of $PATHS_{\omega_{\mathcal{M}}}$, $\mathcal{P}(\prod')$ must also hold. Specifically, let $\prod'$ be a set with a single execution path in which a high-level action *run*($\mathbf{T}_H$) is enabled. This set $\prod'$ cannot satisfies *WTCF* since *WTCF* requires another execution path with the same response time in which no high-level action is enabled. This clearly shows that *WTCF* is not a property of traces but a property of a trace set. The same reasoning can be applied to *STCF* and *RTCF*. □

The big difference between *NTCF* and other policies is that only *NTCF* demands that a low-level task $\mathbf{T}_L$ should not be preempted by a high-level task $\mathbf{T}_H$. This difference and the previous two lemmas lead to the following corollary.

Corollary 22. *To define a real-time information flow security policy for a system, where a low-level task must be preempted by a high-level task to meet the timing constraints, the policy must be specified by a property of a trace set.*

Proof. A preemption of $\mathbf{T}_L$ by $\mathbf{T}_H$, while $\mathbf{T}_L$ is running, always affects (delays) an execution of $\mathbf{T}_L$. This preemption may cause possible information leakage through a covert-timing channel. For a computation path in which the preemption of $\mathbf{T}_L$ by $\mathbf{T}_H$ occurs, there must exist at least one (or more) different computation paths which can obfuscate *Low*'s view on timed actions taken by *High*. Therefore, the policy must be specified by the property of a trace set. □

8. Discussion and conclusion

Our view of real-time systems is that multi-level secure real-time tasks are scheduled and running, according to a scheduling algorithm employed by the systems. From this perspective, we develop a general trace-based framework in which one can carry out a covert-timing channel analysis of a real-time system. In addition, we provide the formal definitions of the Timing Channel Free (*TCF*) policies; if a system satisfies one of our proposed *TCF* security policies, we have demonstrated that the system can achieve a certain level of real-time information flow security. Finally, we compare the relative strength of the proposed *TCF* policies and prove that all *TCF* policies are not properties of traces except *NTCF*.

Our formal framework can be used to model the timed behaviors of MLS real-time tasks and check if the system model satisfies the TCF requirements. When the timed behaviors of real-time tasks do not satisfy any of the TCF polices, an attacker can easily construct a covert-timing channel from a high-level to a low-level task and leak classified information through the timing channel. Some defensive measures must be applied to remove or mitigate the impact of covert-timing channels. The defensive measures typically require modification to an insecure real-time scheduler or scheduling algorithm. For example, one popular defensive measure is to inject a dummy task into the system having timing vulnerabilities to introduce noise into a covert-timing channel. This addition of the dummy task and its influence on timed behaviors of other tasks can be easily incorporated into an existing model via our formal framework. In addition, system engineers can use the formal definition of the TCF policies as a guideline to design and implement a covert-timing channel free scheduler or scheduling algorithm.

Directions for future research are twofold: The first is to investigate if a specific real-time scheduling algorithm such as Rate Monotonic scheduling or Earliest Deadline First scheduling (Liu and Layland, 1973; Liu, 2000) has any timing vulnerabilities (Son and Alves-Foss, 2006a,b). We believe that some real-time scheduling algorithms are more vulnerable to timing attacks than other. The second is to research how to relax the definitions of our *TCF* policies to make them more practical for checking timing vulnerabilities of hard-real-time systems. In hard-real-time systems, system requirements such as timing constraints cannot be compromised by security requirements if a system is to be usable; satisfying timing constraints is the top priority for hard-real-time systems. We should rewrite and relax the security policies if they prevent a system from achieving system requirements; therefore, it is necessary to devise a weaker security policy than our *TCF* which allows some control of real-time information flow without permitting too much.

Acknowledgements

This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expresses or implied, of AFLR and DARPA or the U.S. Government.

Appendix.

In this Appendix, we show a few examples of how our formal framework can be applied to more realistic problems (a set of tasks scheduled by a well-known scheduling algorithm). The notations and terminologies used to describe the well-known scheduling algorithm are as follows:

- A (periodic) task $\mathbf{T}_i$ has the individual jobs and we denote them $J_{i,1}$, $J_{i,2}$, and so on, $J_{i,k}$ being the kth job in $\mathbf{T}_i$.
- The activation (arrival) time of the first job $J_{i,1}$ in each task $\mathbf{T}_i$ is called the phase Φ_i of $\mathbf{T}_i$. The relative deadline D_i of $\mathbf{T}_i$ is the length of unit time from activation by which every job of $\mathbf{T}_i$ must finish.
- A periodic task $\mathbf{T}_i$ with phase Φ_i, period T_i, the maximum (worst-case) execution time C_i, and relative deadline D_i is characterized as the 4-tuple, i.e. $\mathbf{T}_i(\Phi_i, T_i, C_i, D_i)$. The activation time of the periodic task $\mathbf{T}_i$ is $\Phi_i+(k-1)T_i$, $k=0, 1, \ldots$. The task $\mathbf{T}_i$ can be simply characterized as $\mathbf{T}_i(T_i, C_i)$ if $\Phi_i=0$ and $T_i=D_i$ ($\mathbf{T}_i$ has the deadline equal to its period).
- A task is aperiodic if the jobs in it has no deadline and the activation time of the jobs are not known a priori.

In the following example, we assume that the Rate Monotonic (RM) scheduling algorithm (14) is employed to schedule a set of independent periodic tasks. The RM scheduling algorithm schedules a set of independent periodic tasks according to the following rules :

- Tasks with shorter periods (higher request rates) will have higher priorities.
- A priority assigned to a task is fixed.
- A currently executing task is preempted by a newly arrived task with a higher priority (shorter period).

Example 23. Assume a set Γ_{RM} of three independent periodic tasks are concurrently running under RM scheduling in a single processor real-time system, i.e. $\Gamma_{RM}=\{\mathbf{T}_N(20,3), \mathbf{T}_H(10,1), \mathbf{T}_L(5,2)\}$. Since the low-level task $\mathbf{T}_L$ has the shorted period 5, the highest scheduling priority is assigned to $\mathbf{T}_L$ and $\mathbf{T}_L$ is never preempted during its execution. Thus, the finite Timed Computation Tree constructed from this system satisfies Non-preemption Timing Channel Free (NTCF) policy. If tasks are scheduled by a fixed-priority preemptive algorithm such as RM, the covert-timing channel can be eliminated by assigning a higher scheduling priority to a lower security class task than a higher security class task. Obviously, this covert-timing channel free scheduling policy has performance disadvantages, e.g. increased response time for a higher security class task.

Many real-time systems require an integrated approach suitable for scheduling hard deadline periodic tasks along with aperiodic tasks with no firm deadline. In the next example, a scheduling algorithm called the Polling Server (PS) is used to service aperiodic tasks while *RM* is employed to schedule periodic tasks. The PS algorithm is based upon the following approach. When an aperiodic task arrives, it is placed in an aperiodic job queue, waiting for execution. A periodic task called a polling server is created to service aperiodic requests. Like any periodic task, the periodic server task $\mathbf{T}_s$ is characterized by the polling period T_s and the maximum computation time C_s, i.e. $\mathbf{T}_s(T_s, C_s)$. In the PS algorithm, the parameter *Cap* called the server capacity is monitored to make sure that $\mathbf{T}_s$ cannot execute an aperiodic task more than C_s units. At the beginning of each period T_s, *Cap* is

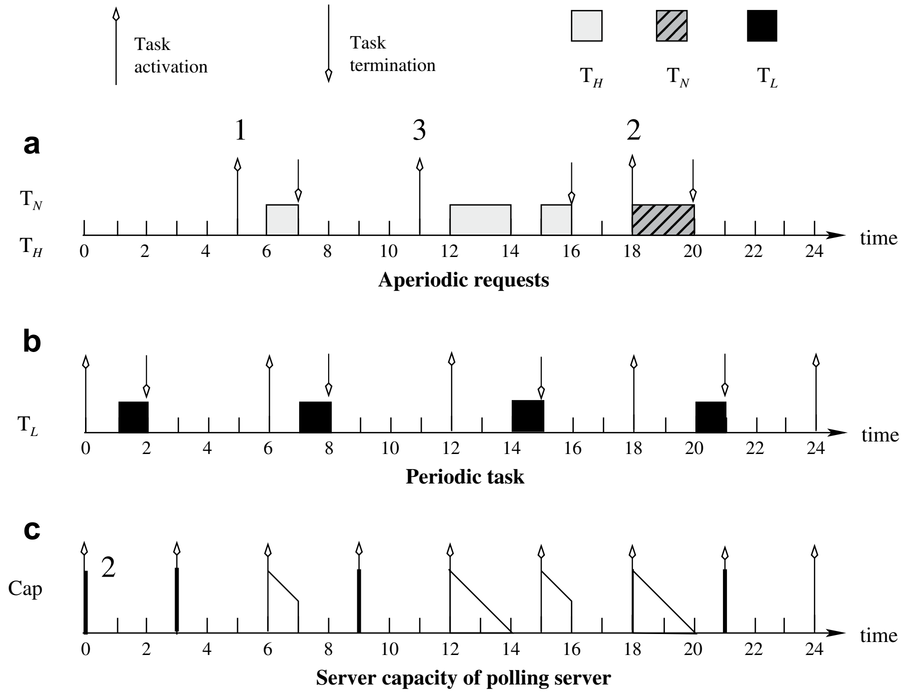

Fig. 8 – Timing diagram.

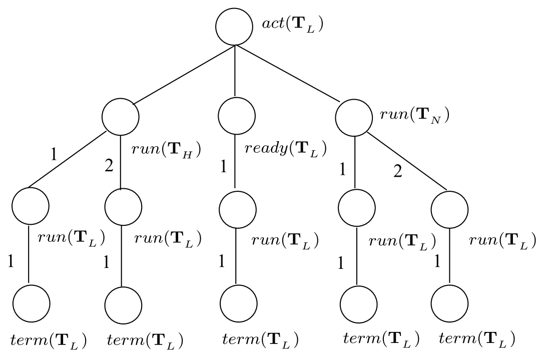

Fig. 9 – Example: finite Timed Computation Tree.

set to C_s. In addition, the aperiodic job queue is examined and the following actions are taken:

- If the queue is found not empty, the polling server executes the aperiodic job(s) until there is no job left to execute in the queue or it executes for C_s units of time, whichever occurs sooner. When the server executes the aperiodic jobs in the queue, it consumes its *Cap* at the rate of one per unit time. When the queue becomes empty or the polling server has consumed all of its server capacity (*Cap* becomes zero), it is immediately suspended and wait for the next polling period for execution.
- If the queue is found empty, the polling server suspends immediately. The polling server will not be ready for execution and is not able to examine the queue again until the next polling period. An aperiodic task which arrives after the aperiodic job queue is examined and found empty must wait for the next polling period to be serviced.

In general, the polling server is scheduled with the same algorithm used for the periodic task, and, once active, it serves the aperiodic requests with the limit of its server capacity.

Example 24. Assume a set $\Gamma_{polling}$ of three independent tasks are concurrently running in a single processor real-time system; two aperiodic tasks, $\mathbf{T}_N$ and $\mathbf{T}_H$, with a finite execution time $e(1 \le e \le N,\ N = 1,2,\ldots)$ and a periodic task $\mathbf{T}_L(6,1)$, i.e. $\Gamma_{polling} = \{\mathbf{T}_N, \mathbf{T}_H, \mathbf{T}_L\}$. The polling server $\mathbf{T}_s(3, 2)$ is created to service both aperiodic tasks. The polling server and the periodic task $\mathbf{T}_L$ are scheduled by the RM scheduling algorithm. Since $\mathbf{T}_s$ has the shorter period than $\mathbf{T}_L$, $\mathbf{T}_s$ has the higher scheduling priority. In addition, we assume that the periodic task $\mathbf{T}_L$ always has the constant delay of one unit of time between activation and release. Thus, after $\mathbf{T}_L$'s activation, one unit of time elapses before the periodic task $\mathbf{T}_L$ is released (starts to execute) if the aperiodic tasks are not ready to run. $\mathbf{T}_L$ is delayed more than one unit of time if the aperiodic tasks are ready to run at the time of its release. Fig. 8 illustrates a few instances of RM-based execution of the polling server and the periodic task, while aperiodic requests are serviced by the polling server. The finite Timed Computation Tree $\omega_{\mathcal{M}}$ in Fig. 9 shows all the possible timed behaviors of the set $\Gamma_{polling}$ of tasks during the transmission period. For better readability, we omit the states with zero duration except the initial state (where $act(\mathbf{T}_L)$ holds) and the terminal states. In addition, the state s_d with the duration d are labeled by only important atomic propositions (either $run(\mathbf{T}_{id})$ or $ready(\mathbf{T}_{id})$). The terminal states s_f are labeled by only $term(\mathbf{T}_L)$ to indicate that $\mathbf{T}_L$ terminates at s_f. Since the response time specific timed behavior set $RT^H_{\omega_{\mathcal{M}}}(t)$ includes $<>_{run}$ for every response time $t \in R_{\omega_{\mathcal{M}}} = \{2, 3\}$, the finite Timed Computation Tree $\omega_{\mathcal{M}}$ satisfies the WTCF policy. However, $\omega_{\mathcal{M}}$ does not satisfy the STCF policy because $RT^H_{\omega_{\mathcal{M}}}(t) \neq RT^H_{\omega_{\mathcal{M}}}$ for every response time t.

REFERENCES

John Agat. Transforming out timing leaks in practice. In: Proc. 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages; 2000, p. 40–53.

Alpern B, Schneider F. Defining liveness. Information Processing Letters 1985;21(4):181–5.

Alur R, Courcoubetis C, Dill D. Model-checking in dense real-time information and computation. Software Engineering Journal 1993;104(1):2–34.

Alur R, Černý P, Zdancewic S. Preserving secrecy under refinement. In: 33rd international colloquium on automata, languages and programming, ICALP, July 2006.

Barbuti R, Tesei L. Decidable notion of timed non-interference. Concurrency Specification and Programming 2003;54(2–3): 137–50.

Cabuk S, Brodley C, Shields C. IP covert timing channels: design and detection. In: Proc. ACM conference on computer and communications security; 2004.

Cover T, Thomas J. Elements of information theory. Wiley-Interscience; 1991.

Fidge C, Zic J. A simple, expressive real-time CCS. In: Proc. 2nd Australasian conf. on parallel & real-time systems; 1995, p. 365–372.

Focardi R, Gorrieri R, Martinelli F. Real-time information flow analysis. IEEE Journal on Selected Areas in Communications 2003;21(1).

Gray J. On introducing noise into the bus-contentiion channel. IEEE Symposium on Research in Security and Privacy 1993.

Hu W. Reducing timing channels with fuzzy time. IEEE Symposium on Research in Security and Privacy 1991.

Janeri J, Darby D, Schnackenberg D. Building higher resolution synthetic clocks for signaling in covert timing channels. In: The eighth IEEE computer security foundations workshop; 1995.

Laroussinie F, Markey N, Schnoebelen Ph. On model checking durational Kripke structures. In: International Conference on Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 2303; 2002. p. 264–79.

Laroussinie F, Markey N, Schnoebelen Ph. Efficient timed model checking for discrete-time systems. Theoretical Computer Science in Lecture Notes in Computer Science 2006;353(1): 249–71.

Liu C, Layland J. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM 1973;20(1).

Liu J. Real-time systems. Prentice Hall; 2000.

Logothetis G, Schneider K. A new approach to the specification and verification of real-time systems. In: Euromicro conference on real-time systems; 2001a, p. 171–180.

Logothetis G, Schneider K. Symbolic model checking of real-time systems. In: Eighth international symposium on temporal representation and reasoning; 2001b, p. 214–223.

Lowe Gavin. Defining information flow quantity. Journal of Computer Security 2004;12(3–4):619–53.

Mantel H. Possibilistic definitions of security – an assembly kit. In: 13th IEEE computer security foudnations workshop; 2000, p. 235–243.

McCullough Daryl. Foundations of ulysses: the theory of security. Rome Air Development Center; 1988. Technical Report RADC-TR-87–222.

McLean J. A general theory of composition for a class of "possibilistic" properties. IEEE Transaction on Software Engineering 1996;22(1):53–67.

Moskowitz I, Kang MH. Covert channels – here to stay. In Proc. COMPASS 94; 1994, p. 235–243.

Schneider FB. Enforceable security policies. ACM Transactions on Information and Systems Security 2000;3(1):30–50.

Shah G, Molina A, Blaze M. Keyboards and covert channels. In: Proc. 15th USENIX security symposium; 2006, p. 59–75.

Son J, Alves-Foss J. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in MLS systems. In: Proc. IEEE workshop on information assurance; 2006a, p. 361–368.

Son J, Alves-Foss J. Covert timing channel capacity of rate monotonic scheduling algorithm in MLS systems. In: The IASTED international conference on communication, network, and information security; 2006b.

Weaver W, Shannon CE. The mathematical theory of communication. University of Illinois Press; 1963.

Zakinthinos A, Lee ES. A general theory of security properties. Proc. of the IEEE Symposium on Security and Privacy 1997: 94–102.

Joon Son is a senior professional staff at Johns Hopkins University Applied Physics Lab. He received his Ph.D in Computer Science from the University of Idaho in 2008. His research interests include Multi-Level Security, formal methods, security policy refinement and network security.

Dr. Jim Alves-Foss is a professor of computer science and is the Director of the Center for Secure and Dependable Systems (est. 1998) at the University of Idaho. He received his PhD in Computer Science from the University of California at Davis in 1991. His primary research interests focus on the design, implementation and verification of high assurance computing systems.