Scheduler-Specific Confidentiality for Multi-Threaded Programs and Its Logic-Based Verification

Marieke Huisman and Tri Minh Ngo

University of Twente, Netherlands
Marieke.Huisman@ewi.utwente.nl
tringominh@gmail.com

Abstract. Observational determinism has been proposed in the literature as a way to ensure confidentiality for multi-threaded programs. Intuitively, a program is observationally deterministic if the behavior of the public variables is deterministic, i.e., independent of the private variables and the scheduling policy. Several formal definitions of observational determinism exist, but all of them have shortcomings; for example they accept insecure programs or they reject too many innocuous programs. Besides, the role of schedulers[1] was ignored in all the proposed definitions. A program that is secure under one kind of scheduler might not be secure when executed with a different scheduler. The existing definitions do not always ensure that an accepted program behaves securely under the scheduler that is used to execute the program.
Therefore, this paper proposes a new formalization of scheduler-specific observational determinism. It accepts programs that are secure when executed under a specific scheduler. Moreover, it is less restrictive on harmless programs under a particular scheduling policy. We discuss the properties of our definition and argue why it better approximates the intuitive understanding of observational determinism. In addition, we also discuss how compliance with our definition can be verified, using model-checking.

1 Introduction

The success of applications, such as e.g., Internet banking and mobile code, depends for a large part on the kind of confidentiality guarantees that can be given to clients. Using formal means to establish confidentiality properties of such applications is a promising approach. Of course, there are many challenges related to this. Many systems for which confidentiality is important are implemented in a multi-threaded fashion. Thus, the outcome of such programs depends on the scheduling policy. Moreover, because of the interactions between threads and the exchange of intermediate results, also intermediate states can be observed. Therefore, to guarantee confidentiality for multi-threaded programs, one should

[1] Scheduler implements the scheduling policy.

consider the whole execution traces, i.e., the sequences of states that occur during program execution.

In the literature, different intuitive definitions of confidentiality are proposed for multi-threaded programs. We follow the approach advocated by Roscoe [10] that the behavior that can be observed by an attacker should be deterministic. To capture this formally, the notion of *observational determinism* has been introduced. Intuitively, observational determinism expresses that a multi-threaded program is secure when its *publicly observable traces* are independent of its confidential data, and independent of the scheduling policy [14]. Several formal definitions are proposed [14, 6, 13], but none of them capture exactly this intuition.

The first formal definition of observational determinism was proposed by Zdancewic and Myers [14]. It states that a program is observationally deterministic iff given any two initial stores s_1 and s_2 that are indistinguishable *w.r.t.* the low variables[2], any two low location traces are equivalent upto stuttering and prefixing, where a low location trace is the projection of a trace into a single low variable location. Zdancewic and Myers consider the traces of each low variable separately. They also argue that prefixing is a sufficiently strong relation, as this only causes external termination leaks of one bit of information [14]. In 2006, Huisman, Worah and Sunesen showed that allowing prefixing of low location traces can reveal more secret information — instead of just one bit of information — even for sequential programs. They strengthened the definition of observational determinism by requiring that low location traces must be stuttering-equivalent [6]. In 2008, Terauchi showed that an attacker can observe the relative order of two updates of the low variables in traces, and derive secret information from this [13]. Therefore, he proposed another variant of observational determinism, requiring that all low store traces — which are the projection of traces into a store containing only all low variables — should be equivalent upto stuttering and prefixing, thus not considering the variables independently.

However, Terauchi's definition is also not satisfactory. First of all, the definition still allows an accepted program to reveal secret information, and second, it rejects too many innocuous programs because it requires the complete low store to evolve in a deterministic way.

In addition, the fact that a program is secure under a particular scheduler does not imply that it is secure under another scheduler. All definitions of observational determinism proposed so far implicitly assume a *nondeterministic* scheduler, and might accept programs that are not secure when executed with a different scheduler. Therefore, in this paper, we propose a definition of scheduler-specific observational determinism that overcomes these shortcomings. This definition accepts only secure programs and rejects fewer secure programs under a particular scheduling policy. It essentially combines the previous definitions: it requires that for any low variable, the low location traces from initial stores s_1

[2] For simplicity, we consider a simple two-point security lattice, where the data is divided into two disjoint subsets H and L, containing the variables with high (private) and low (public) security level, respectively.

and s_2 are stuttering-equivalent. However, it also requires that for any low store trace starting in s_1, there *exists* a stuttering-equivalent low store trace starting in s_2. Thus, any difference in the relative order of updates is coincidental, and no information can be deduced from it. This existential condition strongly depends on the scheduler used when the program is actually deployed, because traces model possible runs of a program under that scheduling policy. In addition, we also discuss the properties of our formalization. Based on the properties, we argue that our definition better approximates the intuitive understanding of observational determinism, which unfortunately cannot be formalized directly.

Of course, we also need a way to verify adherence to our new definition. A common way to do this for information flow properties is to use a type system. However, such a type-based approach is insensitive to control flow, and rejects many secure programs. Therefore, recently, self-composition has been advocated as a way to transform the verification of information-flow properties into a standard program verification problem [3, 1]. We exploit this idea in a similar way as in our earlier work [6, 4], and translate the verification problem into a model-checking problem over a model that executes the program to be verified twice, in parallel with itself. We show that our definition can be characterized by a conjunction of an LTL [7] and a CTL [7] formula. For both logics, good model checkers exist that we can use to verify the information flow property. The characterization is done in two steps: first, we characterize stuttering-equivalence, and prove correctness of this characterization, and second, we use this to characterize our definition of observational determinism.

The rest of this paper is organized as follows. After the preliminaries in Section 2, Section 3 formally discusses the existing definitions of observational determinism and their shortcomings. Section 4 gives our new formal definition of scheduler-specific observational determinism, and discusses its properties. The two following sections discuss verification of this new definition. Finally, Section 7 draws conclusions, and discusses related and future work.

2 Preliminaries

This section presents the formal background for this paper. It describes syntax and semantics of a simple programming language, and formally defines equivalence upto stuttering and prefixing.

2.1 Programs and Traces

We present a simple while-language, extended with parallel composition $\|$, i.e., $C\|C'$ where C and C' are two *threads* which can contain other parallel compositions. A thread is a sequence of commands that can be scheduled by a scheduler. The program syntax is not used in subsequent definitions, but we need it to formulate our examples. Programs are defined as follows, where v denotes a variable, E a side-effect-free expression involving numbers, variables and binary

operators, b a Boolean expression, and ϵ the empty (terminated) program.

$$C ::= \texttt{skip} \mid v := E \mid C; C \mid \texttt{while}\,(b)\,\texttt{do}\ C \mid \\ \texttt{if}\ (b)\ \texttt{then}\ C\ \texttt{else}\ C \mid C \| C \mid \epsilon$$

Parallel programs communicate via shared variables in a global store. For simplicity, we assume that all variables initially are set to 0. Further, we also assume an interleaving semantics with the only restriction that two variable accesses cannot occur simultaneously. We do not consider procedure calls, local memory or locks. These could be added to the language but this would not essentially change the technical results.

Let *Conf*, *Com*, and *Store* denote the sets of *configurations*, *programs*, and *stores*, respectively. A configuration $c = \langle C, s\rangle \in \mathit{Conf}$ consists of a program $C \in \mathit{Com}$ and a store $s \in \mathit{Store}$, where C denotes the program that remains to be executed and s denotes the current program store. A store is the current state of the program memory, which is a map from program variables to values. Let L be a set of low variables. Given a store s, we use $s_{\,|_L}$ to denote the restriction of the store where only the variables in L are defined. We say stores s_1 and s_2 are *low-equivalent*, denoted $s_1 =_L s_2$, iff $s_{1\,|_L} = s_{2\,|_L}$, i.e., the values of all variables in L in s_1 and s_2 are the same.

The small-step operational semantics of our program language is standard. Individual transitions of the operational semantics are assumed to be atomic. As an example, we have the following rules for parallel composition (with their usual counterparts for C_2):

$$\frac{\langle C_1, s_1\rangle \to \langle \epsilon, s_1'\rangle}{\langle C_1 \parallel C_2, s_1\rangle \to \langle C_2, s_1'\rangle} \qquad \frac{\langle C_1, s_1\rangle \to \langle C_1', s_1'\rangle \quad C_1' \neq \epsilon}{\langle C_1 \parallel C_2, s_1\rangle \to \langle C_1' \parallel C_2, s_1'\rangle}$$

We also have a special transition step for terminated programs, i.e., $\langle \epsilon, s\rangle \to \langle \epsilon, s\rangle$, ensuring that all traces are infinite. Thus, we assume that the attacker cannot detect termination.

A multi-threaded program executes threads from the set of not-yet terminated threads. During the execution, a scheduling policy repeatedly decides which thread is picked to proceed next with the computation. Different scheduling policies differ in how they make this decision, e.g., a nondeterministic scheduler chooses threads randomly and hence all possible interleavings of threads are potentially enabled; while a *round-robin* scheduler assigns equal time slices to each thread in circular order.

To cover various kinds of schedulers, our formal definition of schedulers assumes that schedulers are history-dependent, i.e., in addition to the current system configuration, the scheduler's behavior also depends on the trace leading to that configuration. Given initial configuration $\langle C, s\rangle$, an infinite list of configurations $T = c_0 c_1 c_2 ...$ ($T : \mathbb{N}_0 \to \mathit{Conf}$) is a *trace* of the execution of C from s, denoted $\langle C, s\rangle \Downarrow T$, iff $c_0 = \langle C, s\rangle$ and $\forall i \in \mathbb{N}_0.\ c_i \to c_{i+1}$. Let $\mathit{Trace}^*\langle C, s\rangle$ denote the set of all finite prefixes of traces that result from the executions of C from s, i.e., $\mathit{Trace}^*\langle C, s\rangle = \{\pi \mid \pi \sqsubseteq T.\ \langle C, s\rangle \Downarrow T\}$ where $\sqsubseteq$ denotes the prefix

relation on traces. Given a finite trace $\pi \in \mathit{Trace}^*\langle C, s\rangle$, a scheduler δ which determines a set of next possible configurations Q, $Q \subseteq \mathit{Conf}$, is formally defined as follows.

Definition 1. *A* scheduler δ *for* C *starting in* s *is a function* $\delta : \mathit{Trace}^*\langle C, s\rangle \to 2^{\mathit{Conf}}$, *such that for all finite traces* $\pi \in \mathit{Trace}^*\langle C, s\rangle$, *if* $\delta(\pi) = Q$ *then* $\mathit{last}(\pi)$ *can take a transition to any* c *belonging to* Q. *Given* π, *we write* $c_0 \to_\delta c_1 \to_\delta c_2 \ldots \to_\delta c_n$ *if* $c_i \in \delta(c_0 \ldots c_{i-1})$ *for all* $1 \leq i < |\pi|$.

This model of schedulers is general enough to describe any scheduler that uses the full history of computation to pick the threads. Given scheduling policy δ, and configuration $\langle C, s\rangle$, a trace of the execution of C from s under the control of δ is denoted as $\langle C, s\rangle \Downarrow_\delta T$. We simply write $\langle C, s\rangle \Downarrow T$ when the scheduler is nondeterministic.

Let T_i, for $i \in \mathbb{N}$, denote the i^{th} element in the trace, i.e., $T_i = c_i$. We use $T_{\ll i}$ to denote the *prefix* of T upto the index i, i.e., $T_{\ll i} = T_0 T_1 \ldots, T_i$. When appropriate, $T_{\ll i}$ can be considered as an infinite trace stuttering in T_i forever. Further, we use $T_{|_L}$ to denote the projection of a trace to a store containing only the variables in L. Formally: $T_{|_L} = \mathit{map}(\_{|_L} \circ \mathit{store})(T)$, where *map* is the standard higher-order function that applies $(\_{|_L} \circ \mathit{store})$ to all elements in T. When L is a singleton set $\{l\}$, we simply write $T_{|_l}$. Finally, in the examples below, when writing an infinite trace that stutters forever from state T_i onwards, we just write this as a finite trace $T = [T_0, T_1, \ldots, T_{i-1}, T_i]$.

2.2 Stuttering and Prefixing Equivalences

The key ingredient in the various definitions of observational determinism is trace equivalence upto stuttering or upto stuttering and prefixing (based on [9, 6]). It uses the auxiliary notion of *stuttering-equivalence upto indexes* i *and* j.

Definition 2 (Stuttering-equivalence). *Traces* T *and* T' *are* stuttering-equivalent upto i and j, *written* $T \sim_{i,j} T'$, *iff we can partition* $T_{\ll i}$ *and* ${T'}_{\ll j}$ *into* n *blocks such that elements in the* p^{th} *block of* $T_{\ll i}$ *are equal to each other and also equal to elements in the* p^{th} *block of* ${T'}_{\ll j}$ *(for all* $p \leq n$*). Corresponding blocks may have different lengths.*

Formally, $T \sim_{i,j} T'$ *iff there are sequences* $0 = k_0 < k_1 < k_2 < \ldots < k_n = i + 1$ *and* $0 = g_0 < g_1 < g_2 < \ldots < g_n = j + 1$ *such that for each* $0 \leq p < n$ *holds:* $T_{k_p} = T_{k_p+1} = \cdots = T_{k_{p+1}-1} = T'_{g_p} = T'_{g_p+1} = \cdots = T'_{g_{p+1}-1}$.

T *and* T' *are* stuttering-equivalent, *denoted* $T \sim T'$, *iff* $\forall i.\ \exists j.\ T \sim_{i,j} T' \wedge \forall j.\ \exists i.\ T \sim_{i,j} T'$.

Stuttering-equivalence defines an equivalence relation, i.e., it is reflexive, symmetric and transitive.

We say that T and T' are equivalent upto stuttering and prefixing, written $T \sim_p T'$, iff T is stuttering-equivalent to a prefix of T' or vice versa, i.e., $\exists i.\, T \sim {T'}_{\ll i} \vee T_{\ll i} \sim T'$.

3 Observational Determinism in the Literature

Before presenting our improved definition, this section first discusses the existing definitions of observational determinism, and their shortcomings.

3.1 Existing Definitions of Observational Determinism

Given any two initial low-equivalent stores, $s_1 =_L s_2$, a program C is *observationally deterministic*, according to

- Zdancewic and Myers [14]: iff any two low location traces are equivalent upto stuttering and prefixing, i.e., $\forall T, T'.\ \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ \forall l \in L.\ T_{|_l} \sim_p T'_{|_l}$.
- Huisman et al. [6]: iff any two low location traces are equivalent upto stuttering, i.e., $\forall T, T'.\ \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ \forall l \in L.\ T_{|_l} \sim T'_{|_l}$.
- Terauchi [13]: iff any two low store traces are equivalent upto stuttering and prefixing, i.e., $\forall T, T'.\ \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ T_{|_L} \sim_p T'_{|_L}$.

Notice that the existing definitions all have implicitly assumed a nondeterministic scheduler, without mentioning this explicitly. Zdancewic and Myers, followed by Terauchi, allow equivalence upto prefixing. The definition of Huisman et al. is stronger than the one of Zdancewic and Myers, as it only allows stuttering equivalence. Both definitions of Zdancewic and Myers, and Huisman et al. only specify equivalence of traces on each single low location separately; they do not consider the relative order of variable updates in traces, while Terauchi does. In particular, Terauchi's definition is stronger than Zdancewic and Myers' definition as it requires equivalence upto stuttering and prefixing on low store traces instead of on low location traces.

3.2 Shortcomings of These Definitions

Unfortunately, all these definitions have shortcomings. Huisman et al. showed that allowing prefixing of low location traces, as in the definition of Zdancewic and Myers, can reveal secret information, see [6]. Further, as observed by Terauchi, attackers can derive secret information from the relative order of updates, see [13]. It is not sufficient to require that only the low location traces are deterministic for a program to be secure. Therefore, Terauchi required that all low store traces should be equivalent upto stuttering and prefixing. However, allowing prefixing of full low store traces still can reveal secret information. Moreover, the requirement that traces have to agree on updates to all low locations as a whole is overly restrictive. In addition, all these definitions accept programs that behave insecurely under some specific schedulers. All these shortcomings are illustrated below by several examples. In all examples, we assume an observational model where attackers can access the full code of the program, observe the traces of public data, and limit the set of possible program traces by choosing a scheduler.

How equivalences upto prefixing can reveal information

Example 1. Consider the following program. Suppose $\mathtt{h} \in H$ and $\mathtt{l1}$, $\mathtt{l2} \in L$, $\mathtt{h}$ is a Boolean, 0 or 1,

$$\{\mathtt{if}\ (\mathtt{l1} == \mathtt{1})\ \mathtt{then}\ (\mathtt{l2} := \mathtt{h})\ \mathtt{else}\ \mathtt{skip}\} \parallel \mathtt{l1} := \mathtt{1}$$

For notational convenience, let C_1 and C_2 denote the left and right operands of the parallel composition operator in all examples. A low store trace is denoted by a sequence of low stores, containing the values of the low variables in order, i.e., ($\mathtt{l1}$, $\mathtt{l2}$). If we execute this program from several low-equivalent stores for different values of $\mathtt{h}$, we obtain the following low store traces.

$$\text{Case } \mathtt{h} = 0 : T_{|_L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,0)] & \text{execute } C_2 \text{ first} \end{cases}$$
$$\text{Case } \mathtt{h} = 1 : T_{|_L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,1)] & \text{execute } C_2 \text{ first} \end{cases}$$

According to Zdancewic and Myers, and Terauchi, this program is observationally deterministic. However, when $\mathtt{h} = \mathtt{1}$, we can terminate in a state where $\mathtt{l2} = \mathtt{1}$. It means that when the value of $\mathtt{l2}$ changes, an attacker can conclude that surely $\mathtt{h} = \mathtt{1}$; partial information still can be leaked because of prefixing.

We believe that Zdancewic and Myers, and Terauchi defined observational determinism using equivalence upto stuttering *and* prefixing, because they considered that termination could only reveal one bit of information (and technically, it simplified the definition of the type systems that they used to verify the property). However, in our opinion, prefixing allows to leak more than one bit of information, as illustrated by Example 1 and by Huisman et al. [6], and therefore only equivalence upto stuttering should be allowed.

How too strong conditions reject too many programs The restrictiveness of Terauchi's definition stems from the fact that no variation in the relative order of updates is allowed. This rejects many harmless programs.

Example 2. Consider the following program.

$$\mathtt{l1} := \mathtt{3} \parallel \mathtt{l2} := \mathtt{4}$$

If C_1 is executed first, we get the following trace, $T_{|_L} = [(0,0),(3,0),(3,4)]$; otherwise, $T_{|_L} = [(0,0),(0,4),(3,4)]$.

This program is rejected by Terauchi, because not all low store traces are equivalent upto stuttering and prefixing.

How scheduling policies can be exploited by attackers In all examples given so far, a nondeterministic scheduler is assumed. The security of a program depends strongly on the scheduler's behavior, and in practice, the scheduler

may vary from execution to execution. Under a specific scheduler, some traces *cannot occur*. Because an attacker knows the full code of the program, using an appropriate scheduler, secret information can be revealed from the limited set of possible traces. This sort of attack is often called a refinement attack [12, 2], because the choice of scheduling policy refines the set of possible program traces.

Example 3. Consider the following program.

$$\{\{\texttt{if}\ (\texttt{h} > 0)\ \texttt{then sleep(n)}\};\ \texttt{l} := 1\} \parallel \texttt{l} := 0$$

where `sleep(n)` abbreviates `n` consecutive *skip* commands. Under a nondeterministic scheduler, the initial value of `h` cannot be derived; this program is accepted by the definitions of Zdancewic and Myers, and Terauchi.

However, suppose we execute this program using a *round-robin* scheduling policy, i.e., the scheduler picks a thread and then proceeds to run that thread for m steps, before giving control to the next thread. If $m < n$ we obtain low store traces of the following shapes.

$$\text{Case } \texttt{h} \leq 0: \quad T_{|_L} = \begin{cases} [(0),(1),(0)] \text{ execute } C_1 \text{ first} \\ [(0),(0),(1)] \text{ execute } C_2 \text{ first} \end{cases}$$
$$\text{Case } \texttt{h} > 0: \quad T_{|_L} = \begin{cases} [(0),(0),\ldots,(0),(1)] \text{ execute } C_1 \text{ first} \\ [(0),(0),\ldots,(0),(1)] \text{ execute } C_2 \text{ first} \end{cases}$$

Thus, only when $\texttt{h} \leq 0$, we can terminate in a state where $\texttt{l} = 0$. Thus, the final value of `l` may reveal whether `h` is positive or not.

Example 4. Consider the following program.

$$\{\texttt{if}\ (\texttt{h} > 0)\ \texttt{then l1} := 1\ \texttt{else l2} := 1\} \parallel \{\texttt{l1} := 1; \texttt{l2} := 1\} \parallel \{\texttt{l2} := 1; \texttt{l1} := 1\}$$

This program is secure under a nondeterministic scheduler, and it is accepted by the definitions of Zdancewic and Myers, and Huisman et al. However, when an attacker chooses a scheduler which always executes the leftmost thread first, he gets only two different kinds of traces, corresponding to the values of `h`: when $\texttt{h} > 0$, $T_{|_L} = [(0,0),(1,0),(1,1),\ldots]$; otherwise, $T_{|_L} = [(0,0),(0,1),(1,1),\ldots]$.

This program is accepted by the definitions of Zdancewic and Myers, and Huisman et al. but it is not secure under this scheduler: attackers can learn information about `h` by observing whether `l1` is updated before `l2`. Notice that the problem of relative order of updates was shown by Terauchi [13].

To conclude, the examples above show that all the existing definitions of observational determinism allow programs to reveal private data because they allow equivalence upto prefixing, as in the definitions of Zdancewic and Myers, and Terauchi, or do not consider the relative order of updates, as in the definitions of Zdancewic and Myers, and Huisman et al. The definition of Terauchi is also overly restrictive, rejecting many secure programs. Moreover, all these definitions are not scheduler-specific. They accept programs behaving insecurely under a specific scheduling policy. This is our motivation to propose a new definition of scheduler-specific observational determinism. This definition on one hand only accepts secure programs, and on the other hand is less restrictive on innocuous programs w.r.t. a particular scheduler.

4 Scheduler-Specific Observational Determinism

To overcome the problems discussed above, we propose a new definition of scheduler-specific observational determinism. The definition has two conditions: (1) any two location traces for each low variable should be stuttering-equivalent and (2) given a low store trace starting in store s_1, for any low-equivalent store s_2 there exists a low store trace starting in s_2 such that these traces are stuttering-equivalent. Formally, it is defined as follows.

Definition 3 (δ-specific observational determinism (SSOD)).
Given a scheduling policy δ, a program C is δ-specific observationally deterministic w.r.t. *L iff for all initial low-equivalent stores s_1, s_2, $s_1 =_L s_2$, conditions (1) and (2) are satisfied.*

$$(1) \qquad \forall T, T'.\ \langle C, s_1\rangle \Downarrow_\delta T \wedge \langle C, s_2\rangle \Downarrow_\delta T' \Rightarrow \forall l \in L.\ T_{|_l} \sim T'_{|_l}.$$

$$(2) \qquad \forall T.\ \langle C, s_1\rangle \Downarrow_\delta T.\ \exists T'.\ \langle C, s_2\rangle \Downarrow_\delta T' \wedge T_{|_L} \sim T'_{|_L}.$$

Notice that the execution of a program under a nondeterministic scheduler contains all possible interleavings of threads. Thus, given any scheduling policy δ, the set of possible program traces under δ is a subset of the set of program traces under a nondeterministic scheduler.

We claim that this definition approximates the intuitive notion of security even better than the earlier definitions. First of all, it does not allow information to be leaked because of prefixing. Second, all location traces for each low location must be stuttering-equivalent, thus a program must enforce an ordering on the accesses to a single low location, i.e., the sequence of operations performed at a single low location is deterministic [14]. Requiring only that a stuttering-equivalent low location trace exists is not sufficient, as in the following example.

Example 5. Consider the following program, where `h` is a Boolean,

$$\texttt{if}\ (\texttt{h})\ \texttt{then}\ \{\texttt{l} := 0; \texttt{l} := 1\} \parallel \texttt{l} := 0\ \texttt{else}\ \{\texttt{l} := 0; \texttt{l} := 1\} \parallel \{\texttt{l} := 0; \texttt{l} := 0\}$$

This program leaks information under a nondeterministic scheduler, because when `h` is 1, `l` is more likely to contain 1 than 0 in the final states. However, there always exists a matching low location trace for `l`. Therefore, we require instead that the low location traces are deterministic.

It should be noted that a consequence of the requirement that low location traces are deterministic is that programs such as `l:=0‖l:=1` are also rejected, because its set of traces cannot be distinguished from the traces of Example 5.

Third, using the second condition, we ensure that differences in the relative order of updates are independent of private data. This makes our definition more relaxed than Terauchi's, accepting programs as Example 2, but rejecting programs such as `if (h) then {l1:=1;l2:=2} else {l2:=2;l1:=1}` (example from Terauchi [13]), where the relative order of updates reveals information about `h`.

Finally, it should be stressed that SSOD is scheduler-specific. Security is only guaranteed under a particular scheduler; executing the program under a different scheduler might change the program's security. This is in particular relevant for condition (2): existence of a matching trace depends on the scheduler that is chosen.

If we quantify Definition 3 over all possible schedulers, we obtain a *scheduler-independent* definition of observational determinism. Essentially, this boils down to the following requirement: for any two low-equivalent initial stores, any two *low store traces* obtained from the execution of a program under a nondeterministic scheduler are stuttering-equivalent.

To conclude, we explicitly list the different properties of SSOD.

Property 1 (Deterministic low location traces). If a program is accepted by Definition 3, no secret information can be derived from the observable location traces. It is required that the low locations individually evolve deterministically, and thus, private data may not affect the values of low variables.

Property 2 (Relative order of updates). If a program is accepted by Definition 3, the relative order of updates is independent from the private data. This is ensured by the requirement that there always is a matching low store trace (for any possible low-equivalent initial store).

Notice that the insecure programs in Examples 1 and 3 are rejected by our definition under the scheduler that is used to execute the program. The program in Example 4 is secure under a nondeterministic scheduler and it is accepted by our definition instantiated accordingly. However, it is insecure under more specific schedulers that have a smaller set of possible traces. For example if it is executed under a scheduler that always chooses the leftmost thread to execute first then it is rejected by Definition 3.

Property 3 (Less restrictive on harmless programs). Compared with Terauchi's definition, Definition 3 is more permissive: it allows some freedom in the order of individual updates, as long as they are independent of private data.

For example, Example 2 and 4, which are secure, are accepted by our definition instantiated with a nondeterministic scheduler, but rejected by Terauchi.

After having presented an improved definition of observational determinism, the next sections discuss an approach how this definition can be verified formally.

5 A Temporal Logic Characterization of Stuttering Equivalence

For the verification of the SSOD property, we choose to characterize the property as a model-checking problem. The most difficult part of this characterization is to express stuttering-equivalence in temporal logic. This section first discusses our general approach to verification, and then it shows how stuttering-equivalence is characterized in temporal logic. The next section then uses this to express observational determinism as a model-checking problem.

5.1 Self-Composition to Verify Information Flow Properties

A common approach to check information flow properties is to use a type system. However, the type-based approach is not suitable to verify Definition 3. First, type systems for multi-threaded programs often aim to prevent secret information from affecting the thread timing behavior of a program, e.g., secret information can be derived from observing the internal timing of actions [14]. Therefore, type systems that have been proposed to enforce confidentiality for multi-threaded programs are often very restrictive. This restrictiveness makes the application programming become impractical; many intuitively secure programs are rejected by this approach, i.e., $\mathtt{h} := \mathtt{l}; \mathtt{l} := \mathtt{h}$. Besides, it is difficult to enforce stuttering-equivalence via type-based methods without being overly restrictive [13]. In addition, type systems are not suitable to verify existential properties, as the one in Definition 3.

Instead, we use self-composition. This is a technique [3, 1] to transform the verification of information flow properties into a verification problem. It means that we compose a program C with its copy, denoted C', i.e., we execute C and C' in parallel, and consider $C \| C'$ as a single program (called a *self-composed program*). Notice that C' is program C, but with all variables renamed to make them distinguishable from the variables in C [1]. In this model, the original two programs still can be distinguished, and therefore we can express the information flow property as a property over the execution of the self-composed program.

Concretely, in this paper we characterize SSOD with a temporal logic formula. The essence of observational determinism is stuttering-equivalence of execution traces. Therefore, we first investigate the characteristics of stuttering-equivalence and discuss which extra information is needed to characterize this in temporal logic. Based on the idea of self-composition and the extra information, we define a model over which we want the temporal logic formula to hold. After that, a temporal logic formula that characterizes stuttering-equivalence is defined. This formula can be instantiated in different ways, depending on the equivalence relation that is used in the stuttering-equivalence. SSOD is expressed in terms of the stuttering-equivalence characterization. This results in a conjunction of an LTL and a CTL formula (for the syntax and semantics of LTL and CTL, see [7]). Both formulas are evaluated over a single execution of the self-composed program. We show that validity of these formulas is equivalent to the original definition, thus the characterization as a model-checking problem is sound and complete.

5.2 Characteristics of Stuttering-Equivalence

First we look at the characteristics of stuttering-equivalence. Let symbols $\mathbf{a}, \mathbf{b}, \mathbf{c}$, etc. represent states in traces. Given $T \sim T'$ as follows,

index:	0 1 2 3 4 5 ...
$T =$	**a b c d d d** ...
nr of state changes in T:	0 1 2 3 3 3
$T' =$	**a a b b c d** ...
nr of state changes in T':	0 0 1 1 2 3

The top row indicates the indexes of states. The row below each trace indicates the total numbers of state changes, counted from the first state, that happened in the trace. Based on this example, we can make some general observations about stuttering-equivalence that form the basis for our temporal logic characterization.

- Any state change that occurs *first* in trace T at index i, i.e., T_i, will also occur later in trace T' at some index $j \geq i$.
- For any index r between such a *first* and *second* occurrence of a state change, i.e., $i \leq r < j$, at state T'_r, the total number of state changes is *strictly smaller* than the total number of state changes at T_r.
- Similarly for any change that occurs *first* in trace T'.

Notice that these properties exactly characterize stuttering-equivalence (see Appendix A.2 of [5]).

5.3 State Properties

To characterize stuttering-equivalence in temporal logic, we have to come up with a temporal logic formula over a combined trace. As a convention, we use T^1 and T^2 to denote the two component traces. Thus, the i^{th} state of the combined trace contains both T^1_i and T^2_i. The essence of stuttering-equivalence is that any state change occurring in one trace also has to occur in the other trace. Therefore, we have to extend the state with extra information that allows us to determine for a particular state (1) whether the current state is different from the previous one, (2) whether a change occurs first or second, and (3) how many state changes have already happened.

State changes To determine whether a state change occurred, we need to know the previous state. Therefore, we define a *memorizing transition relation*, remembering the previous state of each transition.

Definition 4 (Memorizing transition relation). *Let* $\rightarrow \subseteq (State \times State)$ *be a* transition relation*. The memorizing transition relation* $\rightarrow_m \subseteq (State \times State) \times (State \times State)$ *is defined as:* $(c_1, c'_1) \rightarrow_m (c_2, c'_2) \Leftrightarrow c_1 \rightarrow c_2 \wedge c'_2 = c_1$.

Thus, (c_1, c'_1) makes a memorizing transition to (c_2, c'_2) if (1) c_1 makes a transition to c_2 in the original system, and (2) c'_2 remembers the old state c_1. We use accessor functions *current* and *old* to access the components of the memorized state, such that $current(c_1, c'_1) = c_1 \wedge old(c_1, c'_1) = c'_1$. A state change can be observed by comparing old and current components of a single state.

The order of state changes To determine whether a state change occurs for the first time or has already occurred in the other trace, we use a queue of states, denoted q. Its contents represents the *difference* between the two traces. We have the following operations and queries on a queue: *add*, adds an element

to the end of the queue, *remove*, removes the first element of the queue, and *first*, returns the first element of the queue. In addition, we use an extra state component *lead*, that indicates which component trace added the last state in q, i.e., $lead = m$ $(m = 1, 2)$ if the last element in q was added from T^m. Initially, the queue is empty (denoted ε), and *lead* is 0.

The rules to add/remove a state to/from the queue are the following. Whenever a state change occurs for the first time in T^m, the current state is added to the queue and *lead* becomes m. When this state change occurs later in the other trace, the element will be removed from the queue. When a state change in one trace does not match with the change in the other trace, both q and *lead* become undefined, denoted $\perp$, indicating a blocked queue. If $q = \perp$ (and $lead = \perp$), the component traces are not stuttering-equivalent, and therefore we do not have to check the remainders of the traces. Therefore, operations *add* and *remove* are not defined when q and *lead* are $\perp$.

Formally, these rules for adding and removing are defined as follows. Initially, q is ε and *lead* is 0. Whenever $q \neq \perp$ and $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$,

- if $lead = 3 - m$ and $T_i^m = \mathit{first}(q)$, then $\mathit{remove}(q)$. If $q = \varepsilon$, set $lead = 0$.
- if $lead = m$ or $lead = 0$, then $\mathit{add}(q, T_i^m)$ and set $lead = m$.
- otherwise, set $q = \perp$ and $lead = \perp$.

The number of state changes To determine the number of state changes that have happened, we extend the state with counters nr_ch^1 and nr_ch^2. Initially, both nr_ch^1 and nr_ch^2 are 0, and whenever a state change occurs, i.e., $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$, then nr_ch^m increases by one. Thus, the number of state changes at T_i^1 and T_i^2 can be determined via the values of nr_ch^1 and nr_ch^2, respectively.

5.4 Program Model

Next we define a model over which a temporal logic formula should hold. Given program C and two initial stores s, s', we take the parallel composition of C and its copy C'. In this model, the store of $C \parallel C'$ can be considered as the product of the two separate stores s and s', ensuring that the variables from the two program copies are disjoint, and thus that updates are done locally, i.e., not affecting the store of the other program copy. First, we define the elements of the program model.

States A state of a composed trace is of the form $(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$, where $\langle C_3 \parallel C_4, (s_3, s_4)\rangle$ remembers the old configuration (via the memorizing transition relation of Definition 4), and χ is extra information, as discussed above, of the form $(nr\_ch^1, nr\_ch^2, q, lead)$. We define accessor functions $conf_1$, $conf_2$, and *extra* to extract $(\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$, $(\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$, and χ, respectively.

Thus, in our model, the original two program copies still can be distinguished and the updates of program copies are done locally. Therefore, if $\mathcal{T}$ is a trace

$$\frac{lead = 2 \quad c = first(q) \quad nr\_ch^{1\prime} = nr\_ch^1 + 1 \quad q' = remove(q) \quad lead' = 1}{(nr\_ch^1, nr\_ch^2, q, lead) \overset{c}{\hookrightarrow} (nr\_ch^{1\prime}, nr\_ch^{2\prime}, q', lead')}$$

$$\frac{lead \in \{0,1\} \quad lead' = 1 \quad nr\_ch^{1\prime} = nr\_ch^1 + 1 \quad q' = add(q,c)}{(nr\_ch^1, nr\_ch^2, q, lead) \overset{c}{\hookrightarrow} (nr\_ch^{1\prime}, nr\_ch^{2\prime}, q', lead')}$$

$$\frac{lead \notin \{0,1\} \quad c \neq first(q) \quad nr\_ch^{1\prime} = nr\_ch^1 + 1 \quad q' = \bot \quad lead' = \bot}{(nr\_ch^1, nr\_ch^2, q, lead) \overset{c}{\hookrightarrow} (nr\_ch^{1\prime}, nr\_ch^{2\prime}, q', lead')}$$

Fig. 1. Definition of $\hookrightarrow$

of the composed model, then we can decompose it into two individual traces by functions Π_1 and Π_2, respectively, defined as $\Pi_m = map(conf_m)$. Thus, given a state $\mathcal{T}_i = (\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$ of the composed trace, then $(\Pi_1(\mathcal{T}))_i = (\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$ and $(\Pi_2(\mathcal{T}))_i = (\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$. The current configuration of program copy m can be extracted by function Γ_m, defined as $\Gamma_m = map(current) \circ \Pi_m$. Thus, $(\Gamma_1(\mathcal{T}))_i = \langle C_1, s_1\rangle$ and $(\Gamma_2(\mathcal{T}))_i = \langle C_2, s_2\rangle$. Finally, $extra(\mathcal{T}_i)(x)$ denotes the value of the extra information x at $\mathcal{T}_i$, for $x \in \{nr\_ch^1, nr\_ch^2, q, lead\}$.

Transition Relation The transition relation $\rightarrow_{\chi,\delta}$ is defined as the composition of a relation on the operational semantics, and a relation on the extra information. More precisely, the first component is the memorizing transition relation $\rightarrow_m$ (cf. Definition 4), derived from the transition relation induced by the operational semantics of programs executed under scheduler δ. The second component is a relation $\hookrightarrow \subseteq \chi \times Conf \times \chi$ that describes how the extra information evolves, following the rules in Figure 1. Notice that $\hookrightarrow$ is parametric on the concrete equality relation used. Concretely, $\rightarrow_{\chi,\delta}$ is defined by rules such as (with similar rules for when C_1 terminates, i.e., $\langle C_1, s_1\rangle \rightarrow \langle \epsilon, s_1\rangle$, and the symmetric counterparts for C_2):

$$\frac{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2) \rightarrow_m (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, \langle C_1 \parallel C_2, (s_1, s_2)\rangle) \quad \chi \overset{\langle C_1', s_1'\rangle}{\hookrightarrow} \chi'}{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2, \chi) \rightarrow_{\chi,\delta} (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, \langle C_1 \parallel C_2, (s_1, s_2)\rangle, \chi')}$$

Notice that above we studied stuttering-equivalence in a generic way, where two traces could make a state change simultaneously. However, in the model of the self-composed program, the operational semantics of parallel composition ensures that in every step, either C_1 or C_2, but not both, make a transition. Therefore, for any trace $\mathcal{T}$, state changes do not happen simultaneously in both $\Pi_1(\mathcal{T})$ and $\Pi_2(\mathcal{T})$. This also means that it can never happen that in one step, both *add* and *remove* are applied simultaneously on the queue.

Atomic Propositions and Valuation Next we define the atomic propositions of our program model, together with their valuation. Notice that their valuation

is parametric on the concrete equality relation used. Below, when characterizing SSOD, we instantiate this in different ways, to define stuttering-equivalence on a low location trace, and on a low store trace, respectively.

We define the following atomic propositions (for $m = 1, 2$):

- $\mathit{fst\_ch}^m$ denotes that a state change occurs for the first time in program copy m.
- $\mathit{snd\_ch}^m$ denotes that a state change occurs in program copy m, while program copy $3 - m$ has already made this change.
- $\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m}$ denotes that the number of state changes made by program copy m is less than the total number of state changes made by program copy $3 - m$.

The valuation function λ for these atomic propositions is defined as follows. Let $\mathfrak{c}$ denote a state of the composed trace.

$$\begin{aligned}
\mathit{fst_ch}^m \in \lambda(\mathfrak{c}) \Leftrightarrow\ & \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
& \mathit{extra}(\mathfrak{c})(\mathit{lead}) = m \text{ or } \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 0 \\
\mathit{snd_ch}^m \in \lambda(\mathfrak{c}) \Leftrightarrow\ & \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
& \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 3 - m \text{ and} \\
& \mathit{current}(\mathit{conf}_m(\mathfrak{c})) = \mathit{first}(\mathit{extra}(\mathfrak{c})(q)) \\
\mathit{nr_ch}^m < \mathit{nr_ch}^{3-m} \in \lambda(\mathfrak{c}) \Leftrightarrow\ & \mathit{extra}(\mathfrak{c})(\mathit{nr_ch}^m) < \mathit{extra}(\mathfrak{c})(\mathit{nr_ch}^{3-m})
\end{aligned}$$

Program Model Using the definitions above, we define a program model, encoding the behavior of a self-composed program under a scheduler δ. The characterizations are expressed over this model.

Definition 5 (Program model). *Given a scheduler δ, let C be a program, and s_1 and s_2 be stores. The* program model $\mathcal{M}^{\delta}_{C,s_1,s_2}$ *is defined as* $(\Sigma, \rightarrow_{\chi,\delta}, AP, \lambda, I)$ *where:*

- *Σ denotes the set of all configurations, obtained by executing from the initial configuration under δ, including the extra information, as defined above;*
- *AP is the set of atomic propositions defined above, and λ is their valuation;*
- *$I = \{\langle C \parallel C', (s_1, s_2)\rangle\}$ is the initial configuration of the composed trace.*

5.5 Characterization of Stuttering-Equivalence

Based on the observations and program model above, we characterize stuttering-equivalence by an LTL formula ϕ.

$$\phi = G\Big(\bigwedge_{m \in \{1,2\}} \mathit{fst\_ch}^m \Rightarrow \mathit{nr\_ch}^{3-m} < \mathit{nr\_ch}^m \ \ U\ \mathit{snd\_ch}^{3-m} \Big).$$

This formula expresses the characteristics of stuttering-equivalence: any state change occurring in one component trace will occur later in the other component trace; and in between these changes, the number of state changes at the intermediate states in the latter is strictly smaller than in the first.

We prove formally that ϕ characterizes stuttering-equivalence.

Theorem 1. *Let $\mathcal{T}$ be a composed trace that can be decomposed into T^1 and T^2 with $T^1_0 = T^2_0$, then $T^1 \sim T^2 \Leftrightarrow \mathcal{T} \models \phi$.*

Proof. See Appendix A.2 of [5].

6 Temporal Logic Characterization of SSOD

Based on the results above, this section defines a temporal logic formula characterizing scheduler-specific observational determinism. The formula consists of two parts: one that expresses stuttering-equivalence of low location traces, and one that expresses stuttering-equivalence of low store traces. Both are instantiations of the formula characterizing stuttering-equivalence defined above.

6.1 Atomic Propositions

To support the characterization of stuttering-equivalence in different ways, we define different atomic propositions. To characterize stuttering-equivalence over low location traces, we use atomic propositions $fst\_ch_l^m$, $snd\_ch_l^m$, and ${nr\_ch_l}^m < nr\_ch_l^{3-m}$ for each $l \in L$. To characterize stuttering-equivalence over low store traces, we use atomic propositions $fst\_ch_L^m$, $snd\_ch_L^m$, and ${nr\_ch_L}^m < nr\_ch_L^{3-m}$.

The formal definitions are as defined in the previous section, where equality is instantiated as $=_l$ (for $l \in L$) and $=_L$, respectively.

6.2 Characterization of SSOD

Now we can characterize the SSOD property in temporal logic. A program C is observationally deterministic under δ iff for any two low equivalent stores s_1 and s_2, the following formula holds on the traces of $\mathcal{M}^{\delta}_{C,s_1,s_2}$.

$$\left(\bigwedge_{l \in L} \phi_l\right) \wedge \phi_L, \text{ where}$$

$$\phi_l = G\Big(\bigwedge_{m \in \{1,2\}} fst\_ch_l^m \Rightarrow {nr\_ch_l}^{3-m} < nr\_ch_l^m \ \ U \ snd\_ch_l^{3-m}\Big)$$

$$\phi_L = AG\Big(\bigwedge_{m \in \{1,2\}} fst\_ch_L^m \Rightarrow E({nr\_ch_L}^{3-m} < nr\_ch_L^m \ \ U \ snd\_ch_L^{3-m})\Big)$$

Notice that ϕ_l is an LTL and ϕ_L a CTL formula.

For a program with n low variables, we thus have $n + 1$ verification tasks: n tasks relate to low location traces and one task relates to low store traces. For each task, we instantiate the extra information χ and the equality relation differently.

Theorem 2. *Given program C and initial stores s_1 and s_2 such that $s_1 =_L s_2$, C is observationally deterministic under δ iff*

$$\mathcal{M}^{\delta}_{C,s_1,s_2} \models \left(\bigwedge_{l \in L} \phi_l\right) \wedge \phi_L.$$

Proof. See Appendix A.3 of [5].

7 Conclusion

This paper presents a new scheduler-specific definition of observational determinism: the SSOD property. We claim that it captures the intuitive definition of observational determinism more precisely than the existing definitions. If a program is accepted under a specific scheduler, no secret information can be derived from the publicly observable location traces and the relative order of updates.

Compliance with SSOD can be verified via a characterization as a temporal logic formula. The characterization is developed in two steps: first we characterize stuttering-equivalence, which is the basis of the definition of scheduler-specific observational determinism, and then we characterize the SSOD property itself. The characterization is an important step towards model-checking observational determinism properties.

Related Work The idea of observational determinism originates from the notion of noninterference, which only considers input and output of programs. We refer to [12, 6] for a more detailed description of noninterference, its verification, and a discussion why it is not appropriate for multi-threaded programs.

Roscoe [10] was the first to state the importance of determinism to ensure secure information flow of multi-threaded programs. The work of Zdancewic and Myers, Huisman et al., and Terauchi [14, 6, 13] has been mentioned above. They all propose different formal definitions of observational determinism, with different verification methods. In particular, Zdancewic and Myers, and Terauchi propose type systems. Huisman et al. characterize observational determinism in CTL*, using a special non-standard synchronous composition operator, and also in the polyadic modal μ-calculus (a variation of the modal μ-calculus). The idea of using self-composition was first proposed by Barthe et al. and Darvas et al. [1, 3]. The way self-composition is used here, with a temporal logic characterization, also bears resemblance with temporal logic characterizations of strong bisimulation [8]. Finally, Russo and Sabelfeld take a different approach to ensure security of a multi-threaded program. They propose to restrict the allowed interactions between threads and scheduler [11]. This allows them to present a compositional security type system which guarantees confidentiality for a wide class of schedulers. However, the proposed security specification is similar to noninterference, just considering input and output of a program.

Future Work As future work, we plan to apply the approach on realistic programs. This paper gives a theoretical and general description how the property can be verified. However, in the model-checking literature, also specialized algorithms exist to verify stuttering-equivalence. We will investigate if we can adapt these algorithms to make the verification more efficient. Ultimately, we would like to implement the algorithm as part of an existing model checker tool.

An additional challenge is to make the program model parametric, so that properties can be expressed for varying initial values. This step will be necessary to scale to large applications.

Notice that observational determinism is a *possibilistic* secure information flow property: it only considers the non-determinism that is possible in an execution, but it does not consider the probability that a step will take place. In a separate line of work, we will also study how probability can be used to guarantee secure information flow.

Acknowledgments The authors would like to thank Jaco van de Pol for his useful comments and the anonymous reviewers for useful feedback of an earlier version of this paper. Our work is supported by the Netherlands Organization for Scientific Research as part of the SlaLoM project.

References

1. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.
2. G. Barthe and L. Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 13–22. ACM, 2004.
3. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.
4. M. Huisman and H.-C. Blondeel. Model-checking secure information flow for multi-threaded programs. In *Theory of Security and Applications (Tosca)*, volume 6993 of *LNCS*, pages 148–165. Springer, 2011.
5. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. Technical Report TR-CTIT-11-22, CTIT, University of Twente, Netherlands, 2011.
6. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterization of observation determinism. In *Computer Security Foundations Workshop*. IEEE Computer Society, 2006.
7. M. Huth and M. Ryan. *Logic in computer science: modeling and reasoning about the system*. Cambridge University Press, second edition, 2004.
8. A. Parma and R. Segala. Logical characterizations of bisimulations for discrete probabilistic systems. In *Proceedings of the 10th international conference on Foundations of software science and computational structures*, FOSSACS'07, pages 287–301. Springer-Verlag, 2007.
9. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Processing Letters*, 63:243–246, 1997.
10. A.W. Roscoe. Csp and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.
11. A. Russo and A. Sabelfeld. Security interaction between threads and the scheduler. In *Computer Security Foundations Symposium*, pages 177–189, 2006.
12. A. Sabelfeld and A. Myers. Language-based information flow security. In *IEEE Journal on Selected Areas in Communications*, volume 21, pages 5–19, 2003.
13. T. Terauchi. A type system for observational determinism. In *Computer Science Foundations*, 2008.
14. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43. IEEE Press, June 2003.