Scheduler-related Confidentiality for Multi-threaded Programs

Marieke Huisman and Tri Minh Ngo

University of Twente, Netherlands
`Marike.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. Several formal definitions of observational determinism exist, but all of them have shortcomings; for example they accept insecure programs or they reject too many innocent programs. Besides, all the proposed definitions of observational determinism are not scheduler-independent: 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 a program behaves securely when the scheduling policy changes.
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.
Under the worst case assumption, i.e., where an attacker can choose the scheduler, the security specification should be scheduler-independent. Therefore, in addition, we propose a definition of scheduler-independent observational determinism that is robust with respect to any particular scheduling policy. Thus scheduler-independence means that if a program is accepted by a security specification then an attacker cannot derive any secret information from it, regardless of which scheduler is used.

1 Introduction

The success of applications, such as e.g. Internet banking and mobile code, depends for a large part on the kind of security guarantees that can be given to clients. In particular, confidentiality is important: if users have the feeling that their private data is not sufficiently protected, they will refuse to use these applications. Using formal means to establish confidentiality of such applications is a way to gain the trust of users. 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, one should not only consider input-output behavior, but whole execution traces.

In the literature, different definitions of confidentiality are proposed. For sequential programs, a classical approach is to establish a technical property called noninterference [4]. This states that a program is considered secure whenever the *final values of low variables* are independent of the *initial values of high variables*[1]. However, this definition of noninterference only considers the input-output behavior of a program, and as mentioned above, this is not appropriate for multi-threaded programs.

Instead, for multi-threaded programs, we have to put restrictions on the *execution traces*, i.e., the sequences of states that occur during program execution. Confidentiality for multi-threaded programs requires that the private data is never revealed throughout the execution trace. Different proposals exist that attempt to generalize noninterference to a multi-threaded setting. 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 [18]. In the literature, several formal definitions of observational determinism are proposed [18, 7, 15], but none of these capture exactly this intuitive definition.

The first formal definition of observational determinism was proposed by Zdancewic and Myers [18]. 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, any two low location traces are equivalent upto stuttering and prefixing, where a low location trace is the projection of a trace onto a single low variable location. Thus, the trace of each variable is considered separately. They motivate this choice by arguing that the relative order of two updates can only be observed by code that contains a data race and that data races should be avoided anyway. However, they use a non-standard definition of a data race, saying that there is a data race whenever two accesses to the same variable can happen in any order. In 2006, Huisman, Worah and Sunesen [7] showed that allowing prefixing of location traces can reveal secret information, even for sequential programs. They strengthened the definition of observational determinism by requiring that low location traces must be stuttering equivalent. In 2008, Terauchi showed that even in a program without data races (even in the sense of Zdancewic and Myers), an attacker can observe the relative order of two updates of the low variables, and derive secret information from this [15]. Therefore, he proposes another variant of observational determinism, requiring

[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.

that all low store traces should be stuttering and prefixing equivalent, thus not considering the variables independently.

However, we believe that also Terauchi's definition is 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 innocent 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. We show that all definitions of observational determinism proposed so far are not scheduler-independent, i.e., they accept programs that are insecure under some specific schedulers. Therefore, in this paper, we propose two new definitions of observational determinism that overcome these shortcomings. The definition of scheduler-specific observational determinism 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 variable, the 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 then no information can be deduced from it. This existential condition strongly connects to the used scheduler .

Besides, we consider it very important that the security of a given program should not critically depend on a particular scheduling policy; otherwise security guarantees may be destroyed by a slight change in the scheduling policy. Therefore, based on the definition of scheduler-specific observational determinism, we derive a scheduler-independent definition as well. This definition requires that the low store traces from initial stores s_1 and s_2 are stuttering equivalent. If a program is accepted by this security specification, it is secure under any scheduling policy.

In addition, we also discuss properties and limitations 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.

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 and limitations. In the next section, we propose a definition of observational determinism which is scheduler-independent. Finally, Section 6 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 $\|$. The program syntax is not used in the formalization of the 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.

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

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. [5]).

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. We use accessor functions *prog* and *store*, respectively. A store is the current state of the program memory by mapping a value to the location of each program variable. 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, ensuring that all traces are infinite. Thus, we assume that the attacker cannot detect termination.

$$\langle \epsilon, s\rangle \to \langle \epsilon, s\rangle$$

Given 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 $\langle C, s\rangle$, 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 traces resulting from the executions of a program C from a store s, i.e., $\mathit{Trace}(\langle C, s\rangle) = \{T | \langle C, s\rangle \Downarrow T\}$. Let $\mathit{Trace}^*(\langle C, s\rangle)$ denote the set of all finite prefixes of traces in $\mathit{Trace}(\langle C, s\rangle)$, i.e., $\mathit{Trace}^*(\langle C, s\rangle) = \{\pi | \pi \sqsubseteq T \wedge T \in \mathit{Trace}(\langle C, s\rangle)\}$ where $\sqsubseteq$ denotes the prefix relation on traces. Let $\mathit{last}(\pi)$ denote the last state of a finite trace π.

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, this is implicitly lifted to 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: $T_{|_L} = map(\_{|_L} \circ store)(T)$. 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. For completeness, we give the formal definitions of these equivalences.

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 $\exists i. T \sim T'_{\ll i} \vee T_{\ll i} \sim T'$.*

2.3 Scheduler

A multi-threaded program executes threads from a set of live threads. During the execution, a scheduling policy δ repeatedly decides probabilistically which thread shall be picked to proceed next with the computation. Different scheduling policies differ in how they make this decision, e.g., a uniform scheduler chooses threads randomly and hence all possible interleavings of threads are considered; a *round-robin* scheduler assigns equal time slices to each thread in circular order. They also differ in the information on which they base the decision, e.g., a uniform scheduler needs to know the number of current live threads while a round-robin scheduler, beyond knowing this information, also remembers their previous scheduling choices.

To cover various kinds of schedulers, we let scheduler be history-dependent, i.e., in addition to the current system state, scheduler's behavior also depends on the trace leading to that state. Let C be a program and s be an initial store of C. Given a finite trace $\pi \in Trace^*(\langle C, s\rangle)$, a scheduler δ which schedules the transition $last(\pi) \to c$ with probability $\delta(\pi)(c)$ is formally defined as follows.

Definition 3. *A scheduler δ for C starting at s is a function,*

$$\delta : Trace^*(\langle C, s\rangle) \to \mathcal{D}(Conf),$$

such that for all finite traces $\pi \in Trace^*(\langle C, s\rangle)$, if $\delta(\pi)(c) > 0$ then $last(\pi) \to c$.

Where $\mathcal{D}(X)$ denotes the set of all probability distributions over X. A probability distribution over a set X is a function μ that assigns a probability in $[0, 1]$ to each element of X, i.e., $\mu \in X \to [0, 1]$, such that the sum of the probabilities of all elements is 1, i.e., $\sum_{x \in X} \mu(x) = 1$. If $X = \{x_0, x_1, \ldots\}$, we often write the probability distribution μ as,

$$\mu(x)_{x \in X} = \{x_0 \mapsto \mu(x_0), x_1 \mapsto \mu(x_1), \ldots\},$$

or

$$\mu(x)_{x \in X} = \begin{cases} \mu(x_0) & \text{if } x = x_0 \\ \mu(x_1) & \text{if } x = x_1 \\ \vdots & \end{cases}$$

Furthermore, for $c \to \{c' \mapsto \mu(c')\}$, we simply write $c \to_p c'$ when μ is clear from the context and $\mu(c') = p$. This expression denotes that the probability of a transition from c to c' is p,

This model of scheduler is general enough to describe any scheduler that uses the full history of computation to generate probabilities for picking the threads. In the following, we formally describe the behaviors of some popular scheduling policies, such as uniform scheduler and round robin scheduler.

From the program of the last configuration $last(\pi)$, it is possible to derive the number of current live threads. We store this number in a scheduler variable $nthr$. We also assume that active threads are implicitly numbered consecutively by their positions in the set of live threads, e.g., from 0 to $(nthr - 1)$.

Given $last(\pi) = \langle C_0 \parallel C_1 \parallel \cdots \parallel C_{nthr-1}, s\rangle$, the uniform scheduler $\delta_{\text{Uni}}(\pi)$ is simply,

$$\delta_{\text{Uni}}(\pi)(c) = \mu_{\text{Uni}}(c) = \begin{cases} \frac{1}{nthr} & \text{if } c = \langle C_0 \parallel \cdots \parallel C_i' \parallel \cdots \parallel C_{nthr-1}, s\rangle \\ 0 & \text{otherwise} \end{cases}$$

for any $i \in \{0 \ldots nthr - 1\}$. This means that $last(\pi) \to_{\frac{1}{nthr}} \langle C_0 \parallel \cdots \parallel C_i' \parallel \cdots \parallel C_{nthr-1}, s\rangle$, in which the thread C_i is picked and changed to C_i'.

Let the scheduler variables thr_{pre} denote the position of the executed thread in $last(\pi)$ and $nstp$ denote how many consecutive steps this thread has been picked. A round-robin scheduler which chooses the same thread for m steps before switching to the next thread can be defined as follows,

1. $nstp < m$ and the thread at position thr_{pre} does not terminate,

$$\delta_{\text{Roundr}}(\pi)(c) = \mu_{\text{Roundr}}(c) = \begin{cases} 1 & \text{if } c = \langle C_0 \parallel \cdots \parallel C'_{thr_{pre}} \parallel \cdots \parallel C_{nthr-1}, s\rangle \\ 0 & \text{otherwise} \end{cases}$$

2. $nstp < m$ and the thread at position thr_{pre} terminates, or $nstp = m$

$$\delta_{\text{Roundr}}(\pi)(c) = \mu_{\text{Roundr}}(c) = \begin{cases} 1 & \text{if } c = \langle C_0 \parallel \cdots \parallel C'_{(thr_{pre}+1) \bmod nthr} \parallel \cdots \parallel C_{nthr-1}, s\rangle \\ 0 & \text{otherwise} \end{cases}$$

In this report, if the scheduler is not mentioned explicitly, a uniform scheduler is assumed.

Because a program is always executed under the control of a specific scheduling policy δ, we replace the notation $c_i \rightarrow_p c_j$ by $c_i \rightarrow_{\delta,p} c_j$, which means that the probability of a transition from c_i to c_j under the scheduling policy δ is p. We can write $c_i \rightarrow_p c_j$ whenever δ is clear from the context. We use notation $C_1 \parallel_p C_2$ to denote that the probability of the next transition corresponding to a transition of C_1 is p. We just simply write $C_1 \parallel C_2$ when $p = 1/2$.

We parameterize $\langle C, s\rangle \Downarrow T$ by $\langle C, s\rangle \Downarrow_\delta T$. We write $\langle C, s\rangle \Downarrow T$ when a uniform scheduler is used. Let $Trace_\delta(\langle C, s\rangle)$ denote the set of traces resulting from the executions of a program C from a store s under the scheduling policy δ. Let $Trace^*_\delta(\langle C, s\rangle)$ denote the set of all finite prefixes of traces in $Trace_\delta(\langle C, s\rangle)$. Notice that $Trace_\delta(\langle C, s\rangle) \subseteq Trace(\langle C, s\rangle)$.

3 Observational Determinism in the Literature

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

3.1 Existing Definitions of Observational Determinism

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

– Zdancewic and Myers [18]: iff any two low location traces are equivalent upto stuttering and prefixing.

$$\forall T, T'.\ \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ \ \forall l \in L.\ T_{|_l} \sim_p T'_{|_l}.$$

– Huisman et al. [7]: iff any two low location traces are equivalent upto stuttering.

$$\forall T, T'.\, \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ \forall l \in L.\, T_{|_l} \sim T'_{|_l}.$$

– Terauchi [15]: iff any two low store traces are equivalent upto stuttering and prefixing.

$$\forall T, T'.\, \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ T_{|_L} \sim_p T'_{|_L}.$$

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, 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. We believe that allowing prefixing can reveal secret information. Further, as observed by Terauchi, attackers can derive secret information from the relative order of updates. However, 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. Besides, all these definitions are not scheduler-independent: an accepted program can be insecure under a specific scheduler (as they are all expressed over a nondeterministic scheduler only). These shortcomings are illustrated below by several examples. In all examples, we assume the observational model is that attackers can access the full code of the program, observe the traces of public data, and even possibly limit the set of possible program traces by choosing an appropriate scheduler.

How prefixing equivalences can reveal information Consider the following program (from [7]). Suppose `h` $\in H$ and `l1`, `l2` $\in L$.

Example 1.

```
l1 := 0; l2 := 0;
while (h > 0) do {l1 := l1 + 1; h := h − 1};
l2 := 1;
```

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.

Case `h` $= 1 : T_{|_L} = [(0,0),(1,0),(1,1)]$
Case `h` $= 2 : T_{|_L} = [(0,0),(1,0),(2,0),(2,1)]$
Case `h` $= 3 : T_{|_L} = [(0,0),(1,0),(2,0),(3,0),(3,1)]$
Case `h` $= 4 : T_{|_L} = [(0,0),(1,0),\cdots,(4,0),(4,1)]$

As the low location traces of this program are stuttering and prefixing equivalent, this program is observationally deterministic according to Zdancewic and Myers. However, this program is *insecure* because the final value of `l1` reveals the initial value of `h` (attackers can access the full code of the program). This illustrates that allowing equivalence upto prefixing can reveal secret information.

Terauchi strengthens the definition of Zdancewic and Myers by requiring that the traces need to agree on the low stores as a whole, instead of just individual locations. Therefore, Example 1 is rejected by Terauchi. However, also with Terauchi's definition, partial information still can be leaked because of prefixing, as illustrated by the following example.

Example 2.

```
l1 := 0; l2 := 0;
{if (l1 == 1) then (l2 := h) else skip} || l1 := 1
```

where `h` is a boolean.

Let C_1 and C_2 denote the left and right operands of the parallel composition operator. If we execute this program, we obtain store traces of the following shapes.

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

Thus, when `h` $= 1$, we can terminate in a state where `l2` $= 1$. However, when `h` $= 0$, we can only terminate in a state where `l2` $= 0$. The low store traces generated by this program are equivalent upto stuttering and prefixing. Thus, according to the definitions of Zdancewic and Myers, and Terauchi, this program is observationally deterministic. However, this program is not secure by the fact that when the value of `l2` changes, an attacker can conclude that surely `h` $= 1$.

How the relative order of updates can reveal information Zdancewic and Myers (followed by Huisman et al.) consider each low variable location independently because they argue that internal observations of the two variable locations can only observe the relative order of their updates by using code that contains races. Zdancewic and Myers define a race to happen when two accesses to the same location are unordered. As we mentioned before, this is a non-standard definition of data race, and in addition, this argument about the relative order of updates is not correct. Consider the following program (from Terauchi [15]).

Example 3.

```
l1 := 0; l2 := 0;
if (h > 0) then {l1 := 1; l2 := 1}
           else {l2 := 1; l1 := 1}
```

This program is sequential, and thus does not contain any races. However, an attacker can still observe the relative order of two updates. To illustrate this, if we execute this program, we get executions of the following shapes.

$$\begin{array}{l}\text{Case } \mathtt{h} > 0 : T_{|L} = [(0,0),(1,0),(1,1)] \\ \text{Case } \mathtt{h} \leq 0 : T_{|L} = [(0,0),(0,1),(1,1)]\end{array}$$

Attackers can learn information about `h` by observing whether `l1` is updated before `l2`. Thus this program is *insecure*. This shows that it is not sufficient to require that only the low location traces are deterministic for a program to be secure. Terauchi solves this by strengthening the definition of observational determinism to low store traces, but this results in an overly restrictive definition, as illustrated next.

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 4.

```
l1 := 0; l2 := 0;
l1 := 3 ∥ l2 := 4
```

Let C_1 be "`l1 := 3`", and C_2 "`l2 := 4`". If we execute this program, we get traces of the following shapes.

$$T_{|L} = \begin{cases} [(0,0),(3,0),(3,4)] \text{ execute } C_1 \text{ first} \\ [(0,0),(0,4),(3,4)] \text{ execute } C_2 \text{ first} \end{cases}$$

Thus 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 The security of a program depends strongly on the choice of scheduler. In all examples given so far, a nondeterministic scheduler is assumed. However, the scheduler may vary from execution to execution. Under a specific scheduling policy, some traces *cannot occur*. From the limited set of possible traces, sometimes secret information can be derived. This can be exploited by an attacker that can choose the scheduler. 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 5.

```
l := 0;
{{if (h > 0) then sleep(n)}; l := 1} ∥ l := 0
```

where `sleep(n)` abbreviates `n` consecutive *skip* commands.

Let C_1 and C_2 denote the left and right operands of the parallel composition operator. 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., it 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.

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

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

Thus, with this scheduling policy, when $\mathtt{h} \le 0$, we can terminate in a state where $\mathtt{l} = 0$ or $\mathtt{l} = 1$. However, when $\mathtt{h} > 0$, we can only terminate in a state where $\mathtt{l} = 1$ because the round-robin scheduler will not let C_1 finish the `sleep` command before giving the turn to C_2. Thus, the final value of `l` reveals whether `h` is positive or not. This program shows that Zdancewic and Myers, and Terauchi's definitions are not scheduler-independent.

In this example, there is an encoding of a *timing leak* into an implicit flow. Hence, this attack is also called *internal observable timing* attack [18, 14, 12]. In a multi-threaded program, information can be derived from observing the internal timing of actions. Thus, an attacker does not need access to a clock to learn information about the private data, which makes this attack highly dangerous. Often a timing leak does not manifest itself when a scheduler is completely nondeterministic, but only when a more deterministic scheduling policy is used.

Notice that under a nondeterministic scheduler, the initial value of `h` cannot be derived. However, it is actually desirable that this program is rejected because `l` is not deterministic. This program is similar to the program "$\mathtt{l} := \mathtt{0} \parallel \mathtt{l} := \mathtt{1}$". In the literature it has been shown how nondeterminism of a low variable can be exploited to make other programs reveal information. Suppose one of the two assignments falls on the same cache line as a piece of data 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, and in this way it can be used to derive information about the confidential data, see [18, 16].

However, the program "$\mathtt{l1} := \mathtt{3} \parallel \mathtt{l2} := \mathtt{4}$" in Example 4 is considered secure because it writes to two different locations.

Due to the fact that an attacker knows the full code of the program, when he chooses an appropriate scheduler, he can limit the set of outputs, and then derive secret information. The following example shows that the definition of Huisman et al. is also not scheduler-independent.

Example 6.

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

This program is secure under a nondeterministic scheduler. An attacker can not derive secret information from the observation of public variables because the changes of each low location does not depend on the high variable h. In addition, secret information cannot be derived from the observation of relative order of updates because whether l1 or l2 is updated first does not depend on the value of h. 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.

$$\begin{array}{l} \text{Case } \mathtt{h} > 0 : T_{|_L} = [(0,0),(1,0),(1,1),\ldots] \\ \text{Case } \mathtt{h} \leq 0 : T_{|_L} = [(0,0),(0,1),(1,1),\ldots] \end{array}$$

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. The order of updates helps the attacker derive information about h.

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-independent. They accept insecure programs under some scheduling policies. 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 harmless programs w.r.t. a particular scheduler. Based on this definition, we derive a scheduler-independent definition which is robust with respect to any scheduler.

4 Scheduler-specific Observational Determinism

This section defines observational determinism w.r.t. a particular scheduler. The next section will then discuss how this definition is used to defined scheduler-independent observational determinism. To overcome the problems discussed above, we adapt the existing definitions. 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.

Given a scheduling policy σ, scheduler-specific observational determinism is defined formally as follows.

Definition 4 (σ-specific observational determinism).

*A program C is σ-*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 are satisfied.*

- $\forall T, T'. \langle C, s_1 \rangle \Downarrow_\sigma T \wedge \langle C, s_2 \rangle \Downarrow_\sigma T' \Rightarrow \forall l \in L. T_{|_l} \sim T'_{|_l}$.
- $\forall T. \langle C, s_1 \rangle \Downarrow_\sigma T. \exists T'. \langle C, s_2 \rangle \Downarrow_\sigma T' \wedge T_{|_L} \sim T'_{|_L}$.

We require that low location traces evolve deterministically, and in addition, there always exists a matching relative order of updates. Notice that this definition does not allow information to be leaked because of prefixing equivalence. Moreover, it 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 relates strongly to the used scheduling policy because traces model possible runs of a program under that scheduling policy and refinements of the set of traces, when the scheduling policy changes, cannot guarantee this condition.

4.1 Properties of Scheduler-specific Observational Determinism

To illustrate that Definition 4 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 4, 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. Therefore, an attacker cannot derive any secret information from the individual low location traces.

Property 2 (Deterministic relative order of updates). If a program is accepted by Definition 4, no information can be derived from the relative order of updates because there is always a matching low store trace.

Properties 1 and 2 prevent an attacker from deriving secret information from the observation of the public location traces and the relative order of updates. Thus, all insecure programs in Examples 1,2, 3, and 5 are rejected by our definition under the nondeterministic scheduling policy.

The program in Example 6 is secure under a nondeterministic scheduler and it is accepted by our definition instantiated accordingly. However, it is insecure under the scheduler which 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 σ, if a program is accepted by our definition, instantiated for this scheduler, we can conclude that the program is secure under the scheduling policy σ.

Property 3 (Less restrictive on harmless programs). Compared with Terauchi's definition, Definition 4 accepts more innocent programs.

For example, Example 4, which is secure, is accepted by our definition instantiated with a nondeterministic scheduler, but rejected by Terauchi. Similarly, Example 6 is also accepted by our definition instantiated with a nondeterministic scheduler, but rejected by Terauchi. This illustrates that our definition is more permissive: it allows some freedom in the order of individual updates, as long as a matching execution exists.

Even though the properties above illustrate that Definition 4 captures observational determinism well, there are still some restrictions. In particular, Definition 4 cannot distinguish between a nondeterministic change of a low location variable (not depending on private data) and a change that does depend on private data. For example, the program "`l := 0` ‖ `l := 1`" will be rejected by Definition 4 under a nondeterministic scheduler. However, notice again that it is actually desirable that this program is rejected.

Thus, we have the following property for Definition 4

Property 4 (All low non-deterministic programs are rejected).

However, notice that this does not mean that Example 4, which is considered secure, is rejected. Notice further that the nondeterminism property also applies to all earlier definitions of observational determinism.

5 Scheduler-independent Observational Determinism

The next question is whether there exists a scheduling policy σ and a security specification such that if a program is accepted by this security specification, it is secure under any scheduling policy. This motivates the notion of *scheduler-independent* observational determinism.

Execution of a program under a nondeterministic scheduler means that we consider all possible interleavings of threads. Any scheduling policy σ is a refinement of a nondeterministic scheduler; thus the set of possible program traces under the scheduling policy σ is a subset of the set of possible program traces under a nondeterministic scheduler. If we quantify Definition 4 over all 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. For example, the program "`l := 3` ‖ `h := 5` ‖ `skip`" is secure, regardless of which scheduler is used.

Therefore, the formal definition of scheduler-independent observational determinism can be stated as follows,

Definition 5 (Scheduler-independent observational determinism).
A program C is scheduler-independent observationally deterministic w.r.t. L *iff for all initial low equivalent stores s_1, s_2, $s_1 =_L s_2$, the following condition is satisfied.*

$$\forall T, T'.\ \langle C, s_1\rangle \Downarrow T \wedge \langle C, s_2\rangle \Downarrow T' \Rightarrow\ T_{|_L} \sim T'_{|_L}.$$

The following theorem states that if a program is accepted by Definition 5, it is also σ-specific observationally deterministic under any scheduling policy σ.

Theorem 1. *For any two initial low equivalent stores, if any two store traces obtained from the execution of a program under a nondeterministic scheduler are stuttering equivalent, this program is secure under any scheduling policy.*

Proof. Any scheduling policy is a refinement of a nondeterministic scheduler, and by Definition 5, any two low store traces are stuttering equivalent. Therefore, any two low location traces are stuttering equivalent and any low store trace can be matched (because all low store traces are stuttering equivalent). Thus, all conditions of Definition 4 are respected and the program is σ-specific observationally deterministic.

6 Conclusion

This paper presents a new formal definition of scheduler-specific observational determinism that approximates the intuitive definition of observational observation well. If a program is accepted under a specific scheduler, no secret information can be derived from the publicly observable location traces or the relative order of updates. We also argue that our definition accepts a large class of secure programs.

In the worst case, i.e., in the case of refinement attacks where an attacker can choose a scheduler, the security specification should be scheduler-independent. Therefore, this paper also proposes a definition of scheduler-independent observational determinism. This is derived from the scheduler-specific definition by quantifying over all possible security policies. If a program is accepted by the scheduler-independent security condition, no private information can be derived from it, regardless of which scheduler is used.

6.1 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 [18, 7, 15] has been mentioned above. They all propose different formal definitions of observational determinism, with a corresponding verification method.

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. However, when a scheduler's behavior is *probabilistic*, some threads might be executed more often than others, which opens up the possibility of a probabilistic attack. In order to cope with such attacks, the theory of probabilistic noninterference has been developed [17, 14]. In particular, Sabelfeld and Sands [14] develop a probabilistic noninterference criterion, based on a partial probabilistic low bisimulation which is an adaptation of Larsen and Skou's notion of probabilistic 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.

6.2 Future Work

As future work, we will develop a way to verify adherence to the new definition of observational determinism. 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 will exploit this idea in a similar way as in our earlier work [7, 6] 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 believe that our definition can be characterized by LTL [8] and CTL logics [8]. For both logics, good model checkers exist that we can use to verify the information flow property. We will encode the characterizations of observational determinism in one (or more) model checkers.

In a separate line of work, we will also study how probabilism can be used to guarantee secure information flow. In particular, we plan to study whether the definition of probabilistic noninterference of Sabelfeld and Sands [14] is satisfactory. If it is not, then we will propose a new definition of probabilistic noninterference and also a method to verify it on a multi-threaded program.

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 (NWO).

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. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–22. IEEE Press, 1982.
5. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, third edition.* Addison Wesley, 2005.
6. 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.
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. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Information and Computation*, volume 94, pages 1–28, 1992.
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. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop*, pages 200–214, 2000.
15. T. Terauchi. A type system for observational determinism. In *Computer Science Foundations*, 2008.
16. T.V. Vleck. Timing channels. In *Poster session at IEEE TCSP conference*, 1990. Available via `http://www.multicians.org/timing-chn.html`.
17. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Journal of Computer Security*, volume 7, pages 231–253, 1999.
18. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43. IEEE Press, June 2003.