Noninterfering Schedulers

When Possibilistic Noninterference Implies Probabilistic Noninterference

Andrei Popescu Johannes Hölzl Tobias Nipkow

Technische Universität München

Abstract. We develop a framework for expressing and analyzing the behavior of probabilistic schedulers. There, we define noninterfering schedulers by a probabilistic interpretation of Goguen and Meseguer's seminal notion of noninterference. Noninterfering schedulers are proved to be safe in the following sense: if a multi-threaded program is possibilistically noninterfering, then it is also probabilistically noninterfering when run under this scheduler.

1 Introduction

Noninterference is an important and well-studied formal property modeling *confidentiality*. It was introduced by Goguen and Meseguer (henceforth abbreviated G&M) in the context of deterministic multi-user systems having essentially the following meaning [5, p.11]: "One group of users is noninterfering with another group of users if what the first group does has no effect on what the second group of users can see."

In the context of confidentiality in a language-based setting [16], a quite different notion, usually also termed as noninterference, emerged in work by Volpano et. al. [23]: Assuming the program memory is separated into a *low*, or public, part, which an attacker is able to observe, and a *high*, or private, part, hidden to the attacker, a *sequential* program satisfies noninterference if, upon running it, the high part of the initial memory does not affect the low part of the resulting memory.

Of course, many systems for which confidentiality is important are *concurrent*, such as Internet servers or operating systems. To cope with concurrency, the above language-based notion of noninterference has been generalized in various ways. A major line of work focuses on *possibilistic noninterference*, which roughly states that if an execution allowing certain observations by the attacker is possible, then another execution for which these observations are infirmed is also possible. For this notion, powerful and/or compositional analysis methods have been devised [1, 3, 8, 20]. The downside of possibilistic noninterference is vulnerability under *probabilistic attacks* by running the program multiple times and gathering statistical information and *refinement attacks* via knowledge of the thread scheduling. For example, consider the following program consisting of two threads running in parallel under a uniform probabilistic scheduler [18]:

- while $h > 0$ do $\{h := h - 1\}$; $l := 2$
- $l := 1$

Then, probability to execute $l := 2$ after $l := 1$, i.e., to obtain 2 for the final l, depends on the initial value of h, making the latter inferable from the distribution of the final l.

These problems have been addressed by introducing several (overlapping) notions of *probabilistic* [9, 18, 19] and *scheduler-independent* [9, 17, 24] noninterference and means to enforce them. Proposed scheduler-independent solutions (probabilistic or not)

insure confidentiality in the presence of any scheduler [16, 17, 20] or a large class of schedulers [4, 9, 13], but suffer from various limitations: lack of coping with dynamic thread creation [4], too harsh requirements on individual threads (strong security) [17, 20], too weak confidentiality guarantees on the overall concurrent system [9], the reliance on expensive or not always feasible conditions such as race freedom [24] or termination [9], or non-standard thread-level security primitives [13].

This paper presents a way to alleviate these limitations in a scheduler-dedicated framework. Its main contributions are:

- A framework for analyzing schedulers independently from the concrete operational semantics of threads.
- A notion of noninterfering scheduler obtained by a novel reading, in a probabilistic key, of G&M's seminal notion.
- A result inferring probabilistic noninterference from possibilistic noninterference under the assumption of a noninterfering scheduler (for suitable notions of possibilistic and probabilistic noninterference).

This result captures a large class of schedulers, covers dynamic thread creation, allows timing of thread execution to depend on high data, guarantees a strong security property on the thread system, and does not rely on undecidable properties of the multi-threaded program or special security primitives. Our scheduler noninterference, importing insights from system-based noninterference to language-based noninterference, is a step toward better understanding the complex relationship between these two worlds [7].

We start by introducing the framework for schedulers (§2), carefully factoring in *all* and *only* the information relevant to scheduling. Thus, in order to have fine control over the scheduling policy including dynamic thread creation, we keep an order on thread IDs that indicate who spawned who. On the other hand, for studying the behavior of a scheduler we do not employ concrete thread pools with state-based semantics for threads—instead, we consider *execution scenarios*, i.e., possibilistic interleavings of threads IDs to which the scheduler casts probabilities.

Operational semantics of multi-threaded programs (§3) is separated in two: The *possibilistic semantics* is the usual nondeterministic interleaving semantics; in particular, it yields an execution scenario for each pair (program, initial state). A *probabilistic semantics* is obtained by blending in a scheduler with the execution scenario.

Then we move to discussing noninterference (§4). For defining scheduler noninterference, we identify two groups of users à la G&M: the threads that are *visible*, i.e., will eventually affect the observable part of the system during their execution, and the others, the *invisible* ones. The user's *actions* are, as expected, the very steps taken by the threads. The *observations*, however, require a nonstandard interpretation: Given a visible thread v and letting I denote the collection of invisible threads, the observation of v is the "exit probability" of v through I, i.e., the probability of the system taking zero or more I-steps followed by a v-step. We call the scheduler noninterfering if the observation of each visible thread v is independent of the actions of the invisible threads I, i.e., the exit probability of v through I is the same as the probability of taking v provided I were completely inexistent (including from the execution history).

For *possibilistic* noninterference of programs, we adopt a compositional notion introduced in [9], a weakening of strong security [17] allowing the execution time to

depend on secrets. In fact, our approach as a whole disregards execution time. We take *probabilistic* noninterference to be the notion introduced by Smith [18]: Any two executions that differ only on secret information traverse the same sequence of attacker observations with the same probability—this seems to be the strongest notion of probabilistic noninterference that ignores timing channels.

Further details on the constructions and results from this paper, including more substantial proof sketches, can be found in the technical report [10], which is an identical copy of the paper save for an appendix with additional material.

2 Framework for schedulers

This section introduces the key component of our approach: a framework for studying schedulers in isolation from the concrete (state-based) operational semantics.

In the noninterference literature (e.g., [9, 17, 18]), the thread IDs manipulated by schedulers are typically handled implicitly, as the numeric indexes (positions) of the threads in the pool represented as a list. However, here we endow thread IDs with more structure, able to store information about the parent thread and the order in which the current threads have been spawned (§2.1). We introduce *histories*, i.e., sequences of thread IDs taken so far during the execution, and *rich histories* obtained from augmenting the histories with information about the threads that were available at each point in history—these enriched structures offer useful information concerning the thread waiting time (§2.2). Schedulers are defined as operating on rich histories (§2.3). In order to study scheduler noninterfernce in isolation from a concrete operational semantics, we single out the aspect of thread semantics relevant for the scheduler's behavior: *execution scenarios*, as trees of thread ID interleavings (§2.4). Given an execution scenario, a scheduler induces a Markov chain structure on histories, offering a quantitative interpretation of temporal logic formulas (§2.5) useful later for defining noninterference.

2.1 Thread IDs

We let i, j, k, l range over natural numbers. The *thread IDs*, ranged over by m, n, p, q, will be elements of the set $\mathbf{threadID} = \mathbf{nat}^*$, of words (i.e., finite sequences) of natural numbers. The empty sequence ε will represent the main thread's ID. We write $m{\cdot}n$ for the concatenation of m and n. As usual, we identify single numbers k with singleton words, and thus $k{\cdot}m$ and $m{\cdot}k$ represent the words obtained by pre-appending and post-appending k to m, respectively.

For all $m \in \mathbf{threadID}$, we define the set of IDs that *m may spawn*, $\mathsf{maySp}\ m$, as $\{m{\cdot}k \mid k \in \mathbf{nat}\}$. The full reading of "$n \in \mathsf{maySp}\ m$" is the following: "if m is the ID of a given thread, then n is valid as ID for a thread the given thread may spawn (in the future)". Note that $\forall n.\ \varepsilon \notin \mathsf{maySp}\ n$, i.e., no thread may spawn the main thread.

We also define, for each $m \in \mathbf{threadID}$, the following order $<_m$ on $\mathsf{maySp}\ m$, called the *m-issuing order*: $m{\cdot}k <_m m{\cdot}j$ iff $k < j$. The reading of "$p <_m q$" is "the thread ID p should be generated before q is (in any potential execution)". For example, it holds that $2{\cdot}1 \in \mathsf{maySp}\ 2$ and $2{\cdot}1{\cdot}1 <_{2\cdot1} 2{\cdot}1{\cdot}2$.

The "may spawn" operator and the issuing orders will be means to inform the scheduler about who spawned who and about the order in which spawning happened. In our informal explanations, we shall loosely identify threads with thread IDs.

2.2 Histories

Our schedulers will depend on execution histories indicated as lists of thread IDs. Since on the other hand thread IDs are themselves modeled as lists (sequences), to avoid confusion we use a different notation for lists of threads.

Namely, we let **hist**, the set of *(execution) histories*, ranged over ml, nl, pl, ql, consist of thread ID lists. $[m_0, \ldots, m_{k-1}]$ denotes the history consisting of the indicated thread IDs in the indicated order. $[]$ is the empty history and $[m]$ is a singleton history. $ml \# nl$ denotes the concatenation of histories ml and nl, and we write $ml \# n$ and $n \# ml$ instead of $ml \# [n]$ and $[n] \# ml$, respectively. If $ml = [m_0, \ldots, m_{k-1}]$, $ml\langle ..i\rangle$ is the subhistory of ml containing the first i elements, $[m_0, \ldots, m_{i-1}]$; thus, $ml\langle ..0\rangle = []$ and $ml\langle ..k\rangle = ml$.

Given $n \in \mathbf{threadID}$, $N \subseteq \mathsf{maySp}\ n$ and $M \subseteq \mathbf{threadID}$, M is called an *initial fragment of N w.r.t.* $<_n$ if $M \subseteq N$ and $\forall m \in M.\ \forall m' \in N \setminus M.\ m <_n m'$.

We shall be interested in the relationship between execution histories and the sets of available threads at each point in such histories. We let Ml range over lists of finite sets of thread IDs. A pair (ml, Ml) where $ml = [m_0, \ldots, m_{k-1}]$ and $Ml = [M_0, \ldots, M_k]$ (thus having $\mathsf{length}\ Ml = \mathsf{length}\ ml + 1$) is said to be:

- *start-consistent*, if $M_0 = \{\varepsilon\}$;
- *step-consistent*, if $\forall i < k.\ m_i \in M_i$;
- *termination-consistent*, if $\forall i < k.\ M_i \setminus M_{i+1} \subseteq \{m_i\}$;
- *spawn-consistent*, if $\forall i < k.\ M_{i+1} \setminus M_i$ is an initial fragment of $\mathsf{maySp}\ m_i \setminus (M_0 \cup \ldots \cup M_i)$ w.r.t. $<_{m_i}$.

The above conditions describe the correct interplay between the threads available in the pool at each moment (represented by Ml) and single execution steps taken by the threads (represented by ml). More precisely, we assume that, at moment i, M_i are the available threads and m_i takes a step, yielding the available threads M_{i+1}.

Start consistency: Execution starts with the main thread alone in the thread pool.

Step consistency: Only an available thread can take a step.

Termination consistency: Upon a step taken by thread m_i, the resulted thread pool contains all threads except perhaps m_i (if terminated).

Spawn consistency: Upon a step taken by thread m_i, all newly appearing threads in the pool get IDs that m_i may spawn and, moreover, they get the smallest such IDs that are *fresh*, in the sense of not having been assigned before.

Note that start consistency and step consistency imply that $m_0 = \varepsilon$. A pair (ml, Ml) is called a *rich history* if it is start-, step-, termination-, and spawn- consistent. We let **rhist** denote the set of rich histories.

Rich histories $(ml = [m_0, \ldots, m_{k-1}], Ml = [M_0, \ldots, M_k])$ contain enough information to determine various moments in the life span of threads:

- The set of *current threads*, $\mathsf{Cur}\ Ml$, is the last element in this list, M_k.
- Given $n \in \mathsf{Cur}\ Ml \setminus \{\varepsilon\}$, the *moment when n appeared*, $\mathsf{app}_{Ml}\ n$, is the smallest i such that $n \in M_{i+1}$.
- Given $n \in \{m_0, \ldots, m_{k-1}\}$, the *moment when n was last taken (executed)*, $\mathsf{ltaken}_{ml}\ n$, is the greatest $i < k$ such that $n = m_i$.
- Given $n \in \mathsf{Cur}\ Ml$, the *moment when n was last touched*, $\mathsf{ltouched}_{ml,Ml}\ n$, is: either $\mathsf{ltaken}_{ml}\ n$, if $n \in \{m_0, \ldots, m_{k-1}\}$; or $\mathsf{app}_{Ml}\ n$, otherwise.
- Given $n \in \mathsf{Cur}\ Ml$, the *waiting time for n*, $\mathsf{wait}_{ml,Ml}\ n$, is $k - 1 - \mathsf{ltouched}_{ml}\ n$.

Note that, if both $\mathsf{app}_{Ml}\ n$ and $\mathsf{ltaken}_{ml}\ n$ are defined, i.e., if $n \in \mathsf{Cur}\ Ml \setminus \{\varepsilon\} \cap \{m_0,\ldots,m_{k-1}\}$, then, by step-consistency, $\mathsf{app}_{Ml}\ n < \mathsf{ltaken}_{ml}\ n$—this justifies our definition for $\mathsf{ltouched}_{ml,Ml}\ n$.

2.3 Schedulers

A *scheduler* is a family of functions $(sch_{ml,Ml} : \mathsf{Cur}\ Ml \to \mathbb{R})_{ml,Ml}$, where (ml, Ml) ranges over rich histories, such that $\forall m \in \mathsf{Cur}\ Ml.\ sch_{ml,Ml}\ m \geq 0$ and $\sum_{m \in \mathsf{Cur}\ Ml} sch_{ml,Ml}\ m = 1$. Thus, given a rich history (ml, Ml), a scheduler defines a probability distribution $sch_{ml,Ml}$ on the currently available threads $\mathsf{Cur}\ Ml$. Next we give two standard examples. (See [10, §A] for several others.)

Uniform scheduler. usch assigns all currently available threads equal (history oblivious) probability: $\mathsf{usch}_{ml,Ml}\ m = 1/|\mathsf{Cur}\ Ml|$. Uniform scheduling is the underlying assumption in work by Smith and Volpano on probabilistic noninterference [18, 19, 22].

Round robin scheduler. Given a number j, the round robin scheduler with j step quotas, rsch^j, always schedules with probability 1 the first thread in the queue for j consecutive steps, where threads are ordered in a queue according to their waiting time.

Given (ml, Ml), we define the following *queuing order* on M: $n <_{ml,Ml} n'$ iff

- either $\mathsf{wait}_{ml,Ml}\ n < \mathsf{wait}_{ml,Ml}\ n'$,
- or $\mathsf{wait}_{ml,Ml}\ n = \mathsf{wait}_{ml,Ml}\ n'$ and $n' \in \mathsf{maySp}\ n$,
- or else $n, n' \in \mathsf{maySp}\ p$ and $n' <_p n$ for some p.

$<_{ml,Ml}$ organizes the current thread pool M as a queue based on waiting times, resolving same-waiting-time conflicts as follows: a spawned thread has priority over its parent, two threads spawned at the same time are discriminated by the issuing order. The first (maximum) in this waiting queue, $\max_{ml,Ml} M$, is in the set of threads with highest waiting time and, among these, is the smallest w.r.t. the “may spawn" and issuing orders.

For any history ml, we let $\$(ml)$ be the number of trailing occurrences of its last thread, i.e., the largest number k such that ml has the form $nl \# m^k$, where m^k consists of k repetitions of m. We define rsch^j, the *j-step round robin scheduler*, as follows, for all $(ml, Ml) \in \textbf{rhist}$ and $p \in M = \mathsf{Cur}\ Ml$:

– If $ml = []$, then necessarily $Ml = \{\varepsilon\}$, $M = \{\varepsilon\}$ and $p = \varepsilon$. We put $\mathsf{rsch}^j_{ml,Ml}\ p = 1$.
– If ml has the form $nl \# m$, then we define

$$\mathsf{rsch}^j_{ml,Ml}\ p = \begin{cases} 1, \text{ if } \$(ml) < j \wedge p = m, & \text{(last scheduled thread } m \text{ still has quota)} \\ 1, \text{ if } \$(ml) \geq j \wedge p = \max_{ml,Ml} M, & (m \text{ finished its quota, } p \text{ comes next}) \\ 1, \text{ if } m \notin M \wedge p = \max_{ml,Ml} M, & (m \text{ has terminated, } p \text{ comes next}) \\ 0, \text{ otherwise.} & (p \text{ neither current, nor next to be scheduled}) \end{cases}$$

Previous work on concurrent noninterference [9, 12, 14, 15, 17] considers round robin schedulers almost equivalent to our rsch^j, except for the policy of placing in the pool the newly spawned threads. Namely, while defining the operational semantics of the thread pools modeled as lists of threads, the newly spawned threads are inserted in the list *after* the parent thread. This, together with the policy of the scheduler tape traversing the thread pool from left to right, makes the scheduler non-starvation-free—e.g., if m spawns in one step an identical copy of itself, that copy would be scheduled immediately after m, and thus m and its clones would monopolize execution. Our definition based on the waiting time avoids this problem. Of course, the problem is also

solvable by changing the operational semantics to traverse the thread list from right to left instead. However, this solution reveals a limitation of approaches that hardwire in the thread-pool operational semantics the policy for placing new threads: the need for global changes in order to accommodate desired scheduler properties. By contrast, our approach packs up the whole scheduler behavior in the definition of the scheduler alone.

2.4 Execution scenarios

Although a scheduler depends on rich histories which are essentially *linear* structures, its behavior is better comprehended through what we call execution scenarios, tree-like structures that capture the *branching* of thread interleaving. Given any set H of histories and given $ml = [m_0, \ldots, m_{k-1}] \in H$ such that all its prefixes are also in H:

- Let $\mathsf{Avail}^H\ ml$, the set of thread IDs *available in H at point ml*, be $\{m \in \mathbf{threadID} \mid ml \# m \in H\}$;
- Let $\mathsf{Havail}^H\ ml$, the list of sets of thread IDs *available in H all throughout history ml*, be $[\mathsf{Avail}^H\ ml\langle..0\rangle, \ldots, \mathsf{Avail}^H\ ml\langle..k\rangle]$.
- Given $m \in \mathsf{Avail}^H\ ml$, let $\mathsf{spawns}^H_{ml}\ m$, the *set of threads spawned by one m-step at history ml*, be $\mathsf{Avail}^H\ (ml \# m) \setminus \mathsf{Avail}^H\ ml$.

An *(execution) scenario* is a set Sc of histories such that the following properties hold, where $\preceq$ is the prefix order on lists:

- Prefix Closure: $\forall ml\ nl.\ nl \in Sc \wedge ml \preceq nl \Longrightarrow ml \in Sc$.
- Finite Branching: $\forall ml \in Sc.\ \mathsf{Avail}^{Sc}\ ml$ is finite.
- Consistency: $\forall ml \in Sc.\ (ml, \mathsf{Havail}^{Sc}\ ml) \in \mathbf{rhist}$.
- Boundedness: $\exists k.\ \forall ml \in Sc.\ \forall m \in \mathsf{Avail}^{Sc}\ ml.\ |\mathsf{spawns}^{Sc}_{ml}\ m| \leq k$.

Thus, a scenario is required to form a finitely branching tree for which all finite paths are rich histories and there exists a bound on the number of threads spawned concurrently in one single step. The best way to picture a scenario is as a labeled tree, where the nodes are histories ml (with $[]$ as the root), and the edges coming out of each ml are labeled with the elements of $\mathsf{Avail}^{Sc}\ ml$. For example, if we ignore the circled numbers for now, Fig. 1 shows the finite scenario $Sc = \{[], [\varepsilon], [\varepsilon,0], [\varepsilon,1], [\varepsilon,0,1], [\varepsilon,1,0], [\varepsilon,1,1{\cdot}0], [\varepsilon,0,1,1], [\varepsilon,1,0,1{\cdot}0], [\varepsilon,1,1{\cdot}0,0], [\varepsilon,1,0,1{\cdot}0,1{\cdot}0]\}$. In this scenario, the following happen (among other things): at history $[]$, ε takes one step and terminates, with spawning two threads, 0 and 1 (hence $\mathsf{spawns}^{Sc}_{[]}\ \varepsilon = \{0,1\}$)—this can be seen from the branchings of $[]$ and $[\varepsilon]$: ε is available at history $[]$, while 0 and 1 are available at the successor history $[\varepsilon]$; at history $[\varepsilon]$, after taking one step, 1 terminates, spawning a new thread $1{\cdot}0$; at history $[\varepsilon,0]$, 1 takes two steps and terminates, without spawning any threads.

Note that, in accordance with termination consistency, the described scenario never abandons execution, but proceeds until termination is plausible. E.g., from its appearance (at history $[\varepsilon]$), on any path thread 0 is continuously available before it is taken.

2.5 Scheduler-induced probabilities on scenarios

Given a scenario Sc, a scheduler sch assigns probabilities to branches in Sc—at history ml, the branch $m \in \mathsf{Avail}^{Sc}\ ml$ receives probability $sch_{ml,\ \mathrm{Havail}^{Sc}\ ml}\ m$—and then to finite paths in Sc as the product of probabilities of the branches taken along the path. Fig. 1 shows in circles a possible such assignment of probabilities to a scenario, where, e.g., the history path $[\varepsilon,1,1{\cdot}0]$ has probability $1 * 0.5 * 0.75 = 0.375$.

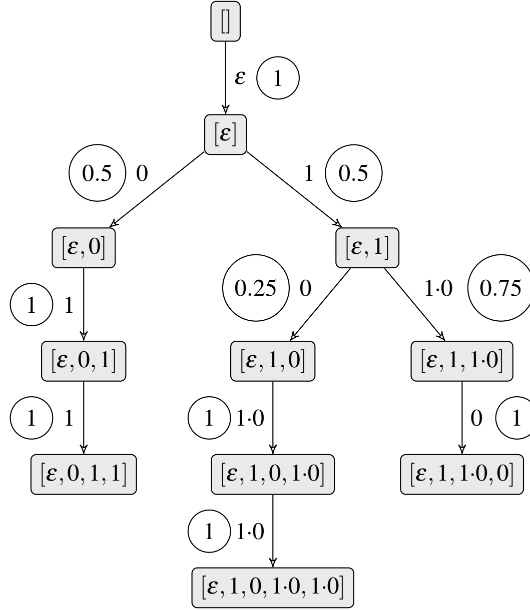

Fig. 1. A scenario with probabilities attached

More generally, let $pl \in Sc$. We let Trace^{Sc}_{pl} be the set of (Sc,pl)-*traces*, which are maximal (finite or infinite) sequences *mt* such that $pl \,\#\, ml \in Sc$ for all finite prefixes *ml* of *mt*. Then we can identify each *ml* such that $pl \,\#\, ml \in Sc$ with a "basic event" $\mathsf{Bev}^{Sc}_{pl,ml}$ consisting of all (Sc,pl)-traces that start with *ml*, i.e., have *ml* as a prefix; we thus postulate that $\mathsf{Bev}^{Sc}_{pl,ml}$ has the probability of *ml* when taken in history *pl*. E.g., in Fig. 1, $\mathsf{Bev}^{Sc}_{[\varepsilon],[1]}$ consists of $\{[1,0,1{\cdot}0,1{\cdot}0],\ [1,1{\cdot}0,0]\}$ and has probability 0.5.

By standard probability theory [6], one can now assign probabilities $\mathsf{P}^{Sc,sch}_{pl}\ Mt$ to certain measurable sets *Mt* of (Sc,pl)-traces, namely, to those in the smallest collection of subsets of Trace^{Sc}_{pl} that is closed under countable union and complement and contains every $\mathsf{Bev}^{Sc}_{pl,ml}$ for which $pl \,\#\, ml \in Sc$. The *Markov chain induced by sch on Sc*, Mc^{sch}_{Sc}, is the family $(\mathsf{Trace}^{Sc}_{pl}, \mathsf{P}^{Sc,sch}_{pl})_{pl \in Sc}$.

The sets of traces describable in linear temporal logic (LTL) are measurable [21]. Thus, to each LTL formula φ, for each history point $pl \in Sc$, we can speak of the (Sc,sch)-*probability of* φ, written $\mathsf{P}^{Sc,sch}_{pl}\ \varphi$ and defined as the probability of the set of (Sc,pl)-traces satisfying φ. Of particular importance for us will be the following LTL formulas and connectives, where U : **hist** $\rightarrow$ **threadID** $\rightarrow$ **bool**, $n \in$ **threadID** and φ and χ are any LTL formulas:

$\mathsf{Takes}\ U$, satisfied by a (Sc,pl)-trace iff that trace takes as first step an element m such that of $U\ pl\ m$ holds.

$\mathsf{Ev}\ \varphi$, satisfied by a trace iff φ eventually holds on some point on that trace.

$\mathsf{Alw}\ \varphi$, satisfied by a trace iff φ always holds (on every point) on that trace.

$\varphi\ \mathsf{Until}\ \chi$, satisfied by a trace iff φ holds on every point on some finite initial fragment of that trace, and χ holds immediately after. (This is the LTL "strong until".)

If U simply tests for equality to a fixed thread n, i.e., $\forall ml\, m.\, U\, ml\, m \iff m = n$, we write $\mathsf{Takes}\, n$ instead of $\mathsf{Takes}\, U$. Note that $\mathsf{Ev}\, (\mathsf{Takes}\, n)$ is satisfied by a trace iff that trace contains n, and $(\mathsf{Takes}\, U)\, \mathsf{Until}\, (\mathsf{Takes}\, n)$ is satisfied by a trace iff that trace takes for a while steps for which U holds, and eventually it takes n.

3 Operational semantics of programs

Next we introduce the state-based small-step semantics, both possibilistic and probabilistic, for shared-memory multi-threaded programs featuring dynamic thread creation.

3.1 Possibilistic semantics

Let **state**, ranged over by s,t, be an unspecified set of memory states. We assume that the individual threads are commands $c,d \in \mathbf{cmd}$ with a semantics given by a transition relation $c \xrightarrow{s} (\gamma, [c_1,\ldots,c_l], s')$, where γ is either $\bot$ or a command c', having the following interpretation: in state s, c takes one step, spawning threads $c_1,\ldots,c_l$, changing the state to s', and: terminating, provided $\gamma = \bot$, or yielding the continuation c', provided γ is a command c'. We assume the transition relation to be total and deterministic, i.e., for all c and s there exists a unique pair $(\gamma, [c_1,\ldots,c_l])$ such that $c \xrightarrow{s} (\gamma, [c_1,\ldots,c_l])$. Also, we assume that each command c is spawn-bounded, in that there exists k (depending on c) such that the number of threads spawned *in one single step* by c or any of its continuations or spawned threads during execution is $\le k$—this is a reasonable assumption for programs written in a concurrent language, where k can be determined by inspecting the syntax. (Spawn-boundedness has an obvious coinductive definition that we omit.)

A *(runtime) configuration* is a tuple $cf = (ml, Ml, thr, s)$ such that (ml, Ml) is a rich history and $thr : \mathsf{Cur}\, Ml \to \mathbf{cmd}$. (ml, Ml) indicates the execution so far, thr the assignment of commands to thread IDs, s the current memory state. We define a labeled transition relation on configurations: $(ml, Ml = [M_0,\ldots,M_k], thr, s) \xrightarrow{m} (ml', Ml', thr', s')$ iff $m \in M_k$ and the following hold, assuming $thr\, m \xrightarrow{s} (\gamma, [c_1,\ldots,c_l], s')$ and letting $p_1 <_m \ldots <_m p_l$ be the first l smallest thread IDs in $\mathsf{maySp}\, m \setminus (M_0 \cup \ldots \cup M_k)$ w.r.t. $<_m$:

- $ml' = ml \# m$.
- $Ml' = Ml \# M'$, where $M' = \begin{cases} M_k \setminus \{m\} \cup \{p_1,\ldots,p_l\}, & \text{if } \gamma = \bot, \\ M_k \cup \{p_1,\ldots,p_l\}, & \text{otherwise.} \end{cases}$
- thr' behaves like thr on elements of $M' \cap M_k$ and additionally sends each p_i to c_i.

The above is the expected one-step semantics of configurations: any currently available thread may take a (possibly terminating) step, spawning 0 or more new threads that are assigned the smallest available thread IDs, and affecting the state; in case of termination, the thread is removed from the pool.

We define $cf \xrightarrow{[m_1,\ldots,m_k]} cf'$ to mean that there exist $cf_0,\ldots,cf_{k-1}$ such that $cf_0 = cf$, $cf_{k-1} = cf'$, and $cf_i \xrightarrow{m_{i+1}} cf_{i+1}$ for all $i < k$.

3.2 From possibilistic to probabilistic semantics, via schedulers

Given c and s, let the *initial configuration of* (c,s), $\mathsf{init}\,(c,s)$, be $([], [\{\varepsilon\}], \varepsilon \mapsto c, s)$. Thus, in $\mathsf{init}\,(c,s)$, c is the single (main) thread and s the current state; during execution, c may of course spawn other threads that will populate the configuration. We define $\mathsf{Sc}_{c,s}$, the

scenario of (c,s), to be $\{ml.\ \exists cf.\ \mathsf{init}\ (c,s) \stackrel{ml}{\to} cf'\}$—that Sc is indeed a scenario follows immediately from the definition of configuration transitions.

Note that, for each $ml \in \mathsf{Sc}_{c,s}$, there exists precisely one $cf = (ml, Ml, thr, s)$ such that $\mathsf{init}\ (c,s) \stackrel{ml}{\to} cf$—we write $\mathsf{config}_{c,s}\ ml$ for this cf. Thus, the pair $(\mathsf{Sc}_{c,s}, \mathsf{config}_{c,s})$ constitutes an alternative description of the *possibilistic* semantics of (c,s) (including complete information about thread spawning and termination). If we also factor in the Markov chain induced by sch on $\mathsf{Sc}_{c,s}$, we obtain a proper notion of *probabilistic* semantics of (c,s) as the triple $(\mathsf{Sc}_{c,s}, \mathsf{config}_{c,s}, \mathsf{Mc}^{sch}_{c,s})$, where we write $\mathsf{Mc}^{sch}_{c,s}$ instead of $\mathsf{Mc}^{sch}_{\mathsf{Sc}_{c,s}}$. We shall also write $\mathsf{Trace}^{c,s}$ and $\mathsf{P}^{sch,c,s}$ instead of $\mathsf{Trace}^{\mathsf{Sc}_{c,s}}$ and $\mathsf{P}^{\mathsf{Sc}_{c,s},sch}$.

4 Noninterference

Here we present our main security result: a notion of noninterfering scheduler that ensures lifting of possibilistic noninterference to probabilistic noninterference. All throughout this section, we fix a scheduler sch and a domain **odom** of observables.

4.1 Noninterfering schedulers

An *observation-augmented scenario* (*OA-scenario*) is a pair (Sc, obs), where $obs : Sc \to$ **odom**. Let (Sc, obs) be an OA-scenario. A thread n is called *visible* at a certain history if it is available and, at some point in the future, n will affect the observables, either directly or indirectly via a spawned thread n', or via a thread spawned by n', etc. Formally, we define inductively the sets $\mathsf{visAvail}^{Sc,obs}\, ml$ of *visible threads available at ml*:

$$\frac{n \in \mathsf{Avail}^{Sc}\, ml \quad obs\ (ml \# n) \neq obs\ ml}{n \in \mathsf{visAvail}^{Sc,obs}\, ml} \qquad \frac{m, n \in \mathsf{Avail}^{Sc}\, ml \quad n \in \mathsf{visAvail}^{Sc,obs}\, (ml \# m)}{n \in \mathsf{visAvail}^{Sc,obs}\, ml}$$

$$\frac{n \in \mathsf{Avail}^{Sc,obs}\, ml \quad n' \in \mathsf{spawns}^{Sc}_{ml}\, n \quad n' \in \mathsf{visAvail}^{Sc,obs}\, (ml \# n)}{n \in \mathsf{visAvail}^{Sc,obs}\, ml}$$

An available thread m is called *invisible* if it is not visible—formally, the predicate $\mathsf{inv}^{Sc,obs}_{ml}\, m$ is defined to mean $m \in \mathsf{Avail}^{Sc,obs}\, ml \setminus \mathsf{visAvail}^{Sc,obs}\, ml$.

A scheduler shall be declared noninterfering if the effect of removing invisible threads is the same as that of hiding them. This property can be formulated in a manner rather faithful to the style of G&M [5] (recalled in the introduction). Our “users" of the system managed by sch are the threads (thread IDs), and at each history point there are two groups of users, visible and invisible, and thus we require that the *observations* of the visible threads do not depend on the *actions* of the invisible ones. Clearly, the actions should be steps taken by the threads. Moreover, we choose the observation of a visible user n at history ml to be the probability that n will be scheduled first among all the visible threads, i.e., the “exit probability" of n after zero or more invisible steps, $\mathsf{P}^{Sc,sch}_{ml}\ (\mathsf{Takes\ inv}^{Sc,obs}\ \mathsf{Until\ Takes}\ n)$. Note that here, unlike in [5], current users may disappear (by termination) and new users may appear (by spawning), and therefore $\mathsf{inv}^{Sc,obs}$ is not a fixed set, but a set evolving over time; this is properly handled by the history-dependent interpretation of temporal formulas.

Having the observations and the actions in place, it is time to zoom in the definition of noninterference from [5] in more technical detail: For all users n of the second group (here, the visible threads), the observation of n based on the history (where “history" means, in [5] as well as here, “the sequence of actions the users have taken in the

past") is required to be the same as the observation of n on the restriction of the history by removing all actions of users from the first group (here, the invisible threads). In order to formally perform this removal, i.e., filter out the (Sc, obs)-invisible actions from histories, we first define recursively $\mathsf{visHist}^{Sc,obs}\ ml$, the visible restriction of a history ml:

$$\mathsf{visHist}^{Sc,obs}[\,] = [\,] \qquad \mathsf{visHist}^{Sc,obs}(ml \,\#\, m) = \begin{cases} (\mathsf{visHist}^{Sc,obs}\ ml) \,\#\, m, & \text{if } m \in \mathsf{visAvail}^{Sc,obs}\ ml, \\ \mathsf{visHist}^{Sc,obs}\ ml, & \text{otherwise.} \end{cases}$$

Moreover, we collect the available visible threads throughout history ml in the set $\mathsf{visHavail}^{Sc,obs}\ ml$ also defined recursively:

$$\mathsf{visHavail}^{Sc,obs}\ [\,] = [\{\varepsilon\}]$$

$$\mathsf{visHavail}^{Sc,obs}\ (ml \,\#\, m) = \begin{cases} (\mathsf{visHavail}^{Sc,obs}\ ml) \,\#\, (\mathsf{visAvail}^{Sc,obs}\ ml), & \text{if } m \in \mathsf{visAvail}^{Sc,obs}\ ml, \\ \mathsf{visHavail}^{Sc,obs}\ ml, & \text{otherwise.} \end{cases}$$

The scheduler sch is called *noninterfering* if the following holds for all OA-scenarios (Sc, obs), all $ml \in Sc$, and all $n \in \mathsf{visAvail}^{Sc,obs}\ ml$:

$$\mathsf{P}^{Sc,sch}_{ml}\left(\mathsf{Takes\ inv}^{obs,Sc}\ \mathsf{Until\ Takes}\ n\right) = sch_{ml',Ml'}\ n,$$

where $(ml', Ml') = (\mathsf{visHist}^{Sc,obs}\ ml, \mathsf{visHavail}^{Sc,obs}\ ml)$.

In the above equality, the lefthand side expresses the observation made by n at history ml, and the righthand side the observation that n would make if any trace of invisible threads were removed (from both the history and the available threads). Since our notion of observation effectively hides invisible threads (in the style of τ-actions form process algebra), the meaning of the above equality can be summarized as

Removal = Hiding (of invisible threads)

Note that the notion of scheduler noninterference is independent of the concrete notion of command at the expense of quantifying universally over all scenarios.

An important question is whether a reasonable class of schedulers are noninterfering. Roughly speaking, any scheduler that is “politically correct", treating its threads uniformly, is noninterfering.

Proposition 1 The uniform and round-robin schedulers from §2.3 are noninterfering.

Proof idea. We fix (Sc, obs) and $ml \in Sc$. We need to show the equality of two functions defined on $n \in \mathsf{visAvail}\ ml$, say $F = G$, where $F\ n = \mathsf{P}^{Sc,sch}_{ml}\left(\mathsf{Takes\ inv}^{obs,Sc}\ \mathsf{Until\ Takes}\ n\right)$ and $G\,n = sch_{ml',Ml'}\ n$.

For usch, noninterference follows immediately from its symmetry, since both F and G are constant on $\mathsf{visAvail}\ ml$. For rsch^j, let m be the last thread in ml and $k = \$(ml)$. If m is visible and $k < j$, then both $F\ n$ and $G\,n$ are either 1, if $n = m$, or 0, otherwise. If m is invisible or $k \geq j$, then both $F\ n$ and $G\,n$ are either 1, if n is the next visible thread in the queue, or 0, otherwise. □

Several other noninterfering schedulers are presented in [10, §A]. It is also instructive to see an *interfering* one: Consider a modification of the round robin that increments the quota at each shift to a new thread. Then consider the history $ml = [n_1, m, m, n_2, n_2]$ with n_1, n_2 visible and m invisible. Since n_2 still has one step in its quota, $F\ n_2 = 1$. On the other hand, $ml' = [n_1, n_2, n_2]$, meaning that, at ml', n_2 yields to n_1, hence $G\,n_2 = 0$.

4.2 Possibilistic noninterference

To discuss noninterference of commands, we fix an attacker-observation function aobs : **state** $\rightarrow$ **odom**. A typical choice of aobs [23] assumes the state consists of values stored

in variables classified as high-security or low-security and defines aobs to return the low-variable part of the state (see [10, §D]). We define possibilistic noninterference by a form of bisimilarity up to invisibility.

Invisibility of a command is defined as "never change the observation on the state", technically, coinductively as the weakest predicate invis satisfying the following property: for all $c, s, \gamma, c_1, \ldots, c_l, s'$ such that $\mathsf{invis}\ c$ and $c \xrightarrow{s} (\gamma, [c_1, \ldots, c_l], s')$, we have that: **(1)** $\mathsf{aobs}\ s' = \mathsf{aobs}\ s$, **(2)** $\mathsf{invis}\ c_i$ for all $i \in \{1, \ldots, k\}$; **(3)** $\gamma \in \mathbf{cmd}$ implies $\mathsf{invis}\ \gamma$.

Possibilistic bisimilarity of two commands is now defined coinductively as the weakest relation $\approx$ satisfying the following property: for all c, d, if $c \approx d$, then either $\mathsf{invis}\ c$ and $\mathsf{invis}\ d$ or, for all $s, t, \gamma, c_1, \ldots, c_l, s', \delta, d_1, \ldots, d_k, t'$ such that $\mathsf{aobs}\ s = \mathsf{aobs}\ t$, $c \xrightarrow{s} (\gamma, [c_1, \ldots, c_l], s')$ and $d \xrightarrow{t} (\delta, [d_1, \ldots, d_k], t')$, we have that: **(1)** $\mathsf{aobs}\ s' = \mathsf{aobs}\ t'$; **(2)** $l = k$ and $c_i \approx d_i$ for all $i \in \{1, \ldots, l\}$; **(3)** if $\gamma = \bot$, then either $\delta = \bot$ or $\mathsf{invis}\ \delta$; **(4)** if $\delta = \bot$, then either $\gamma = \bot$ or $\mathsf{invis}\ \gamma$; **(5)** if $\gamma, \delta \in \mathbf{cmd}$, then $\gamma \approx \delta$.

A command c is called *possibilistically noninterfering* if $c \approx c$. Thus, possibilistic noninterference of c means that alternative executions starting in states indistinguishable by the attacker proceed in a synchronized manner for as long as one of them does not reach an invisible status, moment at which the other is required to also reach such a status; moreover, termination should be matched by either termination or invisibility.

The componentwise extension of this notion to thread pools coincides with the flexible scheduler-independent security introduced by Mantel and Sudbrock in [9]. As argued in [9], this notion is both compositional and flexible enough to allow the execution time of programs to depend on secrets. However, it does share the common limitation of PER approaches [15] aimed at scheduler independence: its rather strong lock-step synchronization nature (albeit only on visible executions).

Note that the example from the introduction does not satisfy possibilistic noninterference since, depending on the initial value of h, one alternative execution may enable the visible action $l := 2$ earlier than another alternative execution. And indeed, our intention with possibilistic noninterference is to guard (in the presence of noninterfering schedulers) against probabilistic attacks of the kind allowed by this program—this will be our main result, Th. 2.

4.3 Probabilistic noninterference

We define probabilistic noninterference following the weak bisimulation approach taken by Smith [18], using an adaptation of a corresponding notion from probabilistic process algebra due to Baier and Hermanns [2]: Roughly, a command shall be deemed probabilistically noninterfering if any two executions of it starting in states that differ only on secret information traverse the same sequence of attacker observations with the same probabilities. As argued in [18, p.11], this notion is suitable for protecting against internal leaks, but not external leaks such as timing.

In our formalism, we can define everything in terms of scenarios and their scheduler-induced Markov chains. Indeed, the function $\mathsf{config}_{c,s}$ introduced in §3.2 "observes", at each execution history ml, the whole thread pool configuration. The attacker's observations on execution histories shall be much more restricted: only the state can be observed, and only through aobs. Namely, assuming $\mathsf{config}_{c,s}\, ml = (ml, Ml, thr, t)$, we define $\mathsf{obs}_{c,s}\ ml = \mathsf{aobs}\ t$. Thus, the OA-scenario $(Sc_{c,s}, \mathsf{obs}_{c,s})$ is a description of the executions of command c starting in state s, as observed by the attacker.

Given $H, H' \subseteq Sc$ and $ml \in Sc_{c,s}$, we define $ml \Rightarrow_H H'$ to be the set of all traces that go through elements of H only and eventually reach an element of H', namely,

$$\{mt \in \mathsf{Trace}_{c,s}.\ \exists nl'.\ [] \neq nl' \preceq mt \ \wedge\ (\forall nl \prec nl'.\ ml \mathbin{\#} nl \in H) \ \wedge\ ml \mathbin{\#} nl' \in H'\},$$

where $\prec$ and $\preceq$ denote the strict and nonstrict prefix orderings on finite or infinite sequences. Note that $ml \Rightarrow_H H'$ is empty unless $ml \in H$.

Given an equivalence relation E, Cls_E denotes its set of equivalence classes, which we simply call *E-classes*. Let (Sc, obs) be an OA-scenario. A relation $E : Sc_{c,s} \to Sc_{c,s} \to \textbf{bool}$ is called a *sch-probabilistic bisimulation* for (c,s) if the following hold:
(I1) E is an equivalence relation on $Sc_{c,s}$ with countable set of equivalence classes.
(I2) For $ml, nl \in Sc_{c,s}$, $E\ ml\ nl$ implies $\mathsf{obs}_{c,s}\ ml = \mathsf{obs}_{c,s}\ nl$.
(I3) For distinct E-classes H, H' and $ml, nl \in H$, $\mathsf{P}^{sch,c,s}_{ml}(ml \Rightarrow_H H') = \mathsf{P}^{sch,c,s}_{nl}(nl \Rightarrow_H H')$.

Thanks to condition (I3), we can define, for any two distinct E-classes H and H', $\mathsf{P}^{sch,c,s}(H \Rightarrow H')$, *the probability of moving from H directly to H' (without visiting any other E-class)*, to be $\mathsf{P}^{sch,c,s}_{ml}(ml \Rightarrow_H H')$ for some (any) $ml \in H$. Thus, a probabilistic bisimulation E provides a class partition of the scenario (I1) so that elements of the same class are indistinguishable both w.r.t. observations (I2) and probabilistic behavior (I3). By (I2), an attacker is only able to observe the sequence of E-classes induced by an execution; by (I3), this sequence is statistically the same (modulo repetition) regardless of the concrete E-class representatives.

We also define a binary version of this indistinguishability relation. (c,s) and (c',s') are called *sch-probabilistically bisimilar* if there exist E, E', F such that:

(1) E and E' are *sch*-probabilistic bisimulations for (c,s) and (c',s'), respectively.
(2) $F : \mathsf{Cls}_E \to \mathsf{Cls}_{E'}$ is a bijection such that
  (a) $\mathsf{obs}_{c,s}\ ml = \mathsf{obs}_{c',s'}\ ml'$ for all ml, ml', H with $ml \in H \in \mathsf{Cls}_E$ and $ml' \in F\ H$,
  (b) $\mathsf{P}^{sch,c,s}(H \Rightarrow H_1) = \mathsf{P}^{sch,c',s'}(F\,H \Rightarrow F\,H_1)$ for all $H, H_1 \in \mathsf{Cls}_E$,
  (c) $F\,H_0 = H'_0$, where H_0 and H'_0 are the equivalence classes of $[]$ in Cls_E and $\mathsf{Cls}_{E'}$.

Finally, c is called *sch-probabilistically noninterfering* if (c,s) and (c,s') are *sch*-probabilistically bisimilar for all s, s' such that $\mathsf{aobs}\ s = \mathsf{aobs}\ s'$.

4.4 Noninterference criterion

We can now state our main result connecting three concepts that were defined mutually independently: possibilistic and probabilistic noninterference of commands and noninterference of schedulers.

Theorem 2 If *sch* is noninterfering and c is possibilistically noninterfering, then c is *sch*-probabilistically noninterfering.

Proof idea. The key of the proof consists of the definition, for any OA-scenario, of its visible sub-OA-scenario obtained from removing, at each history, the currently invisible threads. The latter can be proven probablistically bisimilar to the original OA-scenario, the bisimilarity step being handled using the noninterference of *sch*. Moreover, from the noninterference of c, it follows that $Sc_{c,s}$ and $Sc_{c,s'}$ have the same visible sub-OA-scenario Sc', which makes them probabilistically bisimilar. (See [10, §E].) □

Next we discuss the security requirements and guarantees of this theorem.

Requirement 1 (R1): Scheduler noninterference. This is a background condition that needs to be verified for the scheduler once and for all. Its verification involves quantitative computation with probabilities. However, it is a natural condition expressing a certain symmetry of the scheduler, and its verification tends to be easy for the examples considered in §2.3 (as well as for other examples described in [10, §A]).

Requirement 2 (R2): Possibilistic noninterference. Unlike R2, this condition needs to be verified for each individual program. Fortunately, this style of PER properties is amenable for compositional verification [8, 15, 17, 18]. In particular, the type systems from [3, 4, 9, 18], as well as the harsher ones from [17, 20], are static criteria on multi-threaded programs enforcing this property.

Guarantee (G): Probabilistic noninterference. This appears to be the strongest security guarantee of a probabilistic system provided we ignore timing channels [18]: An attacker making observations of the low part of the memory while the program by multiple running cannot infer any secret, not even by statistically from multiple runs.

In order to further comprehend (G), let us have a look at a consequence in terms of end-to-end security. Given c, s and $S \in \mathbf{odom}$, we define $\mathsf{endUpIn}_{c,s}\, S \subseteq \mathsf{Trace}_{[]}^{sch,c,s}$ as the set of traces that eventually "end up in S", i.e., that eventually reach a point where the attacker observation becomes S and stays constantly S—in LTL, this set is described by the formula $\mathsf{Ev}\ (\mathsf{Alw}\ \mathsf{obs}_{c,s}^{-1})$. Note that the traces in $\mathsf{endUpIn}_{c,s}\, S$ need not be terminating.

Proposition 3 If c is *sch*-probabilistically noninterfering, then, for all c, s, s', S, $\mathsf{aobs}\ s = \mathsf{aobs}\ s'$ implies $\mathsf{P}_{[]}^{sch,c,s}\ (\mathsf{endUpIn}_{c,s}\, S) = \mathsf{P}_{[]}^{sch,c,s'}\ (\mathsf{endUpIn}_{c,s'}\, S)$.

The guarantee of Prop. 3 is that executions starting in indistinguishable states stabilize in any given attacker-indistinguishable class S with the same probabilities. Note that termination implies stabilization (but not vice versa), so in particular Prop. 3 says that if the two executions terminate, then the resulted states have the same probability distribution w.r.t. what the attacker can see.

Example. We assume that programs are specified in a simple while language with thread-spawning facilities, states are assignments of values to variables, variables are classified into low and high, and the attacker observation is the low part of the state. Consider the following multi-threaded program adapted from [9, §5.2]:

$$\mathsf{while\ True\ do}\ \{l_1 := \mathit{inp}_1\ ;\ l_2 := \mathit{inp}_2\ ;\ \mathsf{spawn}\ T\ ;\ \mathsf{spawn}\ T_1\ ;\ \mathsf{spawn}\ T_2\}$$

where T is $h := \mathsf{addH}(l_1, h)$, T_1 is $l := \mathsf{addL}(l_1, l)$, and T_2 is $l := \mathsf{addL}(l_2, l)$.

The program repeatedly performs the following actions: It receives two public values (through input channels modeled here as low variables inp_1 and inp_2 assumed to be volatile) and stores them in the low variables l_1 and l_2. Then it spawns three threads, T, T_1, T_2. T applies the non-atomic operation addH for updating a private database h with l_1, whose timing depends on the value of h. T_1 and T_2 apply the atomic operation addL for updating a public database l with l_1 and l_2, respectively.

This is an intuitively secure program w.r.t. time-insensitive attacks: regardless of the values of the low variables, the execution of the main thread takes the same path, repetitively spawning copies of T_1, T_2 (that assign low to low) and T (that assigns low to high); the execution of T does depend on h, but this is harmless, since T does not affect the low part of the state or the behavior of the other threads. The program is automatically checked to be possibilistically noninterfering by existing type systems

[9, 18] (see also [10, §D]). Our Th. 2 ensures that it is also noninterfering if run under any noninterfering scheduler, in particular, the uniform and round robin ones.

This was a simple example of a kind widely encountered in web computing and operating systems: nonterminating multi-threaded programs providing a form of service. However, it is not proved noninterfering by previous scheduler-independent criteria. In particular, it does not satisfy strong security [17] (since the running time T may depend on secrets) or observational determinism [24] (since T_1 and T_2 are in a data race). Due to nontermination, it also falls outside the scope of the criterion from [9].

5 Conclusions and related work

In this paper, we proposed a novel notion of scheduler noninterference, which was proved to behave securely w.r.t. refinement of nondeterminsim: possibilistic noninterference of the multi-threaded program implies probabilistic noninterference when run under the given scheduler. We have *not* introduced novel notions of possibilistic or probabilistic noninterference, but used (minor adaptations of) existing ones [9, 18]. Consequently, we can employ existing syntactic methods for verifying that programs satisfy the hypothesis of our main result, Th. 2.

Mantel and Sudbrock [9] define flexible scheduler-independent (FSI) security, which we use as our possibilistic noninterference. They also introduce the class of *robust* schedulers and they prove an end-to-end security property, in the style of our Prop. 3, but conditioned by termination of the program. As already discussed, a major improvement of our Th. 2 is freeness from the termination assumption which is both hard to check and often not true. Even ignoring termination, the security guarantee of Th. 2 is significantly stronger than that of [9], as it takes into account the whole sequence of attacker observations throughout execution, and not only at the end of it. Another difference between our setting and [9] is the considered class of schedulers. Like our noninterfering schedulers, the robust schedulers were shown to include the round robin and uniform ones. However, robust schedulers are introduced via a probabilistic simulation relation involving both the scheduler and FSI-secure thread pools. Our noninterference condition for schedulers has a more natural justification in terms of G&M noninterference and is stated in isolation from the concrete operational semantics of threads (although it does employ thread ID interleavings); arguably, it is also easier to check. On the other hand, the notion of scheduler from [9] allows the flexibility of an arbitrary scheduler state—we could not have employed the history-based G&M noninterference had we worked with such general schedulers.

Smith [18] defines probabilistic noninterference via weak probabilistic bisimulation and provides a type system criterion for it, assuming the uniform scheduler. Since the guarantee of Th. 2 is precisely Smith's probabilistic noninterference, our result is in effect a generalization of his results to a wide class of schedulers.

Sabelfeld and Sands [17] introduce *strong security* for thread pools (a PER notion requiring complete lock-step synchronization of alternative executions) and prove security w.r.t. *all* schedulers; moreover, Sabelfeld [15] proves that strong security cannot be weakened if we are after a *compositional* notion covering the whole class of schedulers. Zdancewic and Myers [24] take a whole different approach to scheduler independence, focusing on concurrent programs that are a priori safe under refinement

attacks, in that the attacker's sequence of observations is the same in any execution (observational determinism). This is achieved practically by a data race freedom analysis in conjunction with a type system. Boudol and Castellani [4] describe yet another approach, based on an operational semantics for the scheduler, run in parallel with a thread pool that it controls. They do not cover probabilistic schedulers or dynamic thread creation. Finally, Russo and Sabelfeld [13] achieve scheduler independence by allowing the threads to explicitly change their security levels and the scheduler to discriminate between threads according to their levels. [13, §2] and [9, §6] survey more work on scheduler-independent security. Unlike here, previous work [9, 17] allows schedulers to depend on the low part of the state. This is also possible in our framework and is pursued in [10, §A], but here it has been omitted as it brings no further insight into our method.

This paper was concerned with lifting possibilistic noninterference to probabilistic noninterference. Somewhat complementary, our previous work [11] studies and classifies various notions of possibilistic noninterference and their compositionality w.r.t. language constructs.

Acknowledgment. We thank Jasmin Blanchette and the referees for useful comments and suggestions. This work was supported by the DFG project Ni 491/13–2, part of the DFG priority program Reliably Secure Software Systems (RS^3).

References

1. J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000.
2. C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *CAV*, pages 119–130, 1997.
3. G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP*, pages 382–395, 2001.
4. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.
5. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
6. J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov chains (second edition)*. Springer, 1976.
7. H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *CSFW*, pages 200–214, 2001.
8. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF 2001*, pages 218–232, 2011.
9. H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *ESORICS*, pages 116–133, 2010.
10. A. Popescu, J. Hölzl, and T. Nipkow. Noninterfering schedulers—when posibilistic noninterference implies probabilistic noninterference. Technical report. Available at `http://www4.in.tum.de/~popescua/sched.pdf`.
11. A. Popescu, J. Hölzl, and T. Nipkow. Proving concurrent noninterference. In *CPP*, pages 109–125, 2012.
12. A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *ASIAN 2006*, volume 4435 of *LNCS*, pages 120–135. 2007.

13. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *IEEE Computer Security Foundations Workshop*, pages 177–189, 2006.
14. A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Perspectives of Systems Informatics*, volume 4378 of *LNCS*, pages 474–480. 2007.
15. A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *International Conference on Perspectives of System Informatics*, LNCS, pages 260–273, 2003.
16. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
17. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE Computer Security Foundations Workshop*, pages 200–214, 1999.
18. G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.
19. G. Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.
20. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *ACM Symposium on Principles of Programming Languages*, pages 355–364, 1998.
21. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.
22. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.
23. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
24. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.