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 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 ensure that an accepted program behaves securely under the scheduler that is used to deploy 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 discuss how compliance with our definition can be verified, using model checking. We use the idea of self-composition and we rephrase the observational determinism property for a single program C as a temporal logic formula over the program C executed in parallel with an independent copy of itself. Thus two states reachable during the execution of C are combined into a reachable program state of the self-composed program. This allows to compare two program executions in a single temporal logic formula. The actual characterization is done in two steps. First we discuss how stuttering equivalence can be characterized as a temporal logic formula. Observational determinism is then expressed in terms of the stuttering equivalence characterization. This results in a conjunction of an LTL and a CTL formula, that are amenable to 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 consider the whole execution traces, i.e., the sequences of states that occur during program execution.

In the literature, different definitions of confidentiality are proposed for multi-threaded programs. This paper follows the approach advocated by Roscoe [11] 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 [16]. Several formal definitions are proposed [16, 7, 14], but none of these capture exactly this intuitive definition.

The first formal definition of observational determinism was proposed by Zdancewic and Myers [16]. 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[1], 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 trace of each low variable separately. Zdancewic and Myers also argue that prefixing is sufficiently strong equivalence relation, as this only causes external termination leaks of one bit of information [16].

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 [7]. 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 [14]. 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 stuttering and prefixing equivalent, thus not considering the variables independently.

However, Terauchi's definition is also not satisfactory. This is for several reasons: 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.

[1] 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.

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 non-deterministic 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 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 [7, 5] 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 [8] and a CTL [8] 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 illustrates their shortcomings on several examples. 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 unit of commands that can be scheduled by an 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.

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

Parallel programs communicate via shared variables in a global store. For simplicity, we assume that assignments and lookups are atomic, thus data races (where two variable accesses can occur simultaneously) cannot happen, and we can assume an interleaving semantics (cf. [4]). We also 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):

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

We also have a special transition step for terminated programs, i.e., $\langle \epsilon, s\rangle \rightarrow \langle \epsilon, s\rangle$, 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 live threads, i.e., the set of not-yet terminated threads. During the execution, a scheduling policy repeatedly decides which threads can be 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; and a *round-robin* scheduler assigns equal time slices to each thread in circular order. Given scheduling policy δ, and configuration $\langle C, s\rangle$, an infinite list of configurations $T = c_0c_1c_2...$

($T : \mathbb{N}_0 \rightarrow \mathit{Conf}$) is a *trace* of the execution of C from s under the control of δ, denoted $\langle C, s\rangle \Downarrow_\delta T$, iff $c_0 = \langle C, s\rangle$ and $\forall i \in \mathbb{N}_0.\ c_i \rightarrow c_{i+1}$ under δ. 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 different definitions of observational determinism is the equivalence of traces upto stuttering or upto stuttering and prefixing. The definition of stuttering equivalence is based on [10, 7]. It uses an auxiliary notion of *stuttering equivalence upto indexes i and j*.

Definition 1 (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.

Equivalence upto stuttering and prefixing is defined as one trace being stuttering equivalent to a prefix of the other trace.

Definition 2 (Prefixing and stuttering equivalence). *Traces T and T' are* prefixing and stuttering equivalent, *written $T \sim_p T'$, iff $\exists i. T \sim T'_{\ll i} \vee T_{\ll i} \sim T'$.*

3 Observational Determinism in the Literature

This section presents the existing definitions of observational determinism formally, and discusses their shortcomings. The next section presents our improved definition.

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 [16]: 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. [7]: 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 [14]: 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. This has as an advantage that it removes the obligation to consider program termination. 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 [7]. Further, as observed by Terauchi, attackers can derive secret information from the relative order of updates, see [14]. 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 stuttering and prefixing equivalent. However, allowing prefixing of full low store traces still can reveal secret information. Besides, the requirement that traces have to agree on updates to all the low locations as a whole, as in Terauchi's definition, is overly restrictive. In addition, all these definitions accept programs that behave insecurely under some specific schedulers. These shortcomings are illustrated below by several examples. In all examples, we assume an observational model is 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 prefixing equivalences can reveal information Consider the following program. Suppose $\texttt{h} \in H$ and $\texttt{l1}$, $\texttt{l2} \in L$, $\texttt{h}$ is a Boolean.

Example 1.

$$\begin{array}{l}\texttt{l1} := 0; \texttt{l2} := 0;\\ \{\texttt{if}\ (\texttt{l1} == 1)\ \texttt{then}\ (\texttt{l2} := \texttt{h})\ \texttt{else skip}\} \parallel \texttt{l1} := 1\end{array}$$

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., (`l1`, `l2`). If we execute this program from several low equivalent stores for different values of `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 `l2` changes, an attacker can conclude that surely $\mathtt{h} = \mathtt{1}$; partial information still can be leaked because of prefixing.

How too strong conditions reject too many programs The restrictiveness of Terauchi's definition arises from the fact that no variation in the relative order of updates is allowed at all. This rejects many harmless programs, such as for example,

Example 2.

$$\mathtt{l1} := \mathtt{0};\ \mathtt{l2} := \mathtt{0};\ \ \{\mathtt{l1} := \mathtt{3} \parallel \mathtt{l2} := \mathtt{4}\}$$

If C_1 is executed first, we get the following traces, $T_{|_L} = [(0,0),(3,0),(3,4)]$; 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. However, in practice, the scheduler may vary from execution to execution. The security of a program depends strongly on the scheduler's behavior. Under a specific scheduling policy, some traces *cannot occur*. Due to the fact that an attacker knows the full code of the program, when he chooses 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 [13, 2], because the choice of scheduling policy refines the set of possible program traces. Consider the following example,

Example 3.

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

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 store traces of the following shapes.

$$\begin{array}{ll}
\text{Case } \mathtt{h} \leq 0: & T_{|L} = \begin{cases} [(0),(1),(0)] \text{ execute } C_1 \text{ first} \\ [(0),(0),(1)] \text{ execute } C_2 \text{ first} \end{cases} \\
\text{Case } \mathtt{h} > 0: & T_{|L} = \begin{cases} [(0),(0),\ldots,(0),(1)] \text{ execute } C_1 \text{ first} \\ [(0),(0),\ldots,(0),(1)] \text{ execute } C_2 \text{ first} \end{cases}
\end{array}$$

With this scheduling policy, this program is still accepted by Zdancewic and Myers, and Terauchi. However, when $\mathtt{h} \leq 0$, we can terminate in a state where $\mathtt{l} = 0$. Thus, the final value of `l` may reveal whether `h` is positive or not.

Example 4.

$$\begin{array}{l}
\mathtt{l1} := 0; \mathtt{l2} := 0; \\
\{\mathtt{if}\ (\mathtt{h} > 0)\ \mathtt{then}\ \mathtt{l1} := 1\ \mathtt{else}\ \mathtt{l2} := 1\} \| \{\mathtt{l1} := 1; \mathtt{l2} := 1\} \| \{\mathtt{l2} := 1; \mathtt{l1} := 1\}
\end{array}$$

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 $\mathtt{h} > 0$, $T_{|L} = [(0,0),(1,0),(1,1),\ldots]$; otherwise, $T_{|L} = [(0,0),(0,1),(1,1),\ldots]$.

In this case, this program is still accepted by the definitions of Zdancewic and Myers, and Huisman et al. but this program is not secure anymore. 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 in [14].

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 say that a program is observationally deterministic under a particular scheduler if any two low location traces are stuttering equivalent *and* for any low store trace produced from one initial store, there exists a low store trace produced from the other initial low equivalent store such that these two traces are stuttering equivalent. Our definition does not allow information to be leaked because of prefixing equivalence. Notice that Zdancewic and Myers, and Terauchi allow prefixing equivalence because it removes the obligation to prove program termination in their proposed type systems.

Scheduler-specific observational determinism is defined formally as follows.

Definition 3 (δ-specific observational determinism).
*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, $s_1 =_L s_2$, the following conditions (1) and (2) are satisfied.*

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

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

We require that the low locations individually behave deterministically because in the literature it has been shown how nondeterminism of a low variable can be exploited to make other programs reveal confidential information. Even the simple program "`l` := `0` $\|$ `l` := `1`" can be used to violate confidentiality of another program. If public variables are shared between programs, there exists a channel between them [15]. Suppose that the public variable `l` is shared, i.e., this data is used by another apparently secure program, and access to this data is conditioned on confidential information, then this assignment is more likely to happen last. Therefore, there is a timing channel between two programs and it can be used to derive information about the confidential data, see [16, 15]. Therefore, to be considered secure, 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 [16].

However, notice that the program "`l1` := `3` $\|$ `l2` := `4`" in Example 2 is considered secure because it writes to two different locations.

Besides, this definition also releases the requirement that all low store traces have to agree on the relative order of updates. Our definition differs from the previous definitions of observational determinism in one important aspect: the existential condition. This condition depends strongly on the scheduling policy used to deploy the program because traces model possible runs of a program and refinements of the set of traces, when the scheduling policy changes, cannot guarantee this condition.

Notice that the execution of a program under a nondeterministic scheduler means that we consider all possible interleavings of threads. Given any scheduling policy δ, the set of possible program traces under δ is a subset of the set of possible program traces under a nondeterministic scheduler. If we quantify Definition 3 over all possible schedulers, it requires that each low store trace produced from one initial store under a nondeterministic scheduler must be matched with every low store trace produced from the other initial store. It means that for any two initial low equivalent stores, if any two low store traces obtained from the execution of a program under a nondeterministic scheduler are stuttering equivalent, this program is secure under any scheduling policy δ. Thus, this gives a truly scheduler-independent definition of observational determinism.

4.1 Properties of Scheduler-Specific Observational Determinism

To illustrate that Definition 3 captures the intended meaning of observational determinism best, we discuss different properties of the definition.
Property 1 (Deterministic low location traces). If a program is accepted by Definition 3, no secret information can be derived from the publicly observable location traces. It is required that the low locations individually evolve deterministically, and thus, the values of private variables may not affect the values of low variables.

Property 2 (Deterministic relative order of updates). If a program is accepted by Definition 3, no information can be derived from the relative order of updates because there is always a matching low store trace.

Notice that the insecure programs in Examples 1 and 3 are rejected by our definition under a nondeterministic scheduling policy. 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 the scheduler that always chooses the leftmost thread to execute first; and hence, it is rejected if we instantiate the definition with this scheduler. Thus, given a scheduling policy δ, if a program is accepted by our definition, instantiated for this scheduler, we can conclude that the program is secure under δ.

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 a matching execution exists.
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 how to verify it formally.

5 A Temporal Logic Characterization of Stuttering Equivalence

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 [16]. For to this reason, the type systems proposed to enforce confidentiality for multi-threaded programs are often very restrictive. This restrictiveness makes the application programming become impractical and many intuitively secure programs are rejected by type systems. Besides, it also seems difficult to enforce stuttering equivalence via type-based methods without being overly restrictive [14]. In addition, type systems are not suitable to verify existential properties, as the one in our definition. This can be understood as follows. If the program C is well-typed, then for any two configurations $c_1 = \langle C, s_1 \rangle$ and $c_2 = \langle C, s_2 \rangle$ such that $s_1 =_L s_2$, there exists a configuration, e.g., c', that simulates both [16]. This means that for any two traces T and T' starting in c_1 and c_2 respectively, the low-deterministic properties of T and T' can be simulated by the same trace starting in c'. In other words, if C is well-typed, two sets of traces starting in c_1 and c_2 have the same low-security behavior. Therefore, our definition, which contains an existential quantification, cannot be verified via type-based methods.

Instead, we use self-composition. This is a recently developed technique [3, 1] that transforms the verification of information flow properties into a verification problem. Self-composition means that we compose a program C with its copy, denoted C', i.e., we execute C and C' in parallel, and consider $C \parallel C'$ as a single 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 then we express the information flow property as a property over the executions of the self-composed program.

Concretely, in this paper we characterize observational determinism 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. Observational determinism 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 definitions of LTL and CTL, see [8]). Both formulas are evaluated over a single execution of the self-composed program. We show that the logic formulas are equivalent to the original definitions, thus the characterization as a model checking problem is sound and complete.

5.2 Characteristics of Stuttering Equivalence

We 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 are sound and complete to characterize stuttering equivalence, see Appendix A.2 of [6].

5.3 Extra Information

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_i^1 and T_i^2. 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 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.

How to characterize state change? 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* $\rightarrow \subseteq (\mathit{State} \times \mathit{State})$ *be a* transition relation. *The memorizing transition relation* $\rightarrow_m \subseteq (\mathit{State} \times \mathit{State}) \times (\mathit{State} \times \mathit{State})$ *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 $\mathit{current}(c_1, c'_1) = c_1 \wedge \mathit{old}(c_1, c'_1) = c'_1$.

A state change can now be observed by comparing old and current components of a single state.

How to characterize 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., $\mathit{lead} = m$ ($m = 1, 2$) 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 $\mathit{lead} = \perp$), 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 $\mathit{lead} = 3 - m$ and $T_i^m = \mathit{first}(q)$, then $\mathit{remove}(q)$. If $q = \varepsilon$, set $\mathit{lead} = 0$.
- if $\mathit{lead} = m$ or $\mathit{lead} = 0$, then execute $\mathit{add}(q, T_i^m)$ and set $\mathit{lead} = m$.
- otherwise, set $q = \perp$ and $\mathit{lead} = \perp$.

How to characterize the number of state changes? To determine the number of state changes that have happened, we extend the state with counters $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$. Initially, both $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$ are 0, and whenever a state change occurs, i.e., $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$, then $\mathit{nr\_ch}^m$ increases by one. Thus, the number of state changes at T_i^1 and T_i^2 can be determined via the values of $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$, 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, denoted C', and consider $C \parallel C'$ as a single program. 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 $(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead})$. 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 of the composed model, then we can decompose it into two individual traces by functions Π_1 and Π_2, respectively, defined as $\Pi_m = \mathit{map}(\mathit{conf}_m)$. 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 $\Gamma_m = \mathit{map}(\mathit{current}) \circ \Pi_m$. 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, $\mathit{extra}(\mathcal{T}_i)(x)$ denotes the value of the extra information x at $\mathcal{T}_i$, for $x \in \{\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead}\}$.

Transition Relation: Let $\rightarrow$ be the translation relation induced by the operational semantics of programs, and $\rightarrow_m$ the memorizing transition relation derived from $\rightarrow$ (cf. Definition 4). The transition relation of the program model $\rightarrow_\chi$ is defined using $\rightarrow_m$, and a relation $\rightarrow\ \subseteq \chi \times \mathit{Conf} \times \chi$ that describes how the extra information evolves, following the rules below (with a similar rule 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).

$$\frac{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2) \rightarrow_m (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, c_4) \quad \chi \xrightarrow{\langle C_1', s_1'\rangle} \chi'}{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2, \chi) \rightarrow_\chi (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, c_4, \chi')}$$

where $c_4 = \langle C_1 \parallel C_2, (s_1, s_2)\rangle$ and $\chi \xrightarrow{c} \chi'$ is defined as follows (notice that this relation is parametric on the concrete equality relation used).

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

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

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

Notice that above we studied stuttering equivalence in a generic way, where two traces could make a state change simultaneously. However, in the self-composed program model, 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: 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 observational determinism, we instantiate this in different ways, to define stuttering equivalence on a low location trace, and on a low store trace, respectively.

For $m = 1, 2$,

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

$$\begin{aligned}
\mathit{fst\_ch}^m \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and}\\
&\quad\ \mathit{extra}(\mathfrak{c})(\mathit{lead}) = m \text{ or } \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 0.\\
\mathit{snd\_ch}^m \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and}\\
&\quad\ \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 3 - m \text{ and}\\
&\quad\ \mathit{current}(\mathit{conf}_m(\mathfrak{c})) = \mathit{first}(\mathit{extra}(\mathfrak{c})(q)).\\
\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m} \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^m) < \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^{3-m}).
\end{aligned}$$

Program Model: Using the definitions of state, transition relation and atomic propositions above, we can now 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 $\mathcal{M}^{\delta}_{C,s,s'}$ *is defined as* $(\Sigma, \rightarrow_{\chi}, AP, \lambda, I)$ *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;*
- *$I = \{\langle C \parallel C', (s, s')\rangle\}$ 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.

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

Intuitively, 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 [6].

6 Temporal Logic Characterization of Scheduler-Specific Observational Determinism

Based on the results from the previous section, a temporal logic formula characterizing scheduler-specific observational determinism can be established. 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 Definitions of Atomic Propositions

We define atomic propositions that are used to instantiate the characterization of stuttering equivalence in different ways, so that we can characterize stuttering equivalence over low location traces, and over low store traces. For each $l \in L$, $\mathit{fst\_ch}_l^m$, $\mathit{snd\_ch}_l^m$, and ${\mathit{nr\_ch}_l}^m < \mathit{nr\_ch}_l^{3-m}$ relate to each low variable, and $\mathit{fst\_ch}_L^m$, $\mathit{snd\_ch}_L^m$, and ${\mathit{nr\_ch}_L}^m < \mathit{nr\_ch}_L^{3-m}$ relate to the set of low variables L, where $m = 1$ or 2.

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

6.2 Characterization of Scheduler-Specific Observational Determinism

Now we can give a temporal logic formula characterizing the properties of traces of a program that is observationally deterministic under a scheduler δ. 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 $\mathcal{M}^{\delta}_{C,s_1,s_2}$.

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

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

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

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

Thus, if the program has n low variables, we have $n + 1$ verification tasks, where 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 [6].

7 Conclusion

This paper presents a new formal definition of observational determinism that approximates the intuitive definition of observational determinism well. 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 our definition 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 our definition 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 [13, 7] for a more detailed description of noninterference, its verification, and a discussion why it is not appropriate for multi-threaded programs.

Roscoe [11] 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 [16, 7, 14] has been mentioned above. They all propose different formal definitions of observational determinism, with a corresponding verification method. Zdancewic and Myers propose a type system that requires that the type checked program must be confluent in order to be verified [14]. Terauchi also proposes a type system to verify observational determinism, but this one does not enforce confluence. 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) [7]. The idea of using self-composition was first proposed by Barthe et al. and Darvas et al. [1, 3]. The way self-composition is worked out here, with a temporal logic characterization also bears resemblance with temporal logic characterizations of strong bisimulation [9].

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 [12]. 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 will encode the characterization in one (or more) model checkers. An important challenge is to model the queue, as this can have a strong effect on the state space that has to be examined. 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 nondeterminism that is possible in an execution, but it does not consider the probability that an execution will happen. In a separate line of work, we will also study how probability can be used to guarantee secure information flow.

Acknowledgment: 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.

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.P. 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, New York, NY, USA, 2004. ACM.
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 *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, third edition*. Addison Wesley, 2005.
5. M. Huisman and H.-C. Blondeel. Model-checking secure information flow for multi-threaded programs. In *Theory of Security and Applications (Tosca)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
6. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. Available via `http://www.homeewi.utwente.nl/~ngominhtri/`, full version.
7. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterization of observation determinism. In *Computer Security Foundations Workshop*. IEEE Computer Society, 2006.
8. M. Huth and M. Ryan. *Logic in computer science: modeling and reasoning about the system*. Cambridge University Press, second edition, 2004.
9. 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.
10. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. In *Inf. Processing Letters*, volume 63, pages 243–246, 1997.
11. A.W. Roscoe. Csp and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.
12. A. Russo and A. Sabelfeld. Security interaction between threads and the scheduler. In *Computer Security Foundations Symposium*, pages 177–189, 2006.
13. A. Sabelfeld and A. Myers. Language-based information flow security. In *IEEE Journal on Selected Areas in Communications*, volume 21, pages 5–19, 2003.
14. T. Terauchi. A type system for observational determinism. In *Computer Science Foundations*, 2008.
15. T.V. Vleck. Timing channels. In *Poster session at IEEE TCSP conference*, 1990. Available via `http://www.multicians.org/timing-chn.html`.
16. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43. IEEE Press, June 2003.