TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Noninterfering Schedulers

Andrei Popescu <popescua@in.tum.de>, Johannes Hölzl <hoelzl@in.tum.de>, Tobias Nipkow <nipkow@in.tum.de>

TUM-I1331

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 [6, 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 [2,4,9,20]. 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]:

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

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* [10,18,19] and *scheduler-independent* [10,17,24] 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 [5, 10, 13], but suffer from various limitations: lack of coping with dynamic thread creation [5], too harsh requirements on individual threads (strong security) [17,20], too weak confidentiality guarantees on the overall concurrent system [10], the reliance on expensive or not always feasible conditions such as race freedom [24] or termination [10], 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 [8].

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 [10], 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 appendix.

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., [10, 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 $2{\cdot}1{\cdot}1 <_{2{\cdot}1} 2{\cdot}1{\cdot}2$.

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 $n \in$ **threadID**, $N \subseteq \mathsf{maySp}\ n$ and $M \subseteq$ **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 §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 $(ml, Ml) \in \mathbf{rhist}$ 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 [10, 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 $sch_{ml,\, \mathsf{Havail}^{Sc}\ ml}\ m$—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 $1 * 0.5 * 0.75 = 0.375$.

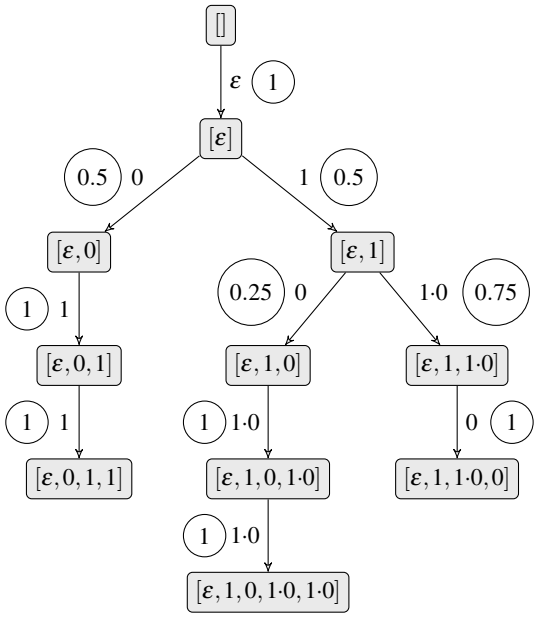

Fig. 1. A scenario with probabilities attached

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 [7], 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** $\to$ **threadID** $\to$ **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 $\leq k$—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 $\{ml.\ \exists cf.\ \mathsf{init}\ (c,s) \xrightarrow{ml} cf'\}$—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 $\mathsf{init}\ (c,s) \xrightarrow{ml} cf$—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 $obs : Sc \to \mathbf{odom}$. 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 [6] (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, $\mathsf{P}^{Sc,sch}_{ml}\ (\mathsf{Takes}\ \mathsf{inv}^{Sc,obs}\ \mathsf{Until}\ \mathsf{Takes}\ n)$. Note that here, unlike in [6], 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 [6] 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 [6] 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 $n \in \mathsf{visAvail}^{Sc,obs}\ ml$:

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

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 $F\ n = \mathsf{P}_{ml}^{Sc,sch}\ (\mathsf{Takes}\ \mathsf{inv}^{obs,Sc}\ \mathsf{Until}\ \mathsf{Takes}\ n)$ and $G\ n = sch_{ml',Ml'}\ n$.

For usch, noninterference follows immediately from its symmetry, since both F and G are constant on 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 §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** → **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 §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 invis c and $c \xrightarrow{s} (\gamma, [c_1, \ldots, c_l], s')$, we have that: **(1)** aobs $s' =$ aobs s, **(2)** invis c_i for all $i \in \{1, \ldots, k\}$; **(3)** $\gamma \in$ **cmd** implies invis γ.

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 invis c and invis d or, for all $s, t, \gamma, c_1, \ldots, c_l, s', \delta, d_1, \ldots, d_k, t'$ such that aobs $s =$ 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)** aobs $s' =$ 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 invis δ; **(4)** if $\delta = \bot$, then either $\gamma = \bot$ or invis γ; **(5)** if $\gamma, \delta \in$ **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 [10]. As argued in [10], 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 [3]: 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 $\mathsf{obs}_{c,s}\, ml =$ 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,

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

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 $E : Sc_{c,s} \to Sc_{c,s} \to \mathbf{bool}$ 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 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 §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 [9, 15, 17, 18]. In particular, the type systems from [4, 5, 10, 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 [10, §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

[10, 18] (see also §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 [10].

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 [10, 18]. Consequently, we can employ existing syntactic methods for verifying that programs satisfy the hypothesis of our main result, Th. 2.

Mantel and Sudbrock [10] 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 [10], 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 [10] 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 [10] 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 [5] 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 [10, §6] survey more work on scheduler-independent security. Unlike here, previous work [10, 17] allows schedulers to depend on the low part of the state. This is also possible in our framework and is pursued in §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. S. Abramsky. A domain equation for bisimulation. *Inf. Comput.*, 92(2):161–218, 1991.
2. J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000.
3. C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *CAV*, pages 119–130, 1997.
4. G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP*, pages 382–395, 2001.
5. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.
6. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
7. J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov chains (second edition)*. Springer, 1976.
8. H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *CSFW*, pages 200–214, 2001.
9. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF 2001*, pages 218–232, 2011.
10. H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *ESORICS*, pages 116–133, 2010.
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.

APPENDIX

This appendix provides more details, namely:

- a more general notion of scheduler depending on the low part of the state, together with more examples (§A),
- technical details on traces and temporal logic (§B),
- a notion of scheduler fairness relevant for streamlining noninterference proofs (§C),
- a recollection of syntactic criteria for possibilistic noninterference, which complement our main results (§D),
- proof sketches (§E).

A More general notion of scheduler and examples

We fix **odom**, a domain of observables, and generalize the notion of scheduler from the main paper as follows: A *scheduler* is a family of functions $(sch_{ml,Ml,S} : \mathsf{Cur}\ Ml \to \mathbb{R})_{ml,Ml,S}$, where (ml,Ml) ranges over rich histories and S over **odom**, such that

$$\forall m \in \mathsf{Cur}\ Ml.\ sch_{ml,Ml,S}\ m \geq 0 \text{ and } \sum_{m \in \mathsf{Cur}\ Ml} sch_{ml,Ml,S}\ m = 1.$$

Thus, a scheduler now depends not only on rich histories (ml,Ml), but also on observations $S \in$ **odom**.

Next we give more scheduler examples.

A.1 Priority schedulers

We consider a generalization of the uniform scheduler that allows a simple binary notion of thread prioritization. Let $w_0, w_1 \in \{0..1\}$ be such that $w_0 + w_1 = 1$ and $0 < w_1 \leq w_0$ (w_0 will be the weight associated to prioritized threads, and w_1 to non-prioritized ones). We assume a function req : **odom** $\to$ **threadID set**, representing at each observation point the set of IDs of threads requesting priority. (E.g, each thread may have a special low boolean variable that it can set to True when requiring priority and reset it to False when priority is no longer needed. Then $req\ S$ can be taken to be the set of all IDs of threads with the "request" variable set to True.) Then, at history ml, the scheduler $\mathsf{psch}^{w_0,w_1,req}$ assigns the probabilities so that the threads in $req\ (obs\ ml)$ have weight w_0 and the others weight w_1.

Formally, fix $(ml,Ml) \in$ **rhist**, $S \in$ **odom** and $m \in M = \mathsf{Cur}\ Ml$. Let $k_0 = |M \cap req\ S|$ and $k_1 = |M| - k_0$. We define the scheduler psch^{w_0,w_1} by:

1. $\mathsf{psch}^{w_0,w_1,req}_{ml,Ml,S}\ m = w_0/(k_0 * w_0 + k_1 * w_1)$, if $m \in req\ S$;
2. $\mathsf{psch}^{w_0,w_1,req}_{ml,Ml,S}\ m = w_1/(k_0 * w_0 + k_1 * w_1)$, if $m \notin req\ S$.

(Since $M \neq \emptyset$, one of k_0 and k_1 is non-zero; hence $k_0 * w_0 + k_1 * w_1 \neq 0$.) Note that the uniform scheduler usch from the main paper is a particular case of $\mathsf{psch}^{w_0,w_1,req}$, for $w_0 = w_1 = 0.5$.

A.2 The round robin species

We next consider a rather general class of schedulers with round-robin-like behavior generalizing the j-step round-robin scheduler from the main paper. Namely, the quota is not merely fixed to j, but depends on the low part of the state via a "choice" function.

A *choice* is a family $g = (g_{m,k,S})_{m \in \mathbf{threadID}, k>0, S \in \mathbf{odom}}$, where $g_{m,k,S} \in \{\mathsf{stay}, \mathsf{move}\}$. Given a choice g, we define rsch^g, the *g-round robin scheduler*, as follows: In a configuration with non-empty history $ml = nl \,\#\, m$, g is used to decide whether the scheduler tape should stay at the last executed thread, m, or move to the first in the waiting queue based on the number of times m has been last scheduled, k, and on the observation S:

(1) if g says "stay" and "stay" is an option, i.e., m is still available, then this thread is scheduled next (with probability 1);
(2) if g says "move", then the queue with the highest waiting time is scheduled next;
(3) regardless of what g says, if "stay" is not an option, then the scheduler tape moves, i.e., the action described at (2) is taken.

Given a choice g, we define rsch^g, the *g-round robin scheduler*, as follows, for all $(ml, Ml) \in \mathbf{rhist}$, $S \in \mathbf{odom}$ and $p \in M = \mathsf{Cur}\, Ml$:

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

$$\mathsf{rsch}^g_{ml,Ml,S}\, p = \begin{cases} 1, \text{ if } g_{m,\$(ml),S} = \mathsf{stay} \wedge p = m, \\ 1, \text{ if } g_{m,\$(ml),S} = \mathsf{move} \wedge p = \max_{ml,Ml} M, \\ 1, \text{ if } m \notin M \wedge p = \max_{ml,Ml} M, \\ 0, \text{ otherwise.} \end{cases}$$

Then the j-step round robin from the main paper, rsch^j, is obtained as rsch^{g^j}, where the choice g^j is given by $g^j_{m,k,S} = \mathsf{stay}$ if $k < j$ and $g^j_{m,k,S} = \mathsf{move}$ otherwise.

Yielding round robins. This is another interesting instance of the round robin species. We assume a function $yield : \mathbf{odom} \to \mathbf{bool}$ indicating whether the currently running thread wishes to yield execution or not. (E.g., this can be achieved by a shared low boolean flag that a thread sets to True when it wants to yield execution.) Then we define the choice $g^{j,yield}$ by $g^{j,yield}_{m,k,S} = \mathsf{stay}$ if $k < j$ and $yield\ S = \mathsf{False}$, and $g^{j,yield}_{m,k,S} = \mathsf{move}$ otherwise. We write $\mathsf{rsch}^{j,yield}$ instead of $\mathsf{rsch}^{g^{j,yield}}$.

A.3 The blocking-uniform species

This scheduler is a hybrid between round robin and uniform, where fair scheduling is not achieved by keeping a linear queue, but rather by considering the available threads as forming a "probabilistic queue". Initially, no thread is blocked. The next thread m is selected with uniform probability among the available threads that are non-blocked, and then, after finishing its quota, m is blocked from future scheduling (i.e., temporarily removed from the probabilistic queue) until all the others have been scheduled. Thus, the probabilistic queue shrinks continuously, until it eventually becomes empty, moment at which all the blocked threads are being de-blocked, and thus the probabilistic queue is being refilled with all the available threads.

Let $(ml = [m_0, \ldots, m_{k-1}], Ml = [M_0, \ldots, M_k])$ be a rich history and let $M = \mathsf{Cur}\, Ml = M_k$. A pair (u, v) with $0 \leq u < v < k$ is said to be a *fair resolution cut* (*FR-cut* for short)

of (ml, Ml) if $M_u = \{m_u, \ldots, m_v\}$, i.e., if all the available threads at point u in history, and only those, have been taken between points u and v. Clearly $(0,1)$ is an FR-cut, since $M_0 = \{\varepsilon\}$ and $m_0 = \varepsilon$. We can therefore speak of the *rightmost FR-cut* of (ml, Ml), defined as the FR-cut (u,v) with the greatest v.

Rightmost FR-cuts (u,v) of rich histories will guide the aforementioned process of shrinking and refilling the probabilistic queue: if $v < k-1$, the queue still needs to shrink, since the last-started fair resolution has not been completed; if $v = k-1$, then the queue needs to be refilled. These FR-cut-based decisions are only made when the j-step quota of the current thread has been finished.

We fix a choice function g as in §A.2. We define busch^g, the *g-blocking-uniform scheduler*, as follows, for all $(ml, Ml) \in \mathbf{rhist}$, $S \in \mathbf{odom}$ and $p \in M = \mathsf{Cur}\, Ml$:

– If $ml = []$, then $Ml = \{\varepsilon\}$, $M = \{\varepsilon\}$ and $p = \varepsilon$, and we put $\mathsf{busch}^g_{ml,Ml,S}\, p = 1$.

– Assume ml has the form $nl \,\#\, m$ and let (u,v) be the rightmost FR-cut in (ml, Ml).

We distinguish three cases:

Case 1: If $g_{m,\$(ml),S} = \mathsf{stay}$ and $m \in M$, we define $\mathsf{busch}^g_{ml,Ml,S}\, p = \begin{cases} 1, \text{ if } p = m, \\ 0, \text{ otherwise.} \end{cases}$

Case 2: If Case 1 does not hold and $v < k-1$, then let $N = (M_k \cap M_u) \setminus \{m_{i+1}, \ldots, m_{k-1}\}$. We define $\mathsf{busch}^g_{ml,Ml,S}\, p = \begin{cases} 1/|N|, \text{ if } p \in N, \\ 0, \quad \text{otherwise.} \end{cases}$

Case 3: If Case 1 does not hold and $v = k-1$, then we define $\mathsf{busch}^g_{ml,Ml,S}\, p = 1/|M|$.

B Technical details on traces, probabilities and temporal logic

B.1 Traces and probabilities for a scenario

Let Sc be a scenario and $pl \in Sc$. We model (Sc, pl)-traces as functions $mt : \mathbf{nat} \to \mathbf{threadID} \cup \{*\}$ such that, for all $k \in \mathbf{nat}$, either $pl \,\#\, [mt\ 0, \ldots, mt\ k] \in Sc$, or $mt\ k = *$. Thus, finite traces are captured as traces that are constant $*$ from a given position.

We shall overload the list concatenation symbol # to operate on traces as well, thus writing $ml \,\#\, mt$ for the concatenation of list ml and trace mt. Given pl and ml such that $pl \,\#\, ml \in Sc$, we define $\mathsf{Bev}^{Sc}_{pl,ml}$ as the set of all (Sc, pl)-traces starting with ml, namely, $\{ml \,\#\, mt \mid mt \in \mathsf{Trace}^{Sc}_{pl \,\#\, ml}\}$. Note that $\mathsf{Bev}^{Sc}_{pl,[]} = \mathsf{Trace}^{Sc}_{pl}$.

For example, in Fig. 1 from the main paper, all traces are finite, i.e., have trailing occurrences of $*$. We have that:

– $\mathsf{Trace}^{Sc}_{[\varepsilon],1]}$ consists of two traces:
  - $0, 1{\cdot}0, 1{\cdot}0, *, *, *, *\ldots$,
  - $1{\cdot}0, 0, *, *, *, *, *\ldots$.
– $\mathsf{Bev}^{Sc}_{[\varepsilon],[1]}$ consists of the above two traces with 1 pre-appended:
  - $1, 0, 1{\cdot}0, 1{\cdot}0, *, *, *, *, *\ldots$,
  - $1, 1{\cdot}0, 0, *, *, *, *, *\ldots$.

Given $pl \in Sc$, we organize Trace^{Sc}_{pl} as a sample space, by defining:

– Gen^{Sc}_{pl}, the *generator set for pl*, to be $\{\mathsf{Bev}^{Sc}_{pl,ml} \mid pl \,\#\, ml \in Sc\}$;

– Salg^{Sc}_{pl}, the (Sc,pl)-*trace sigma-algebra*, to be the sigma-algebra generated by Gen^{Sc}_{pl}.

Given an observation $obs : Sc \to \mathbf{odom}$, a scheduler *sch* induces standard probability spaces on the traces sigma-algebras of the scenario *Sc*, computed as follows. $\mathsf{P}^{Sc,obs,sch}_{pl} : \mathsf{Salg}^{Sc}_{pl} \to [0,1]$ is the unique probability measure that extends the following function $\mathsf{P}^{Sc,obs,sch}_{pl} : \mathsf{Gen}^{Sc}_{pl} \to [0,1]$: Assume $pl \,\#\, ml \in Sc$, where $ml = [m_0,\ldots,m_{k-1}]$. For all $i < k$, let $nl_i = [m_0,\ldots,m_{i-1}]$. Then

$$\mathsf{P}^{Sc,obs,sch}_{pl}\,\mathsf{Bev}^{Sc}_{pl,ml} = \Pi_{i<k}\, sch_{nl_i,\,\mathrm{Havail}^{Sc}\,nl_i,\,obs\,nl_i} m_i.$$

The above is a standard construction in Markov Chain theory, refered to in [7] as the *sequence space*.

We call the elements of Salg^{Sc}_{pl} (Sc,pl)-*events* and the elements of Gen^{Sc}_{pl} (Sc,pl)-*basic events*. We think of an experiment on our sample space as "running" the scenario, which yields an execution. Then, the fact that this execution happened to be in a particular set from Salg^{Sc}_{pl} constitutes an "event".

B.2 Temporal logic operators on traces

We shall work with a simple but rather general notion of temporal logic, featuring a "next" operator over sequences of actions (here, actions will be thread IDs) and infinitary conjunctions and disjunctions (in the style of [1]). All the standard LTL connectives are definable from these.

The LTL formlas, ranged over by φ, ψ, χ, are specified recursively by the following grammar, where U ranges over arbitrary predicates of type $U : \mathbf{hist} \to \mathbf{threadID} \to \mathbf{bool}$ and F over countable sets of LTL formulas:

$$\varphi \;::=\; \mathsf{Non}\;\varphi \mid \mathsf{Or}\;F \mid \mathsf{And}\;F \mid \mathsf{Takes}\;U \mid \mathsf{Next}_{ml}\,\varphi$$

To interpret the formulas, we fix a scenario *Sc*. The notion of an (Sc,pl)-trace satisfying an LTL formula (for $pl \in Sc$), $mt \models^{Sc}_{pl} \varphi$, is defined recursively on φ:

– $mt \models^{Sc}_{pl} \mathsf{Non}\;\varphi$ iff $mt \not\models^{Sc}_{pl} \varphi$;
– $mt \models^{Sc}_{pl} \mathsf{Or}\;F$ iff $\exists \varphi \in F.\; mt \models^{Sc}_{pl} \varphi$;
– $mt \models^{Sc}_{pl} \mathsf{And}\;F$ iff $\forall \varphi \in F.\; mt \models^{Sc}_{pl} \varphi$;
– $mt \models^{Sc}_{pl} \mathsf{Takes}\;U$ iff $mt\;0 \neq *$ and $U\;pl\;(mt\;0)$;
– $mt \models^{Sc}_{pl} \mathsf{Next}_{ml}\,\varphi$ iff $\exists nt.\; mt = ml \,\#\, nt \wedge nt \models^{Sc}_{pl \,\#\, ml} \varphi$.

Above, the interesting cases are Takes and Next:

– $\mathsf{Takes}\,U$ holds for an (Sc,pl)-trace *mt* iff *U* holds for the first step of *mt* at the current point *pl*.
– $\mathsf{Next}_{ml}\,\varphi$ holds for a trace iff *ml* is a prefix of that trace and, after the *ml* steps, φ holds.

We define $\mathsf{trOf}^{Sc}_{pl}\,\varphi \equiv \{mt \mid mt \models^{Sc}_{pl} \varphi\}$, and write $\mathsf{P}^{Sc,obs,sch}_{pl}\,\varphi$ instead of $\mathsf{P}^{Sc,obs,sch}_{pl}\,(\mathsf{trOf}_{pl}\,\varphi)$.

Thanks to the infinitary conjunction and disjunction, we can define all the usual LTL operators as derived operators (below, $\prec$ is the strict prefix ordering):

- φ or $\psi \equiv \mathsf{Or}\{\varphi, \psi\}$;
- φ and $\psi \equiv \mathsf{And}\{\varphi, \psi\}$;
- $\mathsf{Tr} \equiv \mathsf{And}\,\emptyset$;
- $\mathsf{Fls} \equiv \mathsf{Non}\,\mathsf{Tr}$;
- φ Imp $\psi \equiv (\mathsf{Non}\,\varphi)$ or ψ;
- φ Until $\psi \equiv \mathsf{Or}\,\{(\mathsf{Next}_{ml}\,\psi)$ and $(\mathsf{And}\,\{\mathsf{Next}_{nl}\,\varphi \mid nl \prec ml\}) \mid ml \in \textbf{threadID list}\}$;
- Ev $\varphi \equiv \mathsf{Tr}$ Until φ;
- Alw $\varphi \equiv \mathsf{Non}\,(\mathsf{Ev}\,(\mathsf{Non}\,\varphi))$.

If U simply tests for membership to a fixed set M, i.e., $\forall ml\, m.\, U\ ml\ m \iff m \in M$, we write Takes M instead of Takes U. We also define the formula Takes ml, expressing the notion of a trace taking a finite path (as a prefix), as Next ml True. These two notations are mutually coherent, since Takes $\{n\}$ in the first notation turns out equivalent to Takes $[n]$ in the second notation—we simply write Takes n in this case.

Here are some examples of LTL-describable sets of traces. In the situation of Fig. 1 form the main paper, we have:

- $\mathsf{trOf}^{Sc}_{ml}\,(\mathsf{Alw}\,(\mathsf{Takes}\,(\lambda nl\, n.\,\mathsf{True}))) = \emptyset$ for any $ml \in Sc$, since here all traces are finite.
- $\mathsf{trOf}^{Sc}_{[\varepsilon]}\,((\mathsf{Takes}\,0$ or $\mathsf{Takes}\,1)\,\mathsf{Until}\,(\mathsf{Takes}\,1{\cdot}0)) = \mathsf{trOf}^{Sc}_{[\varepsilon]}\,(\mathsf{Takes}\,[1,1{\cdot}0]) \cup \mathsf{trOf}^{Sc}_{[\varepsilon]}\,(\mathsf{Takes}\,[1,0,1{\cdot}0])$.

Here are some examples of probabilities for LTL-describable sets of traces. In the situation of Fig. 1 form the main paper, writing P_{pl} for $\mathsf{P}^{Sc,obs,sch}_{pl}$, we have:

- $\mathsf{P}_{[\varepsilon]}\,(\mathsf{Takes}\,[1,0,1{\cdot}0]) = \mathsf{P}_{[\varepsilon]}\,(\mathsf{Takes}\,1)\ *\ \mathsf{P}_{[\varepsilon,1]}\,(\mathsf{Takes}\,0)\ *\ \mathsf{P}_{[\varepsilon,1,0]}\,(\mathsf{Takes}\,1{\cdot}0) = 0.5 * 0.25 * 1 = 0.125$;
- $\mathsf{P}_{[\varepsilon]}\,((\mathsf{Takes}\,1)\,\mathsf{Until}\,(\mathsf{Takes}\,1{\cdot}0)) = \mathsf{P}_{[\varepsilon]}\,(\mathsf{Takes}\,1)\ *\ \mathsf{P}_{[\varepsilon,1]}\,(\mathsf{Takes}\,1{\cdot}0) = 0.5 * 0.75 = 0.375$.
- $\mathsf{P}_{[\varepsilon]}\,((\mathsf{Takes}\,0$ or $\mathsf{Takes}\,1)\,\mathsf{Until}\,(\mathsf{Takes}\,1{\cdot}0)) = 0.5$.

C Fairness and noninterference of schedulers

Let *sch* be a scheduler. Recall from the main paper that an observation-augmented scenario (OA-scenario) is a pair (Sc, obs) of a scenario Sc and an observation function on it, $obs : Sc \to \textbf{odom}$.

Definition 1. Given an OA-scenario scenario (Sc, obs), *sch* is called (Sc, obs)-*fair* if

$$\forall ml \in Sc.\ \forall m \in \mathsf{Avail}^{Sc}\,ml.\ \mathsf{P}^{Sc,obs,sch}_{ml}\,(\mathsf{Ev}\,(\mathsf{Takes}\,m)) = 1,$$

i.e., if for any history point and any available thread, the probability to eventually take (schedule) that thread is 1.

sch is called *fair* if it is (Sc, obs)-fair for all OA-scenarios (Sc, obs).

Lemma 1. All the example schedulers from the main paper, as well as the priority schedulers $\mathsf{psch}^{w_0,w_1,req}$ (from §A.1) and the yielding round robin schedulers $\mathsf{rsch}^{j,yield}$ (from §A.2), are fair.

We straightforwardly generalize the notion of scheduler noninterference from the main paper to observation-sensible schedulers:

Definition 2. *sch* is called *noninterfering* if the following holds for all OA-scenarios (Sc, obs), all $ml \in Sc$, and all $n \in \mathsf{visAvail}^{Sc,obs}\ ml$:

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

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

D Syntactic criteria for possibilistic noninterference

We let **var**, ranged over by x, y, z be a set of (program) variables. The sets exp, ranged over by e, of expressions over the variables **var**, is defined as usual.

We consider a while language with thread spawning, whose set **cmd** of commands is given by the following grammar:

$$c \quad ::= \quad x := e \mid c_1\ ;\ c_2 \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{spawn}\ [c_1, \ldots, c_k]$$

The set **state** of memory states is taken to consist of assignments of integers to variables (functions from **state** to **int**). Then the (possibilistic) small-step transition semantics assumed in §3.1 can be standardly specified by the inductive rules in Fig. 2, where *cl* ranges over lists of commands, $\mathsf{eval}\ (e, s)$ is the integer obtained from evaluating expression e in state s, and $s[x \mapsto \ldots]$ is memory update of variable x.

$$(x := e) \xrightarrow{s} (\bot, [], s[x \mapsto \mathsf{eval}(e, s)]) \qquad \mathsf{spawn}\ cl \xrightarrow{s} (\bot, cl, s)$$

$$\frac{c_1 \xrightarrow{s} (\bot, cl, s')}{c_1\ ;\ c_2 \xrightarrow{s} (c_2, cl, s')} \qquad \frac{c_1 \xrightarrow{s} (c', cl, s')}{c_1\ ;\ c_2 \xrightarrow{s} (c'\ ;\ c_2, cl, s')}$$

$$\frac{\mathsf{eval}\ e\ s \neq 0}{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{s} (c_1, [], s)} \qquad \frac{\mathsf{eval}\ e\ s = 0}{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{s} (c_2, [], s)}$$

$$\frac{\mathsf{eval}\ e\ s \neq 0}{\mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{s} (c\ ;\ \mathsf{while}\ e\ \mathsf{do}\ c, [], s)} \qquad \frac{\mathsf{eval}\ e\ s = 0}{\mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{s} (\bot, [], s)}$$

Fig. 2. Small-step semantics

We fix $\mathsf{sec} : \mathbf{var} \to \{\mathsf{hi}, \mathsf{lo}\}$ an assignment of security levels (where $\mathsf{lo} < \mathsf{hi}$) to each variable. Let **lvar** be the set of low variables, namely, $\{x \in \mathbf{var}.\ \mathsf{sec}\ x = \mathsf{lo}\}$. The domain **odom** of observables assumed in §4 is taken to consist of integer assignments to low variables, namely, the set of functions from **lvar** to **int**. The attacker observation $\mathsf{aobs} : \mathbf{state} \to \mathbf{odom}$ assumed in §4.2 is defined taking $\mathsf{aobs}\ s$ to be the low part of s, i.e., the restriction of s to **lvar**.

We write $\mathsf{pnonint}\ c$ to indicate that c is possibilistically noninterfering according to §4.2. To have a richer picture of compositional possibilistic methods, we also recall form [17] the notion of strong security (adapted to our formalism).

Strong bisimilarity of two commands is defined coinductively as the strongest relation $\sim$ satisfying the following property: for all c, d, if $c \sim d$, then 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)** $\gamma = \bot$ iff $\delta = \bot$; **(4)** if $\gamma, \delta \in \mathbf{cmd}$, then $\gamma \sim \delta$.

A command c is called *strongly secure*, written $\mathsf{ssecure}\ c$, if $c \sim c$.

Possibilistic type system criteria from the literature can be presented more compactly as recursively defined predicates over the command syntax. We define the predicates $\mathsf{high}, \mathsf{low}, \mathsf{safe} : \mathbf{cmd} \to \mathbf{bool}$ as follows, where, given an expression e, $\mathsf{esec}\ e \in \{\mathsf{hi}, \mathsf{lo}\}$ is the maximum security level of its variables:

- $\mathsf{high}\ (x := e) = (\mathsf{sec}\ x = \mathsf{hi})$
- $\mathsf{high}\ (c_1\ ;\ c_2) = (\mathsf{high}\ c_1\ \wedge\ \mathsf{high}\ c_2)$
- $\mathsf{high}\ (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) = (\mathsf{high}\ c_1\ \wedge\ \mathsf{high}\ c_2)$
- $\mathsf{high}\ (\mathsf{while}\ e\ \mathsf{do}\ c) = \mathsf{high}\ c$
- $\mathsf{high}\ (\mathsf{spawn}\ [c_1, \ldots, c_k]) = (\mathsf{high}\ c_1\ \wedge\ \ldots\ \wedge\ \mathsf{high}\ c_k)$

Thus, $\mathsf{high}\ c$ says that c only updates the high part of the state.

- $\mathsf{low}\ (x := e) = (\mathsf{esec}\ e \leq \mathsf{sec}\ x)$
- $\mathsf{low}\ (c_1\ ;\ c_2) = (\mathsf{low}\ c_1\ \wedge\ \mathsf{low}\ c_2)$
- $\mathsf{low}\ (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) = (\mathsf{esec}\ e = \mathsf{lo}\ \wedge\ \mathsf{low}\ c_1\ \wedge\ \mathsf{low}\ c_2)$
- $\mathsf{low}\ (\mathsf{while}\ e\ \mathsf{do}\ c) = (\mathsf{esec}\ e = \mathsf{lo}\ \wedge\ \mathsf{low}\ c)$
- $\mathsf{low}\ (\mathsf{spawn}\ [c_1, \ldots, c_k]) = (\mathsf{low}\ c_1\ \wedge\ \ldots\ \wedge\ \mathsf{low}\ c_k)$

Thus, $\mathsf{low}\ c$ says that c does not have direct leaks and the control flow of c only depends on low variables: it corresponds to the scheduler-independent type system from [20].

- $\mathsf{safe}\ (x := e) = (\mathsf{esec}\ e \leq \mathsf{sec}\ x)$
- $\mathsf{safe}\ (c_1\ ;\ c_2) = (\mathsf{low}\ c_1\ \wedge\ \mathsf{safe}\ c_2)\ \vee\ (\mathsf{safe}\ c_1\ \wedge\ \mathsf{high}\ c_2)$
- $\mathsf{safe}\ (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) = (\mathsf{esec}\ e = \mathsf{lo}\ \wedge\ \mathsf{safe}\ c_1\ \wedge\ \mathsf{safe}\ c_2)\ \vee\ (\mathsf{high}\ c_1\ \wedge\ \mathsf{high}\ c_2)$
- $\mathsf{safe}\ (\mathsf{while}\ e\ \mathsf{do}\ c) = (\mathsf{esec}\ e = \mathsf{lo}\ \wedge\ \mathsf{safe}\ c)\ \vee\ \mathsf{high}\ c$
- $\mathsf{safe}\ (\mathsf{spawn}\ [c_1, \ldots, c_k]) = (\mathsf{safe}\ c_1\ \wedge\ \ldots\ \wedge\ \mathsf{safe}\ c_k)$

$\mathsf{safe}\ c$ corresponds to the type systems from [4, 10, 18].

Proposition 4 The following implications hold:
(1) $\mathsf{invis}\ c \Longrightarrow \mathsf{pnonint}\ c$; $\mathsf{ssecure}\ c \Longrightarrow \mathsf{pnonint}\ c$.
(2) $\mathsf{high}\ c \Longrightarrow \mathsf{safe}\ c$; $\mathsf{low}\ c \Longrightarrow \mathsf{safe}\ c$.

Point (1) of Prop. 4 expresses relationships between semantic (PER-based) notions of noninterference, while point (2) deals does the same for syntactic properties. The pointed symmetry is not a coincidence: the syntactic properties turn out to be sufficient criteria for the corresponding semantic ones:

Proposition 5 The following implications hold:
(1) $\mathsf{high}\ c \Longrightarrow \mathsf{invis}\ c$
(2) $\mathsf{low}\ c \Longrightarrow \mathsf{ssecure}\ c$
(3) $\mathsf{safe}\ c \Longrightarrow \mathsf{pnonint}\ c$.

[11] discusses in detail such possibilistic syntactic criteria. Of interest to us here is Prop. 5.(3)—it corresponds to the possibilistic type system criterion from [10] guaranteeing FSI-security.

Now, safe applied to the example program from §4.4 returns True, yielding it possibilistically noninterfering by Prop. 5.(3), hence probabilistically noninterfering by Th. 2. If we make explicit the non-atomic nature of the statement $h := \mathsf{addH}(l_1, h)$, say, rewriting it as a composition of two atomic statements $h_1 := \mathsf{addH}_1(l_1, h)\ ;\ h := \mathsf{addH}_2(h_1)$, then the program is still safe, but not low, and in fact it is not strongly secure.

E Proof development

We first introduce a few more auxiliary notions (§E.1). Then we proceed with the development that leads to the proof of the results from the main paper and from the previous section (§E.2). We also state as lemmas and justify facts that were inlined in the paper (e.g., the claim from §3.2 that $Sc_{c,s}$ is a scenario).

E.1 Further concepts

The visible subscenario of an OA-scenario. In the definition of scheduler noninterference, the restriction to visible threads was performed sequence-wise, on histories. The construction can be actually lifted to whole OA-scenarios, defining $\mathsf{visScen}\ (Sc, obs)$ as (Sc', obs'), where $Sc' = \{\mathsf{visHist}^{Sc,obs}\ ml.\ ml \in Sc\}$ and obs' is the restriction of obs to Sc'. (Lemma 8 describes the basic properties of $\mathsf{visScen}\ (Sc, obs)$.)

Invisible lists of thread IDs. We extend the predicate $\mathsf{inv}_{ml}^{Sc,obs}$ from single thread IDs p to lists pl inductively as follows:

$$\mathsf{inv}_{ml}^{Sc,obs}\ [] \qquad \frac{\mathsf{inv}_{ml}^{Sc,obs}\ pl \qquad \mathsf{inv}_{ml\#pl}^{Sc,obs}\ p}{\mathsf{inv}_{ml}^{Sc,obs}\ (pl\ \#\ p)}$$

Thus, $\mathsf{inv}_{ml}^{Sc,obs}\ pl$ says that, at history ml, the pl-steps are invisible.

Probabilistic bisimilarity between OA-scenarios. The notion of *sch*-probabilistic bisimulation for a configuration (c,s) (introduced in §4.3) is defined entirely in terms of the associated OA-scenario $(Sc_{c,s}, \mathsf{obs}_{c,s})$, and thus it also makes sense for arbitrary OA-scenarios (Sc, obs):

Definition 3. A relation $E : Sc \to Sc \to \mathbf{bool}$ is called a *sch-probabilistic bisimulation* on (Sc, obs) if the following hold:
(I1) E is an equivalence relation on Sc with countable set of equivalence classes.
(I2) For $ml, nl \in Sc$, $E\ ml\ nl$ implies $obs\ ml = obs\ nl$.
(I3) For distinct E-classes H, H' and $ml, nl \in H$, $\mathsf{P}_{ml}^{Sc,obs,sch}(ml \Rightarrow_H H') = \mathsf{P}_{nl}^{Sc,obs,sch}(nl \Rightarrow_H H')$.

Note that, given (c,s), a relation $E : Sc_{c,s} \to Sc_{c,s} \to \mathbf{bool}$ is a *sch*-probabilistic bisimulation for (c,s) according to the definition from the main paper iff $E : Sc_{c,s} \to Sc_{c,s} \to \mathbf{bool}$ is a *sch*-probabilistic bisimulation on $(Sc_{c,s}, \mathsf{obs}_{c,s})$ according to Def. 3.

Again, thanks to condition (I3), we can define, for any two distinct E-classes H and H', $\mathsf{P}^{Sc,obs,sch}(H \Rightarrow H')$, *the probability of moving from H directly to H′ (without visiting any other E-class)*, to be $\mathsf{P}_{ml}^{Sc,obs,sch}(ml \Rightarrow_H H')$ for some (any) $ml \in H$.

Moreover, we can define the probability of *staying in H (once H has been reached)*, $\mathsf{P}^{Sc,obs,sch}(H\Downarrow)$, to be $1 - \sum_{H' \in \mathrm{Cls}_E - \{H\}} \mathsf{P}^{Sc,obs,sch}(H \Rightarrow H')$.

The above generalization also applies to *sch*-probabilistic bisimilarity:

Definition 4. Two OA-scenarios (Sc, obs) and (Sc', obs') are called *sch-probabilistically bisimilar* if there exist E, E', F such that:

(1) E and E' are *sch*-probabilistic bisimulations on (Sc, obs) and (Sc', obs'), respectively.
(2) $F : \mathsf{Cls}_E \to \mathsf{Cls}_{E'}$ is a bijection such that
   (a) $obs\ ml = obs\ ml'$ for all ml, ml', H with $ml \in H \in \mathsf{Cls}_E$ and $ml' \in F\ H$,
   (b) $\mathsf{P}^{Sc,obs,sch}\ (H \Rightarrow H_1) = \mathsf{P}^{Sc',obs',sch}\ (F\ H \Rightarrow F\ H_1)$ 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 $\mathsf{Cls}_{E'}$, respectively.

Note that, given (c,s) and (c',s'), they are *sch*-probabilistic bisimilar according to the definition from the main paper iff $(Sc_{c,s}, \mathsf{obs}_{c,s})$ and $(Sc_{c',s'}, \mathsf{obs}_{c',s'})$ are *sch*-probabilistic bisimilar according to Def. 4.

E.2 Proof sketches

Terminology. Many of our proofs will employ structural induction over lists *ml*. Depending on the direction on which we traverse the list inductively, we distinguish between:

- left-to-right induction (LR-induction for short), which proves the fact for $[]$ and then assumes the fact for *ml* and proves it for $m \# ml$;
- right-to-left induction (RL-induction for short), which proves the fact for $[]$ and then assumes the fact for *ml* and proves it for $ml \# m$.

Proof of Lemma 1. For each of the schedulers *sch* below, we fix ml_0 and m such that $ml_0 \in Sc$ and $m \in \mathsf{Avail}^{Sc}\ ml_0$. We shall omit the superscripts Sc, obs, sch—thus, we write P for $\mathsf{P}^{Sc,obs,sch}$ and Avail for Avail^{Sc}, trOf for trOf^{Sc}, etc. We also write l for $|\mathsf{Avail}\ ml_0|$. Then we show that $\mathsf{P}_{ml_0}\ (\mathsf{Ev}\ (\mathsf{Takes}\ m)) = 1$, i.e., that $\mathsf{P}_{ml_0}\ (\mathsf{Alw}\ (\mathsf{Non}\ (\mathsf{Takes}\ m))) = 0$.

Fairness of $\mathsf{psch}^{w_0,w_1,req}$: We adapt an argument sketched by Smith in [18, §5] for the uniform scheduler in a slightly different context. Recall that $0 < w_1 \leq w_0$ and $w_0 + w_1 = 1$. Let k be the bound ensured by the boundedness condition for Sc.

For each $i \in \mathbf{nat}$, we define M_i as the set of (Sc, ml_0)-traces that avoid (i.e., do not contain) m in the first i steps. $(M_i)_{i \in \mathbf{nat}}$ forms a decreasing sequence and $\bigcap_{i \in \mathbf{nat}} M_i = \mathsf{trOf}_{ml_0}\ (\mathsf{Alw}\ (\mathsf{Non}\ (\mathsf{Takes}\ m)))$. Moreover, after i execution steps, at most $i * k$ new threads could have been spawned, hence at most $l + i * k$ could be in the thread pool. Thus, the probability that m is selected at step $i+1$ is at least $\frac{w_1}{w_0} * \frac{1}{l+i*k}$, implying that $\mathsf{P}_{ml_0}\ M_{i+1} \leq \mathsf{P}_{ml_0}\ M_i * (1 - \frac{w_1}{w_0} * \frac{1}{l+i*k})$ for all $i \in \mathbf{nat}$. Inductively, we obtain $\mathsf{P}_{ml_0}\ (\mathsf{Alw}\ (\mathsf{Non}\ (\mathsf{Takes}\ m)))$ $\leq \prod_{i \in \mathbf{nat}} (1 - \frac{w_1}{w_0} * \frac{1}{l+i*k})$.

Thus, it suffices to show $\prod_{i \in \mathbf{nat}} (1 - \frac{w_1}{w_0} * \frac{1}{l+i*k}) = 0$. By a well-known convergence criterion, this is true iff $\sum_{i \in \mathbf{nat}} \frac{w_1}{w_0} * \frac{1}{l+i*k} = \infty$. The latter is true, since $\sum_{i \in \mathbf{nat}} 1/i = \infty$ is again well-known.

Fairness of usch: Covered by the above, since usch is a particular case of $\mathsf{psch}^{w_0,w_1,req}$.

Fairness of rsch^j: If the tape is currently at m and m has taken less than j steps, then m will be scheduled next with probability 1.

Otherwise, after at most $l * j$ steps, m will become first in the waiting queue. Indeed, every n spawned henceforth before m is scheduled will have waiting time smaller than m (and in fact smaller than all elements in $\mathsf{Avail}^{Sc}\ ml_0$), and therefore m will be ahead of n in the queue. As for the remaining threads $\mathsf{Avail}^{Sc}\ ml_0 - \{m\}$, in the worst case at moment ml_0 they are all ahead of m in the queue, but will all be scheduled in the next $l * j$ steps, leaving m first in the queue.

Fairness of $\mathsf{rsch}^{j,yield}$: The same argument as the one for rsch^{j} works here too, since the only difference from rsch^{j} is that with $\mathsf{rsch}^{j,yield}$ the threads may yield earlier than after j steps, making $\mathsf{rsch}^{j,yield}$ at least "as fair" as rsch^{j}.

Fairness of busch^{j}: One can easily show by induction that any history in Sc, and in particular ml_0, has the form $nl_1 \,\#\, \ldots \,\#\, nl_k \,\#\, pl$, where each (i_u, j_u), consisting of the start and end positions of nl_u in ml, forms an FR-cut in $(ml, \mathsf{Havail}^{Sc}\ ml)$. Let $ql = nl_1 \,\#\, \ldots \,\#\, nl_k$.

If m does not appear in pl, then it is still in the probabilistic queue, meaning it will be scheduled with probability 1 in at most $|\mathsf{Avail}^{Sc}\ ql| * j$ steps.

Otherwise, m will have to wait for pl to be completed to a pl' such that pl is a prefix of pl' and the start and end positions of pl' in $ql \,\#\, pl'$ form an FR-cut. There is a finite number of ways in which such completions can happen, since pl' may only contain threads in $\mathsf{Avail}^{Sc}\ ql$, each occurring at most j times. Let a be the greatest length of such a completion pl', and b be the largest number of available threads at history $ql \,\#\, pl'$ for such a completion pl'. Then m will be scheduled with probability 1 in at most $a + b * j$ steps. □

Lemma 2. Let $(Sc, obs) = (\mathsf{Sc}_{c,s}, \mathsf{obs}_{c,s})$. Then:
(1) Sc is a scenario.
(2) If $\mathsf{init}\ (c,s) \xrightarrow{ml} (Ml, ml, thr, s)$ and $\mathsf{invis}\ (thr\ n)$, then $n \notin \mathsf{visAvail}^{Sc,obs}\ ml$.

Proof. (1): Start consistency and prefix closure are immediate. The other scenario conditions follow from the definition of the transition relation on configurations (using the notations from that definition):

- Finite branching: From the fact that the thread ID label is picked from the finite set $\mathsf{Cur}\ Ml$.
- Termination consistency and step consistency: by the choice of M', differing from $\mathsf{Cur}\ Ml$ only w.r.t. m (if this thread has terminated) and the thread IDs picked fresh for the threads spawned by m.
- Boundedness: Assume c is k-spawn-bounded. Then $\forall m \in \mathsf{Avail}^{Sc}\ ml.\ |\mathsf{spawns}^{Sc}_{ml}\ m| \leq k$ follows by easy LR-induction on ml.

(2): We rephrase the desired fact as
$n \in \mathsf{visAvail}^{Sc,obs}\ ml \Longrightarrow (\mathsf{init}\ (c,s) \xrightarrow{ml} (Ml, ml, thr, s) \Longrightarrow \neg\ \mathsf{invis}\ (thr\ n))$.
This follows by induction on the definition of $\mathsf{visAvail}^{Sc,obs}$. □

Lemma 3. Let *sch* be any of the example schedulers from §A (in particular, the examples from the main paper). Then, if *sch* fair, it is also noninterfering.

Proof. We shall use the notations from Def. 2 and shall omit the superscripts Sc, obs, sch. By Lemma 8.(3), we have that $obs\ ml' = obs\ ml$, and let S denote this value. The desired equality is therefore the equality of two functions defined on $n \in \mathsf{visAvail}\ ml$, say $F = G$, where
- $F\ n = \mathsf{P}_{ml}(\mathsf{Takes\ inv\ Until\ Takes}\ n)$,
- $G\ n = sch_{ml',Ml',S}\ n$.

If $\mathsf{visAvail}\ ml = \emptyset$, then the equality holds trivially, so we assume $\mathsf{visAvail}\ ml \neq \emptyset$.

Noninterference of $\mathsf{psch}^{w_0,w_1,req}$: By the definition of the priority scheduler, G is the unique positive function H such that H is not 0 everywhere and the following hold for all $n_1, n_2 \in \mathsf{visAvail}\ ml$:
-(a) If $n_1, n_2 \in req\ S$, then $H\ n_1 = H\ n_2$;
-(b) If $n_1 \notin req\ S$ and $n_2 \notin req\ S$, then $H\ n_1 = H\ n_2$;
-(c) If $n_1 \in req\ S$ and $n_2 \notin req\ S$, then $w_1 * H\ n_1 = w_0 * H\ n_2$.

By fairness, $\forall n \in \mathsf{visAvail}.\ \mathsf{P}_{ml_0}(\mathsf{Ev}(\mathsf{Takes}\ n)) > 0$, hence, since $\mathsf{visAvail}\ ml \neq \emptyset$, we have $\mathsf{P}_{ml_0}(\mathsf{Alw}(\mathsf{Takes\ inv})) < 1$, and hence $\exists n \in \mathsf{visAvail}\ ml.\ F\ n > 0$. Thus, F is not zero everywhere (and is of course positive). Moreover, by the symmetry of the scheduler and the fact that taking invisible steps keeps the observation constantly S, we have that F also satisfies (a)–(c). Hence, F is equal to G.

Noninterference of usch: Covered by the above, since the uniform scheduler is a particular case of priority scheduler.

Noninterference of rsch^g (assuming rsch^g is fair): Let m be the last thread in the history and k the number of occurrences of m at the end of the history. If m is visible and g says "stay" (i.e., $g_{m,k,S} = \mathsf{stay}$), then both $F\ n$ and $G\ n$ are either 1, if $n = m$, or 0, otherwise. If m is invisible or g says "move" (i.e., $g_{m,k,S} = \mathsf{move}$), then both $F\ n$ and $G\ n$ are either 1, if n is the next visible thread in the queue, or 0, otherwise. (The reason why $F\ n$ is 1 if n is the next visible thread in the queue is that, since (by fairness) $\mathsf{P}_{ml}(\mathsf{Alw\ inv}) < 1$, we know that a visible thread will eventually be picked with nonzero probability—the first such thread is of course n; moreover, any nonzero probability is here 1.)

Noninterference of busch^g (assuming busch^g is fair): Similarly to rsch^g, just that blocked threads are also taken into account and probability 1 is replaced by 1 divided to the number of non-blocked visible threads. Namely, let m be the last thread in the history, k the number of occurrences of m at the end of the history, and i the number of currently non-blocked visible threads. If m is visible and g says "stay" (i.e., $g_{m,k,S} = \mathsf{stay}$), then both $F\ n$ and $G\ n$ are either 1, if $n = m$, or 0, otherwise. If m is invisible or g says "move" (i.e., $g_{m,k,S} = \mathsf{move}$), then both $F\ n$ and $G\ n$ are:
— either $1/i$, if n is non-blocked or all non-blocked threads are invisible;
— or 0, otherwise. □

Proof of Proposition 1 from the main paper. Follows from Lemmas 3 and 1. □

Lemma 4. Assume (E_1, E_2, F) is a sch-probabilistic bisimilarity between (Sc_1, obs_1) and (Sc_2, obs_2). Then $\mathsf{P}^{Sc_1,obs_1,sch}(H_1\Downarrow) = \mathsf{P}^{Sc_2,obs_2,sch}(F\,H_1\Downarrow)$ for all $H_1 \in \mathsf{Cls}_{E_1}$

Proof. We shall write P_1 and P_2 instead of $\mathsf{P}^{Sc_1,obs_1,sch}$ and $\mathsf{P}^{Sc_2,obs_2,sch}$. H_1' shall range over Cls_{E_1} and H_2' over Cls_{E_2}. We have that:
$\mathsf{P}_1(H_1\Downarrow) =$ (by standard probability theory)
$1-\sum_{H_1'.\, H_1'\neq H_1}\mathsf{P}_1(H_1 \Rightarrow H_1') =$ (by Def. 4.(2b))
$1-\sum_{H_1'.\, H_1'\neq H_1}\mathsf{P}_2(F\ H_1 \Rightarrow F\ H_1') =$ (since F is injective)
$1-\sum_{H_1'.\, F\ H_1'\neq F\ H_1}\mathsf{P}_2(F\ H_1 \Rightarrow F\ H_1') =$ (since F is surjective)
$1-\sum_{H_2'.\, H_2'\neq F\ H_1}\mathsf{P}_2(F\ H_1 \Rightarrow H_2') =$ (by standard probability theory)
$\mathsf{P}_2(F\ H_1\Downarrow)$. □

Lemma 5. *sch*-probabilistic bisimilarity is an equivalence relation on OA-scenarios.

Proof.
Reflexivity: Take E and E' to be the identity relation and F the identity function.
Symmetry: If (E,E',F) is a *sch*-probabilistic bisimilarity, then so is (E',E,F^{-1}).
Transitivity: If (E,E',F) and (E',E'',G) are *sch*-probabilistic bisimilarities, then so is $(E,E'',G\circ F)$. □

Lemma 6. Let (Sc,obs) be an OA-scenario and $ml,nl,pl \in Sc$. Then the following hold:
(1) $\mathsf{inv}^{Sc,obs}_{ml}\ m \Longrightarrow \mathsf{visAvail}^{Sc,obs}\ (ml\ \#\ n) = \mathsf{visAvail}^{Sc,obs}\ ml$.
(2) $\mathsf{inv}^{Sc,obs}_{ml}\ pl \Longrightarrow \mathsf{visAvail}^{Sc,obs}\ (ml\ \#\ pl) = \mathsf{visAvail}^{Sc,obs}\ ml$.
(3) $ml \preceq nl \Longrightarrow \mathsf{visAvail}^{Sc,obs}\ ml \preceq \mathsf{visAvail}^{Sc,obs}\ nl$.

Proof. (1): By easy induction on the definition of $\mathsf{visAvail}$.
(2): By easy induction on the definition of inv on lists, using point (1).
(3): We rephrase the desired fact into $\mathsf{visAvail}^{Sc,obs}\ ml \preceq \mathsf{visAvail}^{Sc,obs}\ (ml\ \#\ ml')$, which follows by easy RL-induction on ml'. □

Lemma 7. Let (Sc,obs) be an OA-scenario and $ml \in Sc$.
Then $(\mathsf{visHist}^{Sc,obs}\ ml, \mathsf{visHavail}^{Sc,obs}\ ml)$ is a rich history.

Proof. Start consistency: Immediately from the definitions.
Step consitency: By RL-indiction on ml.
Termination consistency and spawn consistency: By RL-induction on ml, using Lemma 6.(1). □

The next lemma shows that visible threads form a subscenario of the original one (1,2) that retains the same observations (3) and is well-behaved w.r.t. the operations on histories (4–6).

Lemma 8. Let (Sc,obs) be an OA-scenario and let $(Sc',obs') = \mathsf{visScen}\ (Sc,obs)$. Then the following hold, where ml ranges over Sc and ml' denotes $\mathsf{visHist}^{Sc,obs}\ ml$:
(1) (Sc',obs') is an OA-scenario.
(2) $Sc' \subseteq Sc$.
(3) $obs'\ ml' = obs\ ml$.
(4) $\mathsf{Avail}^{Sc'}\ ml' = \mathsf{visAvail}^{Sc,obs}\ ml$.
(5) $\mathsf{Havail}^{Sc'}\ ml' = \mathsf{visHavail}^{Sc,obs}\ ml$.
(6) $ml \in Sc' \Longrightarrow \mathsf{visHist}^{Sc,obs}\ ml = ml$.

Proof. (1): All we need to check is that Sc' is a scenario.
Finite branching: Immediate.
Consistency: This is Lemma 7.
Prefix closure: From Lemma 6.(3).
Boundedness: From the boundedness of Sc.
(2)–(6): They all follow by RL-induction on ml using Lemma 6.(1). (Fact (2) should be rephrased as $ml \in Sc \Longrightarrow \mathsf{visHist}^{Sc,obs}\ ml \in Sc$.) □

The next lemma introduces a particular way to construct a *sch*-probabilistic bisimilarity between two OA-scenarios: by quotienting.

Lemma 9. Let (Sc, obs) and (Sc', obs') be OA-scenarios and let $G : Sc \to Sc'$ be a surjection such that:
(1) $G\ [] = []$.
(2) $\forall ml \in Sc.\ obs'\ (G\ ml) = obs\ ml$.
(3) The following holds for all distinct $ml', nl' \in Sc$ and all $ml \in G^{-1}\ ml'$:

$$\mathsf{P}^{obs,Sc,sch}_{ml}\ (ml \Rightarrow_{G^{-1}\ ml'} G^{-1}\ nl') = \begin{cases} \mathsf{P}^{obs',Sc',sch}_{ml'}\ (\mathsf{Takes}\ n), & \text{if } nl' = ml' \#\ n \text{ for some } n, \\ 0, & \text{otherwise.} \end{cases}$$

Then (Sc, obs) and (Sc', obs') are *sch*-probabilistically bisimilar.

Proof. We shall write P instead of $\mathsf{P}^{obs,Sc,sch}$ and P' instead of $\mathsf{P}^{obs',Sc',sch}$.

We define E to be the kernel of G, namely, $E\ ml\ nl \equiv (G\ ml = G\ nl)$ and E' to be the identity on Sc', namely, $E'\ ml'\ nl' \equiv (ml' = nl')$. Then the E'-classes are singletons and, by the definition of $\Rightarrow$,

$$\mathsf{P}'\ (\{ml'\} \Rightarrow \{nl'\}) = \begin{cases} \mathsf{P}'_{ml'}\ (\mathsf{Takes}\ n), & \text{if } nl' = ml' \#\ n \text{ for some } n, \\ 0, & \text{otherwise.} \end{cases}$$

Moreover, the E-classes have the form $G^{-1}\ ml'$ for $ml' \in Sc'$. We define $F : \mathsf{Cls}_E \to \mathsf{Cls}_{E'}$ by $F\ (G^{-1}\ ml') = \{ml'\}$. It is routine to check that (E, E', F) is a *sch*-probabilistic bisimilarity between (Sc, obs) and (Sc', obs'), making (Sc, obs) and (Sc', obs') *sch*-probabilistically bisimilar. □

We are now ready to prove the main result from the main paper, Theorem 2. Here is the proof idea, which reflects our overall goal of separating possibilistic and probabilistic concerns:

1. For noninterfering schedulers, any OA-scenario is probabilistically bisimilar with its visible sub-OA-scenario—this property, stated in Lemma 10, concerns the interaction of the scheduler with execution scenarios and has nothing to do with the concrete possibilistic semantics of commands.
2. On the other hand, any two scenarios given by a noninterfering command and two indistinguishable states have the same visible sub-OA-scenarios—this property, stated in Lemma 11, follows from the possibilistic noninterference of the command and has nothing to do with probabilities or schedulers.
3. Putting together 1 and 2, the result follows.

Lemma 10. If *sch* is noninterfering, then (Sc, obs) and $\mathsf{visScen}\ (Sc, obs)$ are *sch*-probabilistically bisimilar.

Proof. Let $(Sc', obs') = \mathsf{visScen}\ (Sc, obs)$. We shall write P instead of $\mathsf{P}^{obs,Sc,sch}$. Similarly, we shall write P' instead of $\mathsf{P}^{obs',Sc',sch}$.

It suffices to check the hypotheses (1)–(3) of Lemma 9, taking G to be $\mathsf{visHist}^{Sc,obs} : Sc \to Sc'$, which is a surjection by Lemma 8.(2,6). Below, we write $\mathsf{visHist}$ instead of $\mathsf{visHist}^{Sc,obs}$ and inv instead of $\mathsf{inv}^{Sc,obs}$.

(1): Immediately from the definition of $\mathsf{visHist}$.

(2): By RL-induction on ml, using Lemma 6.(2).

(3): Let $ml', nl' \in Sc'$ and $ml \in Sc$ such that $ml' = \mathsf{visHist}\ ml$ and $ml' \neq nl'$. We denote $H_1 = \mathsf{visHist}^{-1}\ ml'$ and $H_2 = \mathsf{visHist}^{-1}\ nl'$. We need to prove the following:
(A) If nl' does not have the form $ml' \# n$ for some n, then $\mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} H_2) = 0$.
(B) If $nl' = ml' \# n$, then $\mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} H_2) = \mathsf{P}'_{ml'}\ (\mathsf{Takes}\ n)$.

We first prove:
(C) $\forall nl \in H_2.\ \mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} \{nl\}) > 0 \Longrightarrow \exists n.\ nl' = ml' \# n$.

Assume $nl \in H_2$ (and hence $nl' = \mathsf{visHist}\ nl$) and $\mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} \{nl\}) > 0$. By the definition of $\Rightarrow$, any trace in $ml \Rightarrow_{H_1} \{nl\}$ must have ml as prefix, then go through some pl such that $ml \# pl'$ is in H_1 for all prefixes pl' of pl, then reach nl via some step n. Thus, $nl = ml \# pl \# n$ where $\mathsf{inv}_{ml}\ pl$. Moreover, since $nl \notin H_1$, we have $n \in \mathsf{visAvail}\ (ml \# pl)$. Hence, by the definition of $\mathsf{visHist}$:
$nl' = \mathsf{visHist}\ nl = \mathsf{visHist}\ (ml \# pl \# n) = (\mathsf{visHist}\ (ml \# pl)) \# n = (\mathsf{visHist}\ ml) \# n = ml' \# n$.

By standard probability theory, we have:
(D) $\mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} H_2) = \sum_{nl \in H_2} \mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} \{nl\})$.

Now, (A) follows from (C) and (D).

In order to prove (B), we assume $nl' = ml' \# n$ and note that the discussion from the proof of (C) actually proves more than (C), namely, it indicates a sound way to rewrite the sum from the lefthand side of (D), considering only the non-zero terms:
$\sum_{nl \in H_2} \mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} \{nl\}) =$ (by the discussion from the proof of (C))
$\sum_{pl.\ \mathsf{inv}_{ml}\ pl} \sum_{n \in \mathsf{visAvail}\ (ml \# pl)} \mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} \{ml \# pl \# n\}) =$
(by standard probability theory)
$\sum_{pl.\ \mathsf{inv}_{ml}\ pl} \sum_{n \in \mathsf{visAvail}\ (ml \# pl)} \mathsf{P}_{ml}\ (\mathsf{Takes}\ (pl \# n)) =$
(by standard probability theory and temporal formula semantics)
$\mathsf{P}_{ml}\ (\mathsf{Takes\ inv\ Until\ Takes}\ n)$.

We have thus proved:
(E) $\mathsf{P}_{ml}\ (ml \Rightarrow_{H_1} H_2) = \mathsf{P}_{ml}\ (\mathsf{Takes\ inv\ Until\ Takes}\ n)$.

Note that, since $n \in \mathsf{visAvail}\ (ml \# pl)$ and $\mathsf{inv}_{ml}\ pl$, it follows from Lemma 6.(2) that:
(F) $n \in \mathsf{visAvail}\ ml$.

We now have:
$\mathsf{P}_{ml}\ (\mathsf{Takes\ inv\ Until\ Takes}\ n) =$ (by (F), since *sch* is noninterfering)
$sch_{ml',\mathrm{visHavail}\ ml,obs\ ml}\ n =$ (by Lemma 8.(3,5))
$sch_{ml',\mathrm{Havail}\ ml',obs'\ ml'}\ n =$ (by the definition of P' and the semantics of Takes)
$\mathsf{P}'_{ml'}\ (\mathsf{Takes}\ n)$.

We have thus proved $\mathsf{P}_{ml}\ (\mathsf{Takes\ inv\ Until\ Takes}\ n) = \mathsf{P}'_{ml'}\ (\mathsf{Takes}\ n)$, which, together with (E), proves (B), as desired. □

Lemma 11. If c is possibilistically noninterfering and $\mathsf{aobs}\ s = \mathsf{aobs}\ s'$, then $\mathsf{visScen}\ (\mathsf{Sc}_{c,s}, \mathsf{obs}_{c,s}) = \mathsf{visScen}\ (\mathsf{Sc}_{c,s'}, \mathsf{obs}_{c,s'})$.

Proof. Let $(Sc, obs) = (\mathsf{Sc}_{c,s}, \mathsf{obs}_{c,s})$ and $(Sc', obs') = (\mathsf{Sc}_{c,s'}, \mathsf{obs}_{c,s'})$. We write visHist, Avail, visAvail and visHavail for $\mathsf{visHist}^{\mathrm{Sc},obs}$, $\mathsf{Avail}^{\mathrm{Sc}}$, $\mathsf{visAvail}^{\mathrm{Sc},obs}$, and $\mathsf{visHavail}^{\mathrm{Sc},obs}$; similarly, we write $\mathsf{visHist}'$, Avail', $\mathsf{visAvail}'$ and $\mathsf{visHavail}'$ for $\mathsf{visHist}^{\mathrm{Sc}',obs'}$, $\mathsf{Avail}^{\mathrm{Sc}'}$, $\mathsf{visAvail}^{\mathrm{Sc}',obs'}$ and $\mathsf{visHavail}^{\mathrm{Sc}',obs'}$.

According to the definition of visScen, we need to prove the following:
(A) $\{\mathsf{visHist}\ ml.\ ml \in Sc\} = \{\mathsf{visHist}'\ ml'.\ ml' \in Sc'\}$.
(Let K denote either of these two sets, after they were proved equal.)
(B) $\forall nl \in K.\ obs\ nl = obs'\ nl$.

Consider the following statement:
(C) For all $ml \in Sc$, there exists $ml' \in Sc'$ such that the following hold, where $cf = \mathsf{config}_{c,s}\ ml = (ml, Ml, thr, t)$ and $cf' = \mathsf{config}_{c,s'}\ ml' = (ml', Ml', thr', t')$:
—(C.1) $\mathsf{aobs}\ t = \mathsf{aobs}\ t'$,
—(C.2) $\mathsf{visHist}\ ml = \mathsf{visHist}'\ ml'$,
—(C.3) $\mathsf{visHavail}\ ml = \mathsf{visHavail}'\ ml'$,
—(C.4) $\forall n \in \mathsf{visAvail}\ ml.\ thr\ n \approx thr'\ n$.

We shall soon give a proof of (C). Its symmetric, (C'), starting with "For all $ml' \in Sc'$, there exists $ml \in Sc$" can then be proved similarly. Let us first show that (C) and (C') imply (A) and (B).
(A): Immediate from (C.2) and (C'.2).
(B): Let $nl \in K$. We obtain ml such that $ml \in Sc$ and $nl = \mathsf{visHist}\ ml$. By Lemma 8.(3) and the definition of $\mathsf{obs}_{c,s}$, we have $obs\ nl = obs\ ml = \mathsf{aobs}\ t$. Similarly, we have $obs\ nl = \mathsf{aobs}\ t'$. Now the desired fact follows from (C.1).

It remains to prove (C), which we do by RL-induction on ml.
Base case: Assume $ml = []$. Take $ml' = []$.
(C.1): By the assumption that $\mathsf{aobs}\ s = \mathsf{aobs}\ s'$.
(C.2): $\mathsf{visHist}\ [] = [] = \mathsf{visHist}'\ []$.
(C.3): $\mathsf{visHavail}\ [] = [\{\varepsilon\}] = \mathsf{visHavail}'\ []$.
(C.4): n is necessarily ε and $thr\ n = c \approx c = thr'\ n$ (by the assumption that c is possibilistically noninterfering).

Induction step: Assume $ml = ml_1\ \#\ m$. Then $\mathsf{init}\ (c,s) \xrightarrow{ml_1} cf_1$ and $cf_1 \xrightarrow{m} cf$, where $cf_1 = \mathsf{config}_{c,s}\ ml_1 = (ml_1, Ml_1, thr_1, t_1)$. By the induction hypothesis, we obtain $ml'_1 \in Sc'$ such that the following hold, where $cf'_1 = \mathsf{config}_{c,s'}\ ml'_1 = (ml'_1, Ml'_1, thr'_1, t'_1)$:
—(C.1$_1$) $\mathsf{aobs}\ t_1 = \mathsf{aobs}\ t'_1$,

—(C.2_1) visHist $ml_1 =$ visHist$'$ ml'_1,
—(C.3_1) visHavail $ml_1 =$ visHavail$'$ ml'_1 (in particular, visAvail $ml_1 =$ visAvail$'$ ml'_1),
—(C.4_1) $\forall n \in$ visAvail ml_1. $thr\ n \approx thr'\ n$.

We have two cases:
Case 1: $m \notin$ visAvail ml_1 (hence by (C.3_1), also $m \notin$ visAvail ml'_1). Then visHist $ml =$ visHist ml_1, and we take ml' to be ml'_1. Then each of the desired facts, (C.i) with i$\in \{1,2,3,4\}$, follows at once from the definitions and the corresponding fact (C.i$_1$).
Case 2: $m \in$ visAvail ml_1 (hence by (C.3_1), also $m \in$ visAvail ml'_1). We take ml' to be $ml'_1 \# m$—then, according tfollowing from the possibilistic noninterference of the command. o the definition of cf', we have $cf'_1 \xrightarrow{m} cf'$.

By the definition of visHist and (C.2_1), we have
visHist $ml =$ (visHist ml_1) $\# m =$ (visHist ml'_1) $\# m =$ visHist$'$ ml', which proves (C.2).

Let $d = thr\ m$ and $d' = thr'\ m$. By (C.4_1), we have $d \approx d'$. Then, also employing the definition of configuration transition for $cf_1 \xrightarrow{m} cf$ and $cf'_1 \xrightarrow{m} cf'$, we have that:
-(a) $d \xrightarrow{t_1} (\gamma, [d_1, \ldots, d_k], t)$,
-(b) $d \xrightarrow{t_1} (\gamma', [d'_1, \ldots, d'_k], t)$,
-(c) aobs $t =$ aobs t', which proves (C.1);
-(d) $d_i \approx d'_i$ for all i;
-(e) γ and γ' either are both in **cmd** and bisimilar, or both are $\perp$, or one of them is $\perp$ and the other invisible.

It remains to check (C.3) and (C.4). Note that, thanks to (C.3_1), in order to prove (C.3), it suffices to prove:
(CC.3) visAvail $ml_1 =$ visAvail$'$ ml'_1.

By the definition of configuration transition for $cf_1 \xrightarrow{m} cf$, we have that the only elements in Avail ml and not in Avail ml_1 are fresh thread IDs $p_1, \ldots, p_k$ associated to $d_1, \ldots, d_k$; and similarly for Avail$'$ ml', Avail$'$ ml'_1 and the IDs $p'_1, \ldots, p'_k$ associated to $d'_1, \ldots, d'_k$. By the choice of these fresh IDs (according to the definition of configuration transition) and (C.3_1), we have that $p_i = p'_i$ for all i. Moreover, by (d), p_i is visible iff p'_i is visible. Therefore we obtain:
-(e) visAvail $ml \setminus A =$ visAvail $ml' \setminus A$, where $A =$ visAvail $ml_1 =$ visAvail$'$ ml'_1 (the latter being equal by (C.3_1)).

To finish the proof of (CC.3), it remains to check that visAvail $ml \cap A =$ visAvail $ml' \cap A$. Each of these two sets is included in the singleton set $\{m\}$, and is empty just in case γ, respectively γ', is $\perp$. We have two subcases:
Case 2.1: $\gamma, \gamma' \in$ **cmd** and $\gamma \approx \gamma'$. Then visAvail $ml \cap A = \{m\} =$ visAvail $ml' \cap A$.
Case 2.2: One of the two, say, γ, is $\perp$ and the other, say, γ', is in **cmd** and is invisible. Then visAvail $ml \cap A = \emptyset$. Moreover, by Lemma 2.(2), $m \notin$ visAvail ml', visAvail $ml' \cap A = \emptyset$.
Case 2.3: $\gamma = \gamma' = \perp$: Then visAvail $ml \cap A = \emptyset =$ visAvail $ml' \cap A$.

Finally, (C.4) follows from (C.4_1) together with (d) and (e). □

Proof of Theorem 2 from the main paper.
By Lemma 11, we have visScen $(\mathsf{Sc}_{c,s}, obs_{c,s}) =$ visScen $(\mathsf{Sc}_{c,s'}, obs_{c,s'})$—let (Sc_0, obs_0) denote this OA-scenario. From Lemma 10, we have that $(\mathsf{Sc}_{c,s}, obs_{c,s})$ and (Sc_0, obs_0) are *sch*-probabilistically bisimilar. Similarly, $(\mathsf{Sc}_{c,s'}, obs_{c,s'})$ and (Sc_0, obs_0) are *sch*-

probabilistically bisimilar. Since *sch*-probabilistic bisimilarity is an equivalence, we obtain that $(\mathsf{Sc}_{c,s}, obs_{c,s})$ and $(\mathsf{Sc}_{c,s'}, obs_{c,s'})$, i.e., (c,s) and (c,s'), are *sch*-probabilistically bisimilar, as desired. □

Proof of Proposition 3 from the main paper.
Let $(Sc, obs) = (\mathsf{Sc}_{c,s}, \mathsf{obs}_{c,s})$ and $(Sc', obs') = (\mathsf{Sc}_{c,s'}, \mathsf{obs}_{c,s'})$. We write P for $\mathsf{P}^{Sc,obs,sch}$ (which is the same as $\mathsf{P}^{sch,c,s}$) and P' for $\mathsf{P}^{Sc',obs',sch}$ (which is the same as $\mathsf{P}^{sch,c',s'}$). By Theorem 2, (Sc, obs) and (Sc', obs') are *sch*-probabilistically bisimilar—let (E, E', F) be a *sch*-probabilistic bisimilarity between them. Let H_0 be the E-class of $[] \in Sc$ (then $F\ H_0$ is the E'-class of $[] \in Sc'$).

Given $H \in \mathsf{Cls}_E$, we define an (E,H)-*sequence* to be a sequence $H_1, \ldots, H_k$ such that:
-(1) $H_k = H$;
-(2) $\forall i \in \{1, \ldots, k-1\}.\ H_i \in \mathsf{Cls}_E$;
-(3) $\forall i \in \{1, \ldots, k-1\}.\ H_i \neq H_{i+1}$.

We write $\mathsf{Seq}(E,H)$ for the set of (E,H)-sequences. Given an (E,H)-sequence $H_1, \ldots, H_k$, we define $\mathsf{Trace}_{H_1,\ldots,H_k}$ to be the set of all traces (starting in $[]$) that visit successively $H_1, \ldots, H_k$, and then stay in H_k. We write $\mathsf{P}_{H_1,\ldots,H_k}$ for $\mathsf{P}_{[]}\ \mathsf{Trace}_{H_1,\ldots,H_k}$. For each H and (E,H)-sequence $H_1, \ldots, H_n$, we have
(A) $\mathsf{P}_{H_1,\ldots,H_k} = (\prod_{i=0}^{k-1}\ \mathsf{P}\ (H_i \Rightarrow H_{i+1})) * \mathsf{P}(H \Downarrow)$.

The notion of E'-sequence $H'_1, \ldots, H'_k$ and its associated probability $\mathsf{P}'_{H'_1,\ldots,H'_k}$ are defined similarly.

In what follows, H ranges over Cls_E.

Let $S \in$ **odom**. By condition (I2) from the Def. 3 applied to E, we have that $obs^{-1}\ S$ is the disjoint union of the family $\{H\}_{H \subseteq obs^{-1}\ S}$. (Remember that H implicitly ranges over Cls_E—thus, by $\{H\}_{H \subseteq obs^{-1}\ S}$, we actually mean $\{H\}_{H \in \mathsf{Cls}_E,\ H \subseteq obs^{-1}\ S}$; and similarly elsewhere.) Hence $\mathsf{endUpIn}_{c,s}\ S$ is the disjoint union of the family $\{\mathsf{Ev}\ (\mathsf{Alw}\ H)\}_{H \subseteq obs^{-1}\ S}$. By standard probability theory, we have
(B) $\mathsf{P}\ (\mathsf{endUpIn}_{c,s}\ S) = \sum_{H \subseteq obs^{-1}\ S}\ \mathsf{P}(\mathsf{Ev}\ (\mathsf{Alw}\ H))$.

For each H, we have that $\mathsf{Ev}\ (\mathsf{Alw}\ H)$ is the disjoint union of the family $(\mathsf{Trace}_{H_1,\ldots,H_n})_{(H_1,\ldots,H_n) \in \mathsf{Seq}(E,H)}$. Hence:
(C) $\mathsf{P}(\mathsf{Ev}\ (\mathsf{Alw}\ H)) = \sum_{(H_1,\ldots,H_n) \in \mathsf{Seq}(E,H)}\ \mathsf{P}_{H_1,\ldots,H_k}$.

Putting (A), (B) and (C) together, we obtain:
$\mathsf{P}\ (\mathsf{endUpIn}_{c,s}\ S) = \sum_{H \subseteq obs^{-1}\ S}\ \mathsf{P}(\mathsf{Ev}\ (\mathsf{Alw}\ H)) = \sum_{H \subseteq obs^{-1}\ S}\ \sum_{(H_1,\ldots,H_n) \in \mathsf{Seq}(E,H)}\ \mathsf{P}_{H_1,\ldots,H_k} = \sum_{H \subseteq obs^{-1}\ S}\ \sum_{(H_1,\ldots,H_n) \in \mathsf{Seq}(E,H)}\ (\prod_{i=0}^{k-1}\ \mathsf{P}\ (H_i \Rightarrow H_{i+1})) * \mathsf{P}(H \Downarrow)$.

We have thus obtained:
(D) $\mathsf{P}\ (\mathsf{endUpIn}_{c,s}\ S) = \sum_{H \subseteq obs^{-1}\ S}\ \sum_{(H_1,\ldots,H_n) \in \mathsf{Seq}(E,H)}\ (\prod_{i=0}^{k-1}\ \mathsf{P}\ (H_i \Rightarrow H_{i+1})) * \mathsf{P}(H \Downarrow)$.

Similarly, we have:
(D') $\mathsf{P}'\ (\mathsf{endUpIn}_{c,s'}\ S) = \sum_{H' \subseteq obs'^{-1}\ S}\ \sum_{(H'_1,\ldots,H'_n) \in \mathsf{Seq}(E',H')}\ (\prod_{i=0}^{k-1}\ \mathsf{P}'\ (H'_i \Rightarrow H'_{i+1})) * \mathsf{P}'(H' \Downarrow)$.

Finally, the right-hand sides of (D) and (D') are equal, since:
- the restriction of F is a bijection between $\{H.\ H \subseteq obs^{-1}\ S\}$ and $\{H'.\ H' \subseteq obs'^{-1}\ S\}$
- $(H_1, \ldots, H_n) \mapsto (F\ H_1, \ldots, F\ H_n)$ is a bijection between (E,H)-sequences and $(E', F\ H)$-sequences. □

Proof of Proposition 4 from §D.

(1): The involved predicates $\mathsf{invis}, \mathsf{ssecure}, \mathsf{pnonint}$, are defined as greatest predicates φ satisfying given conditions, say, Cinvis, Cssecure and Cpnonint.

E.g., Cinvis φ says: for all $c, s, \gamma, c_1, \ldots, c_l, s'$ such that $\varphi\ c$ and $c \xrightarrow{s} (\gamma, [c_1, \ldots, c_l], s')$, we have that: **(1)** $\mathsf{aobs}\ s' = \mathsf{aobs}\ s$, **(2)** $\varphi\ c_i$ for all $i \in \{1, \ldots, k\}$; **(3)** $\gamma \in$ **cmd** implies $\varphi\ \gamma$.

Moreover, it is immediate to check that

Cinvis $\varphi\ c \Longrightarrow$ Cpnonint $\varphi\ c$ and Cssecure $\varphi\ c \Longrightarrow$ Cpnonint $\varphi\ c$ for all φ and c,

which immediately implies

$\mathsf{invis}\ c \Longrightarrow \mathsf{pnonint}\ c$ and $\mathsf{ssecure}\ c \Longrightarrow \mathsf{pnonint}\ c$ for all c.

(2): By easy induction on the structure of c (the recursive conditions defining high and low are easily seen to be stronger than those defining safe).

Proof of Proposition 5 from §D. Let φ range over the syntactic predicates $\{\mathsf{high}, \mathsf{low}, \mathsf{safe}\}$ and let $\overline{\varphi}$ be its corresponding semantic notion, namely, invis if $\varphi = \mathsf{high}$, ssecure if $\varphi = \mathsf{low}$, and pnonint if $\varphi = \mathsf{safe}$.

We first note that:

(A) φ is the weakest predicate satisfying the clauses obtained by replacing its defining recursive equalities with backwards implications—let C_φ denote these clauses.

E.g., high is the weakest predicate satisfying the following properties:

- $(\mathsf{sec}\ x = \mathsf{hi}) \Longrightarrow \mathsf{high}\ (x := e)$
- $\mathsf{high}\ c_1\ \wedge\ \mathsf{high}\ c_2 \Longrightarrow \mathsf{high}\ (c_1\ ;\ c_2)$
- $\mathsf{high}\ c_1\ \wedge\ \mathsf{high}\ c_2 \Longrightarrow \mathsf{high}\ (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$
- $\mathsf{high}\ c \Longrightarrow \mathsf{high}\ (\mathsf{while}\ e\ \mathsf{do}\ c)$
- $\mathsf{high}\ c_1\ \wedge\ \ldots\ \wedge\ \mathsf{high}\ c_k \Longrightarrow \mathsf{high}\ (\mathsf{spawn}\ [c_1, \ldots, c_k])$

Moreover, it is routine to check that $\overline{\varphi}$ also satisfies the clauses C_φ. Together with (A), this proves the desired implications (1)–(3). ([11, §6] gives more details on this proof technique.) □