Securing Interaction between Threads and the Scheduler in the Presence of Synchronization

Alejandro Russo and Andrei Sabelfeld

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract. The problem of information flow in multithreaded programs remains an important open challenge. Existing approaches to specifying and enforcing information-flow security often suffer from over-restrictiveness, relying on non-standard semantics, lack of compositionality, inability to handle dynamic threads, inability to handle synchronization, scheduler dependence, and efficiency overhead for the code that results from security-enforcing transformations. This paper suggests a remedy for some of these shortcomings by developing a novel treatment of the interaction between threads and the scheduler. As a result, we present a permissive noninterference-like security specification and a compositional security type system that provably enforces this specification. The type system guarantees security for a wide class of schedulers and provides a flexible and efficiency-friendly treatment of dynamic threads.

1 Introduction

The problem of information flow in multithreaded programs remains an important open challenge [SM03]. While information flow in sequential programs is relatively well understood, information-flow security specifications and enforcement mechanisms for sequential programs do not generalize naturally to multithreaded programs [SV98]. In this light, it is hardly surprising that Jif [MZZ$^+$06] and Flow Caml [Sim03], the mainstream compilers that enforce secure information flow, lack support for multithreading.

Nevertheless, the need for information flow control in multithreaded programs is pressing because concurrency and multithreading are ubiquitous in modern programming languages. Furthermore, multithreading is essential in security-critical systems because threads provide an effective mechanism for realizing the *separation-of-duties* principle [VM01].

There is a series of properties that are desired of an approach to information flow for multithreaded programs:

– *Permissiveness* The presence of multithreading enables new attacks which are not possible for sequential programs. The challenge is to reject these attacks without compromising the permissiveness of the model. In other words, information flow models should accept as many intuitively secure and useful programs as possible.
– *Scheduler-independence* The security of a given program should not critically depend on a particular scheduler [SS00]. Scheduler-dependent security models suffer

from the weakness that security guarantees may be destroyed by a slight change in the scheduler policy. Therefore, we aim at a security condition that is robust with respect to a wide class of schedulers.

– *Realistic semantics* Following the philosophy of *extensional security* [McL90], we argue for security defined in terms of standard semantics, as opposed to security-instrumented semantics. If there are some nonstandard primitives that accommodate security, they should be clearly and securely implementable.
– *Language expressiveness* A key to a practical security model is an expressive underlying language. In particular, the language should be able to treat dynamic thread creation, as well as provide possibilities for synchronization.
– *Practical enforcement* Another practical key is a tractable security enforcement mechanism. Particularly attractive is compile-time automatic *compositional* analysis. Such an analysis should nevertheless be *permissive*, striving to trade as little expressiveness and efficiency for security as possible.

This paper develops an approach that is compatible with each of these properties by a novel treatment of the interaction between threads and the scheduler. We enrich the language with primitives for raising and lowering the security levels of threads. Threads with different security levels are treated differently by the scheduler, ensuring that the interleaving of publicly-observable events may not depend on sensitive data. As a result, we present a permissive noninterference-like security specification and a compositional security type system that provably enforces this specification. The type system guarantees security for a wide class of schedulers and provides a flexible and efficiency-friendly treatment of dynamic threads.

The main novelty of this paper, compared to a previous workshop version [RS06a], is the inclusion of synchronization primitives into the underlying language.

In the rest of the paper we present background and related work (Section 2), the underlying language (Section 3), the security specification (Section 4), and the type-based analysis (Section 5). We discuss an extension to cooperative schedulers (Section 6), an example (Section 7), implementation issues (Section 8), and present an extension of the framework with synchronization primitives (Section 9), before we conclude the paper (Section 10).

2 Motivation and background

This section motivates and exemplifies some key issues with tracking information flow in multithreaded programs and presents an overview of existing work on addressing these issues.

2.1 Leaks via scheduler

Assume a partition of variables into high (secret) and low (public). Suppose h and l are a high and a low variable, respectively. Intuitively, information flow in a program is secure (or satisfies *noninterference* [Coh78, GM82, VSI96]) if public outcomes of the program do not depend on high inputs. Typical leaks in sequential programs arise from *explicit* flows (as in assignment $l := h$) and *implicit* [DD77] flows via control flow (as in conditional $\texttt{if}\ h > 0\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$).

The ability of sequential threads to share memory opens up new information channels. Consider the following thread commands:

$$c_1 :\ h := 0;\ l := h \qquad\qquad c_2 :\ h := secret$$

where *secret* is a high variable. Thread c_1 is secure because the final value of l is always 0. Thread c_2 is secure because h and *secret* are at the same security level. Nevertheless, the parallel composition $c_1 \parallel c_2$ of the two threads is not necessarily secure. The scheduler might schedule c_2 after assignment $h := 0$ and before $l := h$ is executed in c_1. As a result, *secret* is copied into l.

Consider another pair of thread commands:

$$d_1 : (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\ l := 1 \qquad d_2 :\ \texttt{sleep}(50);\ l := 0$$

These threads are clearly secure in isolation because 1 is always the outcome for l in d_1, and 0 is always the outcome for l in d_2. However, when d_1 and d_2 are executed in parallel, the security of the threadpool is no longer guaranteed. In fact, the program will leak whether the initial value of h was positive into l under many reasonable schedulers.

We observe that program $c_1 \parallel c_2$ can be straightforwardly secured by synchronization. Assuming the underlying language features locks, we can rewrite the program as

$$c_1 : \texttt{lock}; h := 0;\ l := h; \texttt{unlock} \qquad c_2 :\ \texttt{lock};\ h := secret; \texttt{unlock}$$

The lock primitives ensure that the undesired interleaving of c_1 and c_2 is prevented. Note that this solution prevents a *race condition* [SBN$^+$97] in the sense that it is now impossible for the two threads (where one of them attempts writing) to simultaneously access variable h.

Unfortunately, synchronization primitives, which are typically used for race-condition prevention (e.g., [CF07]), offer no general solution. The source of the leak in program $d_1 \parallel d_2$ is *internal timing* [VS99]. The essence of the problem is that the timing behavior of a thread may affect—via the scheduler—the interleaving of assignments. As we will see later in this section, securing interleavings from within the program (such as with synchronization primitives) is a highly delicate matter.

What is the key reason for these flows? Observe that in both cases, it is the interleaving of the threads that introduces leaks. Hence, it is the *scheduler* and its interaction with the threads that needs to be secured in order to prevent undesired information disclosure. In this paper, we suggest a treatment of schedulers that allows the programmer to ensure from within the program that undesired interleavings are prevented.

In the rest of this section, we review existing approaches to information flow in multithreaded programs that are directly related to the paper. We refer to an overview of language-based information security [SM03] for other, less related, work.

2.2 Possibilistic security

Smith and Volpano [SV98] explore *possibilistic noninterference* for a language with static threads and a purely nondeterministic scheduler. Possibilistic noninterference states that possible low outputs of a program may not vary as high inputs are varied. Program $d_1 \parallel d_2$ from above is considered secure because possible final values of l are always 0 and 1, independently of the initial value of h. Because the choice of a scheduler affects the security of the program, this demonstrates that this definition is not scheduler-independent. Generally, possibilistic noninterference is subject to the well known phenomenon that confidentiality is not preserved by refinement [McC87]. Work by Honda et al. [HVY00, HY02] and Pottier [Pot02] is focused on type-based techniques for tracking possibilistic information flow in variants of the π calculus. Forms of noninterference under nondeterministic schedulers have been explored in the context of CCS (see [FG01] for an overview) and CSP (see [Rya01] for an overview).

2.3 Scheduler-specific security

Volpano and Smith [VS99] have investigated *probabilistic noninterference* for a language with static threads. Probabilities in their multithreaded system come from the scheduler, which is assumed to select threads *uniformly*, i.e., each thread can be scheduled with the same probability. Volpano and Smith introduce a special primitive in order to help protecting against internal timing leaks. This primitive is called `protect`, and it can be applied to any command that contains no loops. A protected command `protect`(c) is executed atomically, *by definition* of its semantics. Such a primitive can be used to secure program $d_1 \parallel d_2$ as:

$$d_1: \ \texttt{protect(if}\ h > 0\ \texttt{then}\ \texttt{sleep(100)}\ \texttt{else}\ \texttt{skip);} \qquad d_2: \ \texttt{sleep(50);}\ l := 0$$
$$l := 1$$

The timing difference is not visible to the scheduler because of the atomic semantics of `protect`. The `protect` primitive is, however, nonstandard. It is not obvious how such a primitive can be implemented (unless the scheduler is cooperative [RS06b, TRH07]). A synchronization-based implementation would face some nontrivial challenges. In the case of program $d_1 \parallel d_2$, a possible implementation of `protect` could attempt locking all other threads while execution is inside of the `if` statement:

$$\begin{aligned} d_1:\ & \texttt{lock}; (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep(100)}\ \texttt{else}\ \texttt{skip});\\ & \texttt{unlock}; \texttt{lock}; l := 1; \texttt{unlock}\\ d_2:\ & \texttt{lock}; \texttt{sleep(50)}; \texttt{unlock}; \texttt{lock}; l := 0; \texttt{unlock} \end{aligned}$$

Although this implementation prevents race conditions related to simultaneous access of variable l, unfortunately, such an implementation is insecure. The somewhat subtle reason is that when the execution is inside of the `if` statement, the other threads do not become *instantly locked*. Thread d_2 can still be scheduled, which could result in blocking and updating the wait list for the lock with d_2.

For simplicity, assume that `sleep(n)` is an abbreviation for n consecutive `skip` commands. Consider a scheduler that picks thread d_1 first and then proceeds to run a thread for 70 steps before giving the control to the other thread. If $h > 0$ then d_1 will run for 70 steps and, while being in the middle of `sleep(100)`, the control will be given to thread d_2. Thread d_2 will try to acquire the lock but will block, which will result in

d_2 being placed as the first thread in the wait list for the lock. The scheduler will then schedule d_1 again, and d_1 will release the lock with `unlock` and try to grab the lock with `lock`. However, it will fail because d_2 is the first in the wait list. As a result, d_1 will be put behind d_2 in the wait list. Further, d_2 will be scheduled to set l to 0, release the lock, and finish. Finally, d_1 is able to grab the lock and execute $l := 1$, release the lock, and finish. The final value of l is 1. If, on the other hand, $h \leq 0$ then, clearly, d_1 will finish within 70 steps, and the control will be then given to d_2, which will grab the lock, execute $l := 0$, release the lock, and finish. The final value of l in this case is 0, which demonstrates that the program is insecure. Generally, under many schedulers, chances for $l := 0$ in d_2 to execute before $l := 1$ in d_1 are higher if the initial value of h is positive. Thus, the above implementation fails to remove the internal timing leak.

This example illustrates the need for a tighter interaction with the scheduler. The scheduler needs to be able to suspended certain threads instantly. This flexibility motivates the introduction of the `hide` and `unhide` constructs in this paper.

Returning to probabilistic scheduler-specific noninterference, Smith has continued this line of work [Smi01] to emphasize practical enforcement. In contrast to previous work, the security type system accepts `while` loops with high guards when no assignments to low variables follow such loops. Independently, Boudol and Castellani [BC01, BC02] provide a type system of similar power and show possibilistic noninterference for typable programs. This system does not rely on `protect`-like primitives but winds up rejecting assignments to low variables that follow conditionals with high guards.

The approaches above do not handle dynamic threads. Smith [Smi03] has suggested that the language can be extended with dynamic thread creation. The extension is discussed informally, with no definition for the semantics of `fork`, the thread creation construct. A compositional typing rule for `fork` is given, which allows spawning threads under conditionals with high guards. However, the uniform scheduler assumption is critical for such a treatment (as it is also for the treatment of `while` loops). Consider the following example:

$$e_1:\ l := 0 \qquad e_2:\ l := 1 \qquad e_3:\ \texttt{if}\ h > 0\ \texttt{then}\ \texttt{fork}(\texttt{skip}, \texttt{skip})\ \texttt{else}\ \texttt{skip}$$

This program is considered secure according to [Smi03]. Suppose the scheduler happens to first execute e_3 and then schedule the first thread (e_1) if the threadpool has more than three threads and the second thread (e_2) otherwise. This results in an information leak from h to l because the size of the threadpool depends on h. Note that the above program is insecure for many other schedulers. A minor deviation from the strictly uniform probabilistic choice of threads may result in leaking information.

A possible alternative aimed at scheduler-independence is to force threads (created in branches of `if`s with high guards) along with their children to be protected, i.e., to disable all other threads until all these threads have terminated (this can be implemented by, for example, thread priorities). Clearly, this would take a high efficiency tall on the encouraged programming practice of placing dedicated potentially time-consuming computation in separate threads. For example, creating a new thread for establishing a network connection is a much recommended pattern [Knu02, Mah04].

The above discussion is another motivation for a tighter interaction between threads and the scheduler. A flexible scheduler would accommodate thread creation in a sensitive context by scheduling such threads independently from threads with attacker-observable assignments. This motivates the introduction of the `hfork` construct in this paper.

2.4 Scheduler-independent security

Sabelfeld and Sands [SS00] introduce a scheduler-independent security condition (with respect to possibly probabilistic schedulers) and suggest a type-based analysis that enforces this condition. The condition is, however, concerned with *external timing* leaks, which implies that the attacker is powerful enough to observe the actual execution time. External timing models rely on the underlying operating system and hardware to preserve the timing properties of a given program. Furthermore, the known padding techniques (e.g., [Aga00, SS00, KM07]) might arbitrarily change the efficiency of the resulting code.

In the present work, we assume a weaker attacker and aim for a more permissive security condition and analysis. Similarly to much related work (e.g., [VS99, Smi03, ZM03, HWS06, RS06b, RHNS07, BRRS07]) our attacker model does not permit observations of the execution time. The attacker may observe public outcomes of a program however, which is sufficient to launch attacks via internal timing. These attacks are dangerous because they can be magnified to leak all secrets in a single run (see, e.g., [RHNS07]).

2.5 Security via low determinism

Inspired by Roscoe's *low-view determinism* [Ros95] for security in a CSP setting, Zdancewic and Myers [ZM03] develop an approach to information flow in concurrent systems. According to this approach, a program is secure if its publicly-observably results are deterministic and unchanged regardless of secret inputs. This avoids refinement attacks from the outset. However, low-view determinism security rejects intuitively secure programs (such as $l := 0 \parallel l := 1$), introducing the risk of rejecting useful programs. Analysis enforcing low-view determinism are inherently noncompositional because the parallel composition with a thread assigning to low variables is not generally secure.

Recently, Huisman et al. [HWS06] have suggested a temporal logic-based characterization of low-view determinism security. This characterization enables high-precision security enforcement by known model-checking techniques.

2.6 Security in the presence of synchronization

Andrews and Reitman [AR80] propose a logic for reasoning about information flow in a language with semaphores. However, the logic comes with no soundness arguments or decision algorithms.

External timing-sensitive security has been extended to languages with semaphores primitives by Sabelfeld [Sab01] and message passing by Sabelfeld and Mantel [SM02]. Although our focus is internal timing, the semantic presentation of semaphores from the former work serves as a useful starting point for this paper.

$$
\begin{aligned}
c ::= {} & \texttt{stop} \mid \texttt{skip} \mid v := e \mid c; c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ b\ \texttt{do}\ c \\
& \mid \texttt{hide} \mid \texttt{unhide} \mid \texttt{fork}(c, \vec{d}) \mid \texttt{hfork}(c, \vec{d})
\end{aligned}
$$

Fig. 1. Command syntax

$$
\langle \texttt{skip}, m \rangle \rightharpoonup \langle \texttt{stop}, m \rangle \qquad \frac{\langle e, m \rangle \downarrow n}{\langle x := e, m \rangle \rightharpoonup \langle \texttt{stop}, m[x \mapsto n] \rangle}
$$

$$
\frac{\langle c_1, m \rangle \overset{\alpha}{\rightharpoonup} \langle \texttt{stop}, m' \rangle \quad \alpha \in \{\bullet\!\rightsquigarrow, \rightsquigarrow\!\bullet, \circ_{\vec{d}}, \bullet_{\vec{d}}\}}{\langle c_1; c_2, m \rangle \overset{\alpha}{\rightharpoonup} \langle c_2, m' \rangle}
$$

$$
\frac{\langle c_1, m \rangle \overset{\alpha}{\rightharpoonup} \langle c_1', m' \rangle \quad \alpha \in \{\bullet\!\rightsquigarrow, \rightsquigarrow\!\bullet, \circ_{\vec{d}}, \bullet_{\vec{d}}\}}{\langle c_1; c_2, m \rangle \overset{\alpha}{\rightharpoonup} \langle c_1'; c_2, m' \rangle}
$$

$$
\frac{\langle e, m \rangle \downarrow \texttt{True}}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m \rangle \rightharpoonup \langle c_1, m \rangle} \qquad \frac{\langle e, m \rangle \downarrow \texttt{False}}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m \rangle \rightharpoonup \langle c_2, m \rangle}
$$

$$
\frac{\langle e, m \rangle \downarrow \texttt{True}}{\langle \texttt{while}\ e\ \texttt{do}\ c, m \rangle \rightharpoonup \langle c; \texttt{while}\ e\ \texttt{do}\ c, m \rangle} \qquad \frac{\langle e, m \rangle \downarrow \texttt{False}}{\langle \texttt{while}\ e\ \texttt{do}\ c, m \rangle \rightharpoonup \langle \texttt{stop}, m \rangle}
$$

$$
\langle \texttt{hide}, m \rangle \overset{\rightsquigarrow\bullet}{\rightharpoonup} \langle \texttt{stop}, m \rangle \qquad \langle \texttt{unhide}, m \rangle \overset{\bullet\rightsquigarrow}{\rightharpoonup} \langle \texttt{stop}, m \rangle
$$

$$
\langle \texttt{fork}(c, \vec{d}), m \rangle \overset{\circ_{\vec{d}}}{\rightharpoonup} \langle c, m \rangle \qquad \langle \texttt{hfork}(c, \vec{d}), m \rangle \overset{\bullet_{\vec{d}}}{\rightharpoonup} \langle c, m \rangle
$$

Fig. 2. Semantics for commands

Recently, Russo et al. [RHNS07] have proposed a transformation that closes internal timing leaks by spawning sensitive computation in dedicated threads. Semaphores play a crucial role for the synchronization of these threads. However, contrary to this work, the source language for the transformation lacks semaphores.

3 Language

In order to illustrate our approach, we define a simple multithreaded language with dynamic thread creation. The syntax of language commands is displayed in Figure 1. Besides the standard imperative primitives, the language features hiding (`hide` and `unhide` primitives) and dynamic thread creation (`fork` and `hfork` primitives).

3.1 Semantics for commands

A command c and a memory m together form a *command configuration* $\langle c, m \rangle$. The semantics of configurations are presented in Figure 2. A small semantic step has the form $\langle c, m \rangle \overset{\alpha}{\rightharpoonup} \langle c', m' \rangle$ that updates the command and memory in the presence of a possible event α. Events range over the set $\{\bullet\!\rightsquigarrow, \rightsquigarrow\!\bullet, \circ_{\vec{d}}, \bullet_{\vec{d}}\}$, where $\vec{d}$ is a set of threads. The sequential composition rule propagates events to the top level. We describe the meaning of the events in conjunction with the rules that involve the events.

Two kinds of threads are supported by the semantics, low and high threads, partitioning the threadpool into low and high parts. The intention is to hide—via the scheduler—the (timing of the) execution of the high threads from the low threads.

The hiding command `hide` moves the current thread from the low to the high part of the threadpool. This is expressed in the semantics by event $\rightsquigarrow\bullet$ that communicates to the scheduler to treat the thread as high (whether or not the thread was already high). The unhiding command `unhide` has the dual effect: it communicates to the scheduler by event $\bullet\rightsquigarrow$ that the thread should be treated as low. To intuitively illustrate how to utilize `hide` and `unhide`, we modify the motivating example given in Section 2.1, where we wrap the branching command around `hide` and `unhide` commands as follows:

$$d_1 : \texttt{hide};\ (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\ \texttt{unhide};\ l := 1$$
$$d_2 : \texttt{sleep}(50);\ l := 0$$

Initially, both threads, d_1 and d_2 are treated as low by the scheduler. After executing `hide`, d_1 is temporarily considered as a high thread and d_2 is not scheduled for executing until running the command `unhide`. As a consequence, the timing differences introduced by the branching instruction in d_1 are not visible to d_2 and internal-timing leaks are thus avoided.

Although `hide` and `unhide` commands are nonstandard, we will show that, unlike `protect`, they can be straightforwardly implemented.

We define independent commands `hide` and `unhide` instead of forcing them to wrap code blocks syntactically (cf. `protect`). We expect this choice to be useful when adding exceptions to the language. For example, consider the following program

$$\texttt{try}\ \{\ \texttt{if}\ l_1\ \ \texttt{then}\ l_2 := 1; \texttt{hide}; c_1\ \texttt{else}\ l_2 := 0; \texttt{hide}; c_2\ \}\ \texttt{catch}\ \{\texttt{unhide}; c_3\}$$

where command `try` determines code blocks that might throw an exception and command `catch` states exception handlers. Variables l_1,l_2, and l_3 are public. Commands c_1 and c_2 contain branches whose guards involve secrets. Command c_3 is part of the exception handler. In this program, the `unhide` command in the exception handler refers to several `hide` primitives under the `try` statement.

Commands $\texttt{fork}(c, \vec{d})$ and $\texttt{hfork}(c, \vec{d})$ dynamically spawn a collection $\vec{d}$ of threads (commands) $\vec{d} = d_1 \ldots d_n$ while the current thread runs command c. The difference between the two primitives is in the generated event. Command `fork` signals about the creation of low threads with event $\circ_{\vec{d}}$ (where $\circ$ is read "low") while `hfork` indicates that new threads should be treated as high by event $\bullet_{\vec{d}}$ (where $\bullet$ is read "high").

3.2 Semantics for schedulers

Figure 3 depicts the semantic rules that describe the behavior of the scheduler. A scheduler is a program σ (written in a language, not necessarily related to one from Figure 1) that, together with a memory ν, forms a *scheduler configuration* $\langle\!|\sigma, \nu|\!\rangle$. We assume that the scheduler memory is disjoint from the program memory. The scheduler memory contains variable q that regulates for how many steps a thread can be scheduled. *Live* (i.e., ready to execute) threads are tracked by variable t that consists of low and high parts. The low part is named by $t_\circ$, while the high part is composed of two subpools named $t_\bullet$ and t_e. Threads in $t_\bullet$ are always high, but threads in t_e were low in the past, are high at present, and might eventually be low in the future. Threads are moved back and forth from $t_\circ$ to t_e by executing the hiding and unhiding commands. Variable r represents the running thread. Variable s regulates whether low threads may be scheduled. When s is $\circ$, both low and high threads may be scheduled. However, when s is $\bullet$, only high threads may be scheduled, preventing low threads from observing internal timing information about high threads. In addition, the scheduler might have some internal variables.

Whenever a scheduler-operation rule handles an event, it either corresponds to processing information from the top level (such as threads creation and termination) or to communicating information to the top level (such as thread selection). The rules allow to derive steps of the form $\langle\!| \sigma, \nu |\!\rangle \xrightarrow{\alpha} \langle\!| \sigma', \nu' |\!\rangle$. By convention, we refer to the variables in ν as q, t, r and s and variables in ν' as q', t', r' and s'. When these variables are not explicitly mentioned, we adopt the convention that they remain unchanged after the transition. We assume that besides event-driven transitions, the scheduler might perform internal operations that are not visible at the top level (and may not change the variables above). We abstract away from these transitions, assuming that their event labels are empty. Although the transition system in Figure 3 is nondeterministic, we only consider deterministic instances of schedulers for simplicity. We expect a natural generalization of our results to probabilistic schedulers.

The rules can be viewed as a set of basic assumptions that we expect the scheduler to satisfy. We abstract away from the actual scheduler implementation—it can be arbitrary, as long as it satisfies these basic assumptions and runs infinitely long. We discuss an example of a scheduler that conforms to these assumptions in Section 4.

The rule for event $\alpha^{r}_{\vec{d}}$ ensures that the scheduler updates the appropriate part of the threadpool (low or high, depending on α) with newly created threads. Operation $N(\vec{d})$ returns thread identifiers for $\vec{d}$ and generates fresh ones when new threads are spawn by `fork` or `hfork`. The rule for event $r \rightsquigarrow$ keeps track of a nonterminal step of thread r; as an effect, counter q is decremented. A terminal step of thread r results in a $r \rightsquigarrow \times$ event, which requires the scheduler to remove thread r from the threadpool. Events $\uparrow_\circ r'$ and $\uparrow_\bullet r'$ are driven by the scheduler's selection of thread r'. Note the difference in selecting low and high threads. A low thread can only be selected if the value of s is $\circ$, as discussed above.

Events $r \rightsquigarrow \bullet$ and $\bullet \rightsquigarrow r$ are triggered by the `hide` and `unhide` commands, respectively. The scheduler handles event $r \rightsquigarrow \bullet$ by moving the current thread from the low to the high part of the threadpool and setting s' to $\bullet$. Upon event $\bullet \rightsquigarrow r$, the scheduler moves the thread back to the low part of the threadpool, setting s' to $\circ$.

Events $r \rightsquigarrow \bullet \times$ and $\bullet \rightsquigarrow r \times$ are triggered by `hide` and `unhide`, respectively, when they are the last commands to be executed by a thread.

$$\frac{q>0 \qquad q'=q-1 \qquad t'_\alpha = t_\alpha \cup N(\vec{d})}{\llparenthesis \sigma,\nu \rrparenthesis \overset{\alpha^{r}_{\vec{d}}}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}\ \alpha \in \{\bullet,\circ\}$$

$$\frac{q>0 \qquad q'=q-1}{\llparenthesis \sigma,\nu \rrparenthesis \overset{r\rightsquigarrow}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis} \qquad \frac{q>0 \qquad q'=0 \qquad \forall\alpha\in\{\bullet,\circ\}.t'_\alpha = t_\alpha\backslash\{r\}}{\llparenthesis \sigma,\nu \rrparenthesis \overset{r\rightsquigarrow\times}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

$$\frac{q=0 \qquad s=\circ \qquad q'>0 \qquad r'\in t_\circ \cup t_\bullet}{\llparenthesis \sigma,\nu \rrparenthesis \overset{\uparrow_\circ r'}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis} \qquad \frac{q=0 \qquad q'>0 \qquad r'\in t_\bullet \cup t_e}{\llparenthesis \sigma,\nu \rrparenthesis \overset{\uparrow_\bullet r'}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

$$\frac{q>0 \qquad q'=q-1 \qquad s'=\bullet \qquad t'_\circ = t_\circ\backslash\{r\} \qquad t'_e=\{r\}}{\llparenthesis \sigma,\nu \rrparenthesis \overset{r\rightsquigarrow\bullet}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

$$\frac{q>0 \qquad q'=0 \qquad s'=\circ \qquad t'_\circ = t_\circ\cup\{r\} \qquad t'_e=\emptyset}{\llparenthesis \sigma,\nu \rrparenthesis \overset{\bullet\rightsquigarrow r}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

$$\frac{q>0 \qquad q'=0 \qquad s'=\bullet \qquad \forall\alpha\in\{\bullet,\circ\}.t'_\alpha = t_\alpha\backslash\{r\} \qquad t'_e=\emptyset}{\llparenthesis \sigma,\nu \rrparenthesis \overset{r\rightsquigarrow\bullet\times}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

$$\frac{q>0 \qquad q'=0 \qquad s'=\circ \qquad \forall\alpha\in\{\bullet,\circ\}.t'_\alpha = t_\alpha\backslash\{r\} \qquad t'_e=\emptyset}{\llparenthesis \sigma,\nu \rrparenthesis \overset{\bullet\rightsquigarrow r\times}{\rightharpoondown} \llparenthesis \sigma',\nu' \rrparenthesis}$$

Fig. 3. Semantics for schedulers

3.3 Semantics for threadpools

The interaction between threads and the scheduler takes place at the top level, the level of *threadpool configurations*. These configurations have the form $\llparenthesis \vec{c}, m, \sigma, \nu \rrparenthesis \xrightarrow{\alpha} \llparenthesis \vec{c'}, m', \sigma', \nu' \rrparenthesis$, where α ranges over the same set of events as in the semantics for schedulers.

The semantics for threadpool configurations is displayed in Figure 4. The dynamic thread creation rule is triggered when the running thread c_r generates a thread creation event $\alpha_{\vec{d}}$, where α is either $\bullet$ or $\circ$. This event is synchronized with scheduler event $\alpha^{r}_{\vec{d}}$ that requests the scheduler to handle the new threads depending on whether α is high or low.

If c_r does not spawn new threads or terminate, then its command rule is synchronized with scheduler event $r \rightsquigarrow$. If c_r terminates in a transition without labels, then scheduler event $r\rightsquigarrow\times$ is required for synchronization in order to update the threadpool information in the scheduler memory. If c_r terminates with $\rightsquigarrow\bullet$ (resp., $\bullet\rightsquigarrow$) then synchronization with $r \rightsquigarrow\bullet\times$ (resp., $\bullet\rightsquigarrow r\times$) is required to record both termination and hiding (resp., unhiding).

Scheduler event $\uparrow_\alpha r'$ triggers a selection of a new thread r' without affecting the commands in the threadpool or their memory. Finally, entering and exiting the high part

$$\frac{\langle c_r, m\rangle \overset{\alpha\,\vec{d}}{\rightharpoonup} \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \overset{\alpha^{r}_{\vec{d}}}{\rightharpoondown} \langle \sigma', \nu'\rangle \qquad \alpha \in \{\bullet, \circ\}}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{\alpha^{r}_{\vec{d}}}{\rightarrow} \langle c_1 \dots c_{r-1} c_r' \vec{d} c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \rightharpoonup \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \overset{r\leadsto}{\rightharpoondown} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{r\leadsto}{\rightarrow} \langle c_1 \dots c_{r-1} c_r' c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \rightharpoonup \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{r\leadsto\times}{\rightharpoondown} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{r\leadsto\times}{\rightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{\leadsto\bullet}{\rightharpoonup} \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{r\leadsto\bullet\times}{\rightharpoondown} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{r\leadsto\bullet\times}{\rightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{\bullet\leadsto}{\rightharpoonup} \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{\bullet\leadsto r\times}{\rightharpoondown} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{\bullet\leadsto r\times}{\rightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle \sigma, \nu\rangle \overset{\uparrow\alpha\, r'}{\rightharpoondown} \langle \sigma', \nu'\rangle \qquad \alpha \in \{\circ, \bullet\}, r' \in \{1, \dots, n\}}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{\uparrow\alpha\, r'}{\rightarrow} \langle c_1 \dots c_n, m, \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{\alpha}{\rightharpoonup} \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \overset{\alpha}{\rightharpoondown} \langle \sigma', \nu'\rangle \qquad \alpha \in \{r \leadsto \bullet, \bullet \leadsto r\}}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{\alpha}{\rightarrow} \langle c_1 \dots c_{r-1} c_r' c_{r+1} \dots c_n, m', \sigma', \nu'\rangle}$$

Fig. 4. Semantics for threadpools

of the threadpool is performed by synchronizing the current thread and the scheduler on events $r \leadsto \bullet$ and $\bullet \leadsto r$.

Let $\rightarrow^*$ stand for the transitive and reflexive closure of $\rightarrow$ (which is obtained from $\overset{\alpha}{\rightarrow}$ by ignoring events). If for some threadpool configuration *cfg* we have $cfg \rightarrow^* cfg'$, where the threadpool of cfg' is empty, then *cfg terminates* in cfg', denoted by $cfg \Downarrow cfg'$. Recall that schedulers always run infinitely; however, according to the above definition, the entire program terminates if there are no threads to schedule. We assume that $m(cfg)$ extracts the program memory from threadpool configuration *cfg*.

3.4 On multi-level extensions

Although the semantics accommodates two security levels for threads, extensions to more levels do not pose significant challenges. Assume a security lattice $\mathcal{L}$, where security levels are ordered by a partial order $\sqsubseteq$, with the intention to only allow leaks from data at level ℓ_1 to data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$. The low-and-high policy discussed above forms a two-level lattice with elements low and $high$ so that $low \sqsubseteq high$ but $high \not\sqsubseteq low$.

```
d : fork_{low}(c_{low}); fork_{medium}(c_{medium}); fork_{high}(c_{high});
  hide_{medium};
    if $k$ then $h := 3$; else $k := 1$; $k' := 3$;
    hide_{high};
      if $h$ then $h := 0$; else $h := 4$; $h' := 3$;
    unhide_{high};
  $k'' := 5$;
  unhide_{medium};
```

Fig. 5. Example of multi-level commands $\mathtt{hide}_\ell$, $\mathtt{unhide}_\ell$, and $\mathtt{fork}_\ell$

In the presence of a general security lattice, the threadpool is partitioned into as many parts as the number of security levels. Commands $\mathtt{hide}_\ell$, $\mathtt{unhide}_\ell$, and $\mathtt{fork}_\ell$ are parameterized over security level ℓ. Initially, all threads are in the $\bot$-threadpool. Whenever a thread executes a $\mathtt{hide}_\ell$ command, it enters ℓ-threadpool. The semantics needs to ensure that no threads from ℓ'-threadpools, for all ℓ' such that $\ell \not\sqsubseteq \ell'$ may execute until the hidden thread reaches $\mathtt{unhide}_\ell$. Naturally, command $\mathtt{fork}_\ell$ creates threads in ℓ-threadpool.

To illustrate the use of commands $\mathtt{hide}_\ell$, $\mathtt{unhide}_\ell$, and $\mathtt{fork}_\ell$, we present the thread d in Figure 5. We assume three security levels in our lattice: low, $medium$, and $high$, where $low \sqsubseteq medium \sqsubseteq high$. Commands c_{low}, c_{medium}, and c_{high} describe low, medium, and high threads, respectively. Variables l, k, and h (and their prime versions) are associated with security levels low, $medium$, and $high$, respectively. The program starts by spawning three threads at different security levels. Before the first hide, the low-threadpool is composed by the threads d and c_{low}, while threads c_{medium} and c_{high} are placed in the $medium$ and $high$-threadpools, respectively. At this point, any of the threads can be scheduled. Once executed $\mathtt{hide}_{medium}$, thread c_{low} is not scheduled for execution until reaching the command $\mathtt{unhide}_{medium}$. After executing the first branching instruction, $\mathtt{hide}_{high}$ is executing. Then, thread c_{medium} is not able to run and only d and c_{high} can be executed at that point of the program. After executing $\mathtt{unhide}_{high}$, thread c_{medium} can be scheduled to run. Finally, c_{low} can be scheduled to run after executing $\mathtt{unhide}_{medium}$.

We will discuss how general multi-level security can be defined and enforced in Sections 4 and 5, respectively.

4 Security specification

We specify security for programs via noninterference. The attacker's view of program memory is defined by a *low-equivalence* relation $=_L$ such that $m_1 =_L m_2$ if the projections of the memories onto the low variables are the same $m_1|_L = m_2|_L$. As formalized in Definition 4 below, a program is secure under some scheduler if for any two initial low-equivalent memories, whenever the two runs of the program terminate, then the resulting memories are also low-equivalent.

We generalize this statement to a class of schedulers, requiring schedulers to comply to the basic assumptions from Section 3 and also requiring that they themselves are not leaky, i.e., that schedulers satisfy a form of noninterference.

Scheduler-related events have different distinguishability levels. Events $\circ\overset{r}{\underset{\vec{d}}{\rightarrow}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\circ r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r is a low thread and r' can be either a low or a high thread) operate on low threads and are therefore low events. On the other hand, events $\bullet\overset{r}{\underset{\vec{d}}{\rightarrow}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\bullet r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r and r' are high threads) are high.

With the security partition defined on scheduler events, we specify the indistinguishability of scheduler configurations via *low-bisimulation*. Because we only consider deterministic schedulers, an equivalent trace-based definition is possible. However, we have chosen a bisimulation-based definition of indistinguishability because it is both intuitive and concise. The intuition behind indistinguishability of scheduler configurations is this: A candidate relation R is a low-bisimulation if the following conditions hold. For two configurations that are related by R, if one of them (say the first) can make a high step to some other configuration then this other configuration will be related to the second configuration. If none of the configurations can make a high step, but one of the configurations can make a low step, then the other one should also be able to make a low step with the same label and the resulting configurations must be related by R. Formally:

Definition 1. *A relation R is a* low-bisimulation *on scheduler configurations if whenever $\langle\!|\sigma_1, \nu_1|\!\rangle\ R\ \langle\!|\sigma_2, \nu_2|\!\rangle$, then*

- *if $\langle\!|\sigma_i, \nu_i|\!\rangle \xrightarrow{\alpha} \langle\!|\sigma_i', \nu_i'|\!\rangle$ where α is high and $i \in \{1, 2\}$, then $\langle\!|\sigma_i', \nu_i'|\!\rangle\ R\ \langle\!|\sigma_{3-i}, \nu_{3-i}|\!\rangle$;*
- *if the case above cannot be applied and $\langle\!|\sigma_i, \nu_i|\!\rangle \xrightarrow{\alpha} \langle\!|\sigma_i', \nu_i'|\!\rangle$ where α is low and $i \in \{1, 2\}$, then $\langle\!|\sigma_{3-i}, \nu_{3-i}|\!\rangle \xrightarrow{\alpha} \langle\!|\sigma_{3-i}', \nu_{3-i}'|\!\rangle$ and $\langle\!|\sigma_i', \nu_i'|\!\rangle\ R\ \langle\!|\sigma_{3-i}', \nu_{3-i}'|\!\rangle$.*

Note the condition "if the case above cannot be applied", which corresponds to the case where none of the configurations can make a high step. Scheduler configurations are low-indistinguishable if there is a low-bisimulation that relates them:

Definition 2. *Scheduler configurations $\langle\!|\sigma_1, \nu_1|\!\rangle$ and $\langle\!|\sigma_2, \nu_2|\!\rangle$ are* low-indistinguishable *(written $\langle\!|\sigma_1, \nu_1|\!\rangle \sim_L \langle\!|\sigma_2, \nu_2|\!\rangle$) if there is a low-bisimulation R such that $\langle\!|\sigma_1, \nu_1|\!\rangle\ R\ \langle\!|\sigma_2, \nu_2|\!\rangle$.*

Noninterference for schedulers requires low-bisimilarity under any memory:

Definition 3. *Scheduler σ is* noninterferent *if $\langle\!|\sigma, \nu|\!\rangle \sim_L \langle\!|\sigma, \nu|\!\rangle$ for all ν.*

Figure 6 displays an example of a scheduler in pseudocode. This is a round-robin scheduler that keeps track of two lists of threads: low and high ones. The scheduler interchangeably chooses between threads from these two lists, when possible. It waits for events generated by the running thread (expressed by primitive `receive`). Functions `head`, `tail`, `remove`, and `append` have the standard semantics for list operations. Operation $N(\vec{d})$, variables $t_\circ$, $t_\bullet$, s, r, and q have the same purpose as described in Section 3.2. Constant M is a positive natural number. Variable $turn$ encodes the interchangeable choices between low and high threads. Function $run(r)$ launches the

```
$t_\circ := [c]; t_\bullet := []; r := c; s := 0; turn := 0;$
while (True) do {
$q := M; run(r);$
while $(q > 0)$ do {
  receive
   $\circ^r_{\vec{d}}$ :       $t_\circ :=$ append$(t_\circ, N(\vec{d}));$
   $\bullet^r_{\vec{d}}$ :       $t_\bullet :=$ append$(t_\bullet, N(\vec{d}));$
   $r\rightsquigarrow$:      skip;
   $r\rightsquigarrow\times$ :  $t_\circ :=$ remove$(r, t_\circ); t_\bullet :=$ remove$(r, t_\bullet);$
            $q := 0;$
   $r\rightsquigarrow\bullet$ :  $t_\circ :=$ remove$(r, t_\circ); t_\bullet :=$ remove$(r, t_\bullet);$
            $t_\bullet :=$ append$(t_\bullet, [r]); s := 1;$
   $\bullet\rightsquigarrow r$ :  $t_\circ :=$ append$(t_\circ, [r]);$
            $t_\bullet :=$ remove$(r, t_\bullet); s := 0; q := 0;$
   $r\rightsquigarrow\bullet\times$ : $t_\circ :=$ remove$(r, t_\circ); t_\bullet :=$ remove$(r, t_\bullet);$
            $s := 1; q := 0;$
   $\bullet\rightsquigarrow r\times$ : $t_\circ :=$ remove$(r, t_\circ); t_\bullet :=$ remove$(r, t_\bullet);$
            $s := 0; q := 0;$
  end receive;
  $q := q - 1$
};
$turn := (turn + 1)$ mod 2;
if $((turn = 1)$ or $(s = 1))$
then $\{r :=$ head$(t_\bullet); t_\bullet :=$ append(tail$(t_\bullet), [r])\}$
else $\{r :=$ head$(t_\circ); t_\circ :=$ append(tail$(t_\circ), [r])\}$
}
```

Fig. 6. Round-robin scheduler

execution of thread r. It is not difficult to show that this schedulers complies to the assumptions from Section 3.2, and that it is noninterferent.

Suppose the initial scheduler memory is formed according to $\nu_{init} = \nu[t_\circ \mapsto \{c\}, t_\bullet \mapsto \emptyset, t_e \mapsto \emptyset, r \mapsto 1, s \mapsto \circ, q \mapsto 0]$ for some fixed ν. Security for programs is defined as a form of noninterference:

Definition 4. *Program c is* secure *if for all σ, m_1, and m_2 where σ is noninterferent and $m_1 =_L m_2$, we have*

$$\langle\!|c, m_1, \sigma, \nu_{init}|\!\rangle \Downarrow cfg_1 \;\; \& \;\; \langle\!|c, m_2, \sigma, \nu_{init}|\!\rangle \Downarrow cfg_2 \Longrightarrow m(cfg_1) =_L m(cfg_2)$$

A form of scheduler independence is built in the definition by the universal quantification over all noninterferent schedulers. Although the universally quantified condition may appear difficult to guarantee, we will show that the security type system from Section 5 ensures that any typable program is secure. Note that this security definition is *termination-insensitive* [SM03] in that it ignores nonterminating program runs. Our approach can be applied to termination-sensitive security in a straightforward manner, although this is beyond the scope of this paper.

$$\frac{\forall v \in \mathit{FV}(e).\Gamma(v) = \mathit{low}}{\Gamma \vdash e : \mathit{low}} \qquad \frac{\exists v \in \mathit{FV}(e).\Gamma(v) = \mathit{high}}{\Gamma \vdash e : \mathit{high}}$$

$$\frac{}{\Gamma \vdash \texttt{skip} : \Gamma(\mathit{hc})} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \sqcup \Gamma(\mathit{pc}) \sqcup \Gamma(\mathit{hc}) \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e : \Gamma(\mathit{hc})}$$

$$\frac{\Gamma \vdash c_1 : \tau_1 \qquad \Gamma_{\mathit{hc}}, \mathit{hc} \mapsto \tau_1 \vdash c_2 : \tau_2}{\Gamma \vdash c_1; c_2 : \tau_2} \qquad \frac{\Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \mathit{high} \vdash c : \tau}{\Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \mathit{low} \vdash c : \tau}$$

$$\frac{\Gamma \vdash e : \tau_e \qquad \tau_e \sqsubseteq \Gamma(\mathit{hc}) \qquad (\Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \tau_e \sqcup \Gamma(\mathit{pc}) \sqcup \Gamma(\mathit{hc}) \vdash c_i : \Gamma(\mathit{hc}))_{i=1,2}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma(\mathit{hc})}$$

$$\frac{\Gamma \vdash e : \tau_e \qquad \tau_e \sqsubseteq \Gamma(\mathit{hc}) \qquad \Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \tau_e \sqcup \Gamma(\mathit{pc}) \sqcup \Gamma(\mathit{hc}) \vdash c : \Gamma(\mathit{hc})}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \Gamma(\mathit{hc})}$$

$$\frac{\Gamma(\mathit{pc}) = \mathit{low} \qquad \Gamma(\mathit{hc}) = \mathit{low}}{\Gamma \vdash \texttt{hide} : \mathit{high}} \qquad \frac{\Gamma(\mathit{pc}) = \mathit{low} \qquad \Gamma(\mathit{hc}) = \mathit{high}}{\Gamma \vdash \texttt{unhide} : \mathit{low}}$$

$$\frac{\Gamma \vdash c : \mathit{low} \qquad \Gamma(\mathit{hc}) = \mathit{low} \qquad \Gamma \vdash \vec{d} : \mathit{low}}{\Gamma \vdash \texttt{fork}(c, \vec{d}) : \mathit{low}}$$

$$\frac{\Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \Gamma(\mathit{hc}) \vdash c : \mathit{high} \qquad \Gamma(\mathit{hc}) = \mathit{high} \qquad \Gamma_{\mathit{pc}}, \mathit{pc} \mapsto \Gamma(\mathit{hc}) \vdash \vec{d} : \mathit{high}}{\Gamma \vdash \texttt{hfork}(c, \vec{d}) : \mathit{high}}$$

Fig. 7. Security type system

As common, noninterference can be expressed for a general security lattice $\mathcal{L}$ by quantifying over all security levels $\ell \in \mathcal{L}$ and demanding two-level noninterference between data at levels ℓ_1 such that $\ell_1 \sqsubseteq \ell$ (acting as low) and data at levels ℓ_2 such that $\ell_2 \not\sqsubseteq \ell$ (acting as high).

5 Security type system

This section presents a security type system that enforces the security specification from the previous section. We proceed by going over the typing rules and stating the soundness theorem.

5.1 Typing rules

Figure 7 displays the typing rules for expressions and commands. Suppose Γ is a *typing environment* which includes security type information for variables (whether they are low or high) and two variables, *pc* and *hc*, ranging over security types (low or high). By convention, we write Γ_v for Γ restricted to all variables *but* v.

Expression typing judgments have the form $\Gamma \vdash e : \tau$ where τ is low only if all variables in e (denoted $FV(e)$) are low. If there exists a high variable that occurs in e then τ must be $high$. Expression types make no use of type variables *pc* and *hc*.

Command typing judgments have the form $\Gamma \vdash c : \tau$. As a starting point, let us see how the rules track sequential-style information flow. The assignment rule ensures that information cannot leak *explicitly* by assigning an expression that contains high variables into a low variable. Further, *implicit* flows are prevented by the program-counter mechanism [DD77, VSI96]. This mechanism ensures that no assignments to low variables are allowed in the branches of a control statement (`if` or `while`) when the guard of the control statement has type $high$. (We call such `if`'s and `while`'s *high*.) This is achieved by the program-counter type variable *pc* from the typing context Γ. The intended guarantee is that whenever $\Gamma_{pc}, pc \mapsto high \vdash c : \tau$ then c may not assign to low variables. The typing rules ensure that branches of high `if`'s and `while`'s may only be typed in a high *pc* context.

Security type variables *hc* (that describes *hiding context*) and τ (that describes the command type) help tracking information flow specific to the multithreaded setting. The main job of these variables is to record whether the current thread is in the high part of the threadpool ($hc = high$) or is in the low part ($hc = low$). Command type τ reflects the level of the hiding context after the command execution.

The type rules for `hide` and `unhide` raise and lower the level of the thread, respectively. Condition $\tau_e \sqsubseteq \Gamma(hc)$ for typing high `if`'s and `while`'s ensures that high control commands can only be typed under high *hc*, which enforces the requirement that high control statements should be executed by high threads.

The type system ensures that there are no `fork` (but possibly some `hfork`) commands in high control statements. This is entailed by the rule for `fork`, which requires low *hc*.

By removing the typing rules for `hide`, `unhide`, `hfork`, and the security type variables *hc* and τ from Figure 7, we obtain a standard type system for securing information flow in sequential programs (cf. [VSI96]). This illustrates that our type provides a general technique for modular extension of systems that track information flow in a sequential setting.

Extending the type system to an arbitrary security lattice $\mathcal{L}$ is straightforward: the main modification is that security levels ℓ in $\texttt{hide}_\ell$, $\texttt{unhide}_\ell$, and $\texttt{fork}_\ell$ may be allowed only if the level of *hc* is also ℓ.

5.2 Soundness

We enlist some helpful lemmas for proving the soundness of the type system. The proofs of all lemmas, theorems, and corollaries are reported in the appendix. The first lemma states that high control commands must be typed with high *hc*.

Lemma 1. *If* $\Gamma \vdash c : \tau$*, where* $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ *or* $c = \texttt{while}\ e\ \texttt{do}\ c$*, and* $\Gamma \vdash e : high$*, then* $\Gamma(hc) = high$*.*

The following lemma states that commands with $high$ guards and `hfork`s cannot contain `hide` or `unhide` commands as part of them.

Lemma 2. *If $\Gamma_{hc,pc}, pc \mapsto high, hc \mapsto high \vdash c : high$, then c does not contain* `hide` *and* `unhide`.

The following lemma states that threads in the high part of the threadpool do not update low variables.

Lemma 3. *If $\Gamma_{hc}, hc \mapsto high \vdash c : \tau$ and $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$, then $m =_L m'$ and $\alpha \notin \{\circ, \rightsquigarrow \bullet\}$.*

The next lemma states that threads created by `hfork` always remain in the high part of the threadpool.

Lemma 4. *If $\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash c : high$ and $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$ and $c' \neq$* `stop`*, then $\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash c' : high$.*

As stated by the following lemma, threads that are moved to the low part of the threadpool are kept in the high part of it until an `unhide` instruction is executed.

Lemma 5. *If $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c : low$ for some given τ_c and $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$, where $c' \neq$* `stop` *and $\alpha \neq \bullet \rightsquigarrow r$, then $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c' : low$.*

The following lemma states that threads in the low part of the threadpool preserve low-equivalence of memories.

Lemma 6. *For a given command c such that $\Gamma_{hc}, hc \mapsto low \vdash c : low$, memories m_1 and m_2 such that $m_1 =_L m_2$, and $\langle\!| c, m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c', m_1' |\!\rangle$; it holds that $\langle\!| c, m_2 |\!\rangle \xrightarrow{\alpha} \langle\!| c', m_2' |\!\rangle$ and $m_1' =_L m_2'$.*

The next lemma states that threads remain in the low part of the threadpool as long as no `hide` instruction is executed.

Lemma 7. *If $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto low \vdash c : low$ for some given τ_c and $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$, where $c' \neq$* `stop` *and $\alpha \neq r \rightsquigarrow \bullet$, then $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto low \vdash c' : low$.*

Another important lemma is that commands `hide` and `unhide` are matched in pairs.

Lemma 8. *If $\Gamma_{hc}, hc \mapsto low \vdash$ `hide`$; c : low$, then there exist commands c' and p such that $c \in \{c';$`unhide`, `unhide`, $c';$`unhide`$;p$, `unhide`$;p\}$, where c' has no* `unhide` *commands.*

In order to establish the security of typable commands, we need to firstly identify the following subpools of threads from a given configuration.

Definition 5. *Given a scheduler memory ν and a thread pool $\vec{c}$, we define the following subpools of threads: $L(\vec{c}, \nu) = \{c_i\}_{i \in t_\circ \cap N(\vec{c})}$, $H(\vec{c}, \nu) = \{c_i\}_{i \in t_\bullet \cap N(\vec{c})}$, and $EL(\vec{c}, \nu) = \{c_i\}_{i \in t_e \cap N(\vec{c})}$.*

These three subpools of threads, $L(\vec{c})$ (*low*), $H(\vec{c})$ (*high*) and $EL(\vec{c})$ (*eventually low*), behave differently when the overall threadpool is run with low-equivalent initial memories. Threads from the low subpool match in the two runs, threads from the high subpool do not necessarily match (but they cannot update low memories in any event), and threads from the eventually low subpool will *eventually match*. The above intuition is captured by the following theorem. First, we define what "eventually match" means.

Definition 6. *Given a command p, we define the relation* eventually low, *written $\sim_{el,p}$, on empty or singleton sets of threads as follows:*

- $\emptyset \sim_{el,p,\emptyset} \emptyset$;
- $\{c\} \sim_{el,p,\{n\}} \{d\}$ *if $N(c) = N(d) = n$, and there exist commands c' and d' without* `unhide` *instructions such that $c \in \{c'; \texttt{unhide}, \texttt{unhide}\}$ and $d \in \{d'; \texttt{unhide}, \texttt{unhide}\}$ or $c \in \{c'; \texttt{unhide}; p, \texttt{unhide}; p\}$ and $d \in \{d'; \texttt{unhide}; p, \texttt{unhide}; p\}$.*

Two traces that start with low-indistinguishable memories might differ on commands (although keeping the command type). We need to show that this difference will not affect the sequence of low-observable events and low-observable memory changes. In order to show this, we define an *unwinding* [GM84] property, which is similar to the low-bisimulation property for schedulers. This unwinding property below establishes an invariant on two configurations that is preserved by low steps in lock-step and is unchanged by high steps with any of the configurations.

Theorem 1. *Given a command p and the multithreaded configurations $\langle\vec{c_1}, m_1, \sigma_1, \nu_1\rangle$ and $\langle\vec{c_2}, m_2, \sigma_2, \nu_2\rangle$ so that $m_1 =_L m_2$, written as $R_1(m_1, m_2)$, $N(\vec{c_1}) = H(\vec{c_1}, \nu_1) \cup L(\vec{c_1}, \nu_1) \cup EL(\vec{c_1}, \nu_1)$, written as $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, sets $H(\vec{c_1}, \nu_1)$, $L(\vec{c_1}, \nu_1)$, and $EL(\vec{c_1}, \nu_1)$ are disjoint, written as $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $L(\vec{c_1}, \nu_1) = L(\vec{c_2}, \nu_2)$, written as $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, written as $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $(\Gamma[\mathsf{hc} \mapsto low] \vdash c_i : low)_{i \in L(\vec{c_1}, \nu_1)}$, written as $R_6(\vec{c_1}, \nu_1)$, $(\Gamma[\mathsf{hc} \mapsto high, \mathsf{pc} \mapsto high] \vdash c_i : high)_{i \in H(\vec{c_1}, \nu_1) \cup H(\vec{c_2}, \nu_2)}$, written as $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $(\Gamma[\mathsf{hc} \mapsto high] \vdash c_i : low)_{i \in EL(\vec{c_1}, \nu_1) \cup EL(\vec{c_2}, \nu_2)}$, written as $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, and $\langle\sigma_1, \nu_1\rangle \sim_L \langle\sigma_2, \nu_2\rangle$, written as $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, then:*

i) if $\langle\vec{c_i}, m_i, \sigma_i, \nu_i\rangle \xrightarrow{\alpha} \langle\vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i\rangle$ where α is high and $i \in \{1, 2\}$, then there exists p' such that $R_1(m'_i, m_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, $R_7(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, and $R_9(\sigma'_i, \nu'_i, \sigma_{3-i}, \nu_{3-i})$;

ii) if the above case cannot be applied, and if $\langle\vec{c}_i, m_i, \sigma_i, \nu_i\rangle \xrightarrow{\alpha} \langle\vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i\rangle$ where α is low and $i \in \{1, 2\}$, then $\langle\vec{c}_{3-i}, m_{3-i}, \sigma_{3-i}, \nu_{3-i}\rangle \xrightarrow{\alpha} \langle\vec{c}\,'_{3-i}, m'_{3-i}, \sigma'_{3-i}, \nu'_{3-i}\rangle$ where there exists p' such that $R_1(m'_i, m'_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_5(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, $R_7(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, and $R_9(\sigma'_i, \nu'_i, \sigma'_{3-i}, \nu'_{3-i})$.

Corollary 1 (Soundness). *If $\Gamma_{\mathsf{hc}}, \mathsf{hc} \mapsto low \vdash c : low$ then c is secure.*

6 Extension to cooperative schedulers

It is possible to extend our model to cooperative schedulers. This is done by a minor modification of the semantics and type system rules. One can show that the results from Section 5 are preserved under these modifications.

The language is extended with primitive `yield` whose semantics is as follows:

$$\langle\!|\texttt{yield}, m|\!\rangle \xrightarrow{\not\rightsquigarrow} \langle\!|\texttt{stop}, m|\!\rangle$$

The semantics for commands also needs to propagate label $\not\rightsquigarrow$ in the sequential composition rules.

Event $\not\rightsquigarrow$ signals to the scheduler that the current thread yields control. The scheduler semantics needs to react to such an event by resetting counter q' to 0:

$$\frac{q > 0 \qquad q' = 0}{\langle\!|\sigma, \nu|\!\rangle \xrightarrow{r\not\rightsquigarrow} \langle\!|\sigma', \nu'|\!\rangle} \qquad \frac{q > 0 \qquad q' = 0 \qquad \forall\alpha \in \{\bullet, \circ\}. t'_\alpha = t_\alpha \backslash \{r\}}{\langle\!|\sigma, \nu|\!\rangle \xrightarrow{r\not\rightsquigarrow\times} \langle\!|\sigma', \nu'|\!\rangle}$$

We need to ensure that the only possibility to schedule another thread is by generating event $\not\rightsquigarrow$. Hence, we add premise $q' = \infty$ to the semantics rules for schedulers that handle events $\uparrow_\bullet r'$ and $\uparrow_\circ r'$. Additionally, the last rule in Figure 4 now allows α to range over $\{r \rightsquigarrow \bullet, \bullet \rightsquigarrow r, r \not\rightsquigarrow\}$, which propagates yielding events $\not\rightsquigarrow$ from threads to the scheduler. Similar to scheduler events $r \rightsquigarrow \bullet\times$ and $\bullet \rightsquigarrow r\times$, a new transition is added to the threadpool semantics to include the case when `yield` is executed as the last command by a thread.

At the type-system level, yielding control while inside a high control command, as well as inside `hide`/`unhide` pairs, is potentially dangerous. These situations are avoided by a type rule for `yield` that restricts *pc* and *hc* to low:

$$\frac{\Gamma(pc) = low \qquad \Gamma(hc) = low}{\Gamma \vdash \texttt{yield} : \Gamma(hc)}$$

A theorem that implies soundness for the modified type system can be proved similarly to Theorem 1.

Recently, we have suggested a mechanism for enforcing security under cooperative scheduling [RS06b]. Besides checking for explicit and implicit flows, the mechanism ensures that there are no `yield` commands in high context. Similarly, the rule above implies that `yield` may not appear in high context. On the other hand, the mechanism from [RS06b] allows no dynamic thread creation in high context. This is improved by the approach sketched in this section, because it retains the flexibility that is offered by `hfork`.

7 Ticket purchase example

In Section 2, we have argued that a flexible treatment of dynamic thread creation is paramount for a practical security mechanism. We illustrate, by an example, that the security type system from Section 5 offers such a permissive treatment without compromising security.

Consider the code fragment in Figure 8. This fragment is a part of a program that handles a ticket purchase. Variables have subscripts indicating their security levels (l for low and h for high). Suppose f_l contains public data for the flight being booked

```
...
$n_l := computeMilesFor(f_l);$
$m_h := miles(p_l);$
$s_h := statusOf(p_l);$
$o_h := s_h;$
if $(m_h + n_l > 50000)$
  then fork$(s_h := GOLD, updateStatus);$
$ok_l := printTicket(p_l, f_l, d_l);$
...
$updateStatus :$
if $(o_h \neq GOLD)$ then $changeStatus(p_l, GOLD);$
$e_h := extraMiles(m_h, n_l, s_h);$
$m_h := updateMiles(p_l, m_h + n_l + e_h)$
```

Fig. 8. Ticket purchase code

(including the class and seat details), p_l contains data for the passenger being processed. Variable n_l is assigned the (public) number of frequent-flier miles for flight f_l. Variable m_h is assigned the current number of miles of passenger p_l, which is secret. Variable s_h is assigned the (secret) status (e.g., $BASIC$ or $GOLD$) of passenger p_l. The value of s_h is then stored in o_h. Variable ok_l records if the procedure to print a ticket has been successful.

The next line is a control statement: if the updated number $m_h + n_l$ of miles exceeds 50000 then a new thread is spawn to perform a status update $updateStatus$ for the passenger. The status update code involves a computation for extra miles (due to the passenger status) and might involve a request $changeStatus$ to the status database. As potentially time-consuming computation, it is arranged in a separate thread. The final computation in the main thread prints the ticket.

This program creates threads in a high context because the guard of the `if` in the main thread depends on m_h. Furthermore, the main thread contains an assignment to a low variable (ok_l) after the instructions that branches on secrets. Because of this, the program is rejected by the type systems of Smith [Smi01] as well as Boudol and Castellani [BC01, BC02]. Nevertheless, a minor modification of the program (which can be easily automated) by replacing `if` $(m_h + n_l > 50000)$ `then fork`$(s_h := GOLD, updateStatus)$ with

```
hide;
if$(m_h + n_l > 50000)$ then hfork$(s_h := GOLD, updateStatus)$ else skip;
unhide
```

results in a typable (and therefore secure) program.

8 Feasibility study of an implementation

As discussed in Section 2, it is important that the proposed security mechanism for regulating the interaction between threads and the scheduler is feasible to put into effect in practice.

We have analyzed two well-known thread libraries: the GNU Pth [Eng05] and the NPTL [DM03] libraries for the cooperative and preemptive concurrent model, respectively. Generally, the cooperative model has been widely used in, for instance, GUI programming, when few computations are performed, and most of the time the system waits for events. The preemptive model is popular in operating systems, where preemption is essential for resource management. We have not analyzed the libraries in full detail, focusing on a feasibility study of the presented interaction between threads and the scheduler.

The GNU Pth library is well known by its high level of portability and by only using threads in user space. This library is suitable to implement the primitives `hide` and `unhide` as well as a scheduler based on the round-robin policy from Section 4. Besides reacting to the commands `hide` and `unhide`, the scheduler could be modified to include one list of threads for each security level, in this case, low and high. Such scheduler interchangeably chooses between elements of those lists depending on the value of s (i.e., low and high threads when $s = \circ$, and only high ones otherwise). Based on these ideas, the work described in [TRH07] implements the scheduler of a library that provides information-flow security for multithreaded programs.

On the other hand, the NPTL library is more complex. It maps threads in user space to threads in kernel space by using low-level primitives in the code. Nevertheless, it would be possible to apply the similar modifications that we described for the GNU Pth library. The interaction between threads and the scheduler becomes more subtle in this model due to the operations performed at the kernel space. The responsiveness of the kernel for the whole system would depend on temporal properties of code wrapped by `hide` and `unhide` primitives.

9 Synchronization primitives

Synchronization mechanisms are of fundamental importance to concurrent programs. We focus on *semaphores* [Dij02] because they are simple yet widely used synchronization primitives. In principle, the language described in Section 3 allows synchronization of threads by implementing *busy waiting* algorithms. While making synchronization possible, these algorithms also introduce performance degradation. Conversely, *blocked waiting*, which commonly underlies semaphore implementations, does not have this drawback. Semaphores, and generally any other mechanism based on *blocked waiting*, can potentially affect the security of programs. Therefore, it is important to provide policies regarding the utilization of such primitives in order to guarantee confidentiality. In this section, we extend the language, semantics and type system described previously to include semaphores primitives and provably show that noninterference is preserved for well-typed programs.

9.1 Extended language

The extended syntax of the language is displayed in Figure 9. A semaphore is a special variable, written sem, that ranges over nonnegative integers and can only be manipulated by two commands: `wait`(sem) and `signal`(sem). We assume, without losing

$$
\begin{aligned}
c ::= {} & \texttt{stop} \mid \texttt{skip} \mid v := e \mid c; c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ b\ \texttt{do}\ c \\
& \mid \texttt{hide} \mid \texttt{unhide} \mid \texttt{fork}(c, \vec{d}) \mid \texttt{hfork}(c, \vec{d}) \mid \texttt{wait}(sem) \mid \texttt{signal}(sem)
\end{aligned}
$$

Fig. 9. Extended command syntax

$$
\frac{\langle sem, m\rangle \downarrow 0}{\langle \texttt{wait}(sem), m\rangle \overset{b(sem)}{\rightharpoonup} \langle \texttt{stop}, m\rangle}
\qquad
\frac{\langle sem, m\rangle \downarrow n \quad n > 0}{\langle \texttt{wait}(sem), m\rangle \rightharpoonup \langle \texttt{stop}, m[sem \mapsto n-1]\rangle}
$$

$$
\langle \texttt{signal}(sem), m\rangle \overset{u(sem)}{\rightharpoonup} \langle \texttt{stop}, m\rangle
$$

Fig. 10. Semantics for `wait`() and `signal`()

$$
\frac{q = 0 \quad s = \circ \quad q' > 0 \quad r' \in (t_\circ \cup t_\bullet)\backslash t_w}{\langle \sigma, \nu\rangle \overset{\uparrow_\circ r'}{\rightharpoondown} \langle \sigma', \nu'\rangle}
\qquad
\frac{q = 0 \quad q' > 0 \quad r' \in (t_\bullet \cup t_e)\backslash t_w}{\langle \sigma, \nu\rangle \overset{\uparrow_\bullet r'}{\rightharpoondown} \langle \sigma', \nu'\rangle}
$$

$$
\frac{q' = q' - 1 \qquad t'_w = t_w \cup \{r\}}{\langle \sigma, \nu\rangle \overset{b^r}{\rightharpoondown} \langle \sigma', \nu'\rangle}
\qquad
\frac{q' = q' - 1 \qquad t'_w = t_w \backslash \{a\}}{\langle \sigma, \nu\rangle \overset{u^r_a}{\rightharpoondown} \langle \sigma', \nu'\rangle}
$$

$$
\frac{q' = q' - 1 \qquad \forall \alpha \in \{\bullet, \circ\}. t'_\alpha = t_\alpha \backslash \{r\}}{\langle \sigma, \nu\rangle \overset{b^r \times}{\rightharpoondown} \langle \sigma', \nu'\rangle}
$$

$$
\frac{q' = q' - 1 \qquad t'_w = t_w \backslash \{a\} \qquad \forall \alpha \in \{\bullet, \circ\}. t'_\alpha = t_\alpha \backslash \{r\}}{\langle \sigma, \nu\rangle \overset{u^r_a \times}{\rightharpoondown} \langle \sigma', \nu'\rangle}
$$

Fig. 11. Extended semantics for schedulers

generality, that every semaphore variable is initialized with 0. The semantics for these commands (in the line of [Sab01]) is shown in Figure 10. Command `wait`(sem) blocks a thread if sem has a value of 0, indicated by event $\overset{b(sem)}{\rightharpoonup}$, or otherwise decrements its value by 1. Command `signal`(sem) triggers event $\overset{u(sem)}{\rightharpoonup}$.

9.2 Extended semantics for schedulers

Threads that are blocked on semaphore variables cannot be scheduled. Clearly, schedulers need to know when threads are blocked (or not) in order to decide if they can be chosen to run. For this purpose, we introduce a new scheduler variable t_w that stores the set of blocked threads. The semantic rules involving this variable are shown in Figure 11. Rules for selecting threads to run, represented by events $\uparrow_\circ r'$ and $\uparrow_\bullet r'$, are adapted to rule out blocked threads. Observe how threads placed in t_w are removed

$$\frac{\langle c_r, m\rangle \overset{b(sem)}{\rightharpoonup} \langle c'_r, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{b^r}{\rightarrow} \langle \sigma', \nu'\rangle \qquad w(sem) = \vec{d}}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{b^r_{sem}}{\longrightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n, m', \sigma', \nu', w[sem \mapsto \vec{d c'_r}]\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{u(sem)}{\rightharpoonup} \langle c'_r, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{u^r_a}{\rightarrow} \langle \sigma', \nu'\rangle \qquad w(sem) = c_a\vec{d}}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{u^r_{sem}}{\longrightarrow} \langle c_1 \dots c_{r-1} c'_r c_{r+1} \dots c_n c_a, m', \sigma', \nu', w[sem \mapsto \vec{d}]\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{u(sem)}{\rightharpoonup} \langle c'_r, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{u^r_r}{\rightarrow} \langle \sigma', \nu'\rangle \qquad w_{sem} = \langle\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{u^r_{sem}}{\longrightarrow} \langle c_1 \dots c_{r-1} c'_r c_{r+1} \dots c_n c_a, m', \sigma', \nu', w\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{b(sem)}{\rightharpoonup} \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{b^r\times}{\rightarrow} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{b^r_{sem}\times}{\longrightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n, m', \sigma', \nu', w\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{u(sem)}{\rightharpoonup} \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{u^r_a\times}{\rightarrow} \langle \sigma', \nu'\rangle \qquad w(sem) = c_a\vec{d}}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{u^r_{sem}\times}{\longrightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n c_a, m', \sigma', \nu', w[sem \mapsto \vec{d}]\rangle}$$

$$\frac{\langle c_r, m\rangle \overset{u(sem)}{\rightharpoonup} \langle \mathtt{stop}, m'\rangle \qquad \langle \sigma, \nu\rangle \overset{u^r_r\times}{\rightarrow} \langle \sigma', \nu'\rangle \qquad w(sem) = \langle\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu, w\rangle \overset{u^r_{sem}\times}{\longrightarrow} \langle c_1 \dots c_{r-1} c_{r+1} \dots c_n c_a, m', \sigma', \nu', w\rangle}$$

Fig. 12. Threadpool semantics for semaphores primitives

from the possible values of r'. Events b^r and u^r_a indicate to the scheduler that threads r and a have been blocked and unblocked, respectively. Events $b^r\times$ and $u^r_a\times$ provide to the scheduler the same information as events b^r and u^r_a together with the fact that thread r has terminated.

9.3 Extended semantics for threadpools

The action of blocking and unblocking threads occurs at the level of threadpool configurations. For that reason, such configurations are extended with FIFO queues of waiting threads. More precisely, the extended threadpool configurations have the form $\langle \vec{c}, m, \sigma, \nu, w\rangle$ where w is a function from semaphores to a list of blocked threads. Semantic rules in Figure 4 are easily extended to consider w into account and therefore we omit the details here. Observe that the extended version of those rules do not modify w since they do not block or unblock threads at all.

Semantic rules for semaphore operations at the level of threadpools are shown in Figure 12. Event b^r_{sem} is triggered when the top level configuration receives a $b(sem)$ signal and the blocked thread is placed at the end of the queue associated with sem. When a $u(sem)$ signal is generated by the running thread, it awakes the first thread in the queue associated with sem and triggers event u^r_{sem}. Moreover, it communicates to

```
fork(skip, wait($s_2$); $l := 0$; signal($p$); signal($f$));
fork(skip, wait($s_1$); $l := 1$; signal($p$); signal($f$));
 if $h \geq 0$ then signal($s_1$); wait($p$); signal($s_2$)
         else signal($s_2$); wait($p$); signal($s_1$);
wait($p$); wait($f$); wait($f$);
```

Fig. 13. Attack using semaphores

the scheduler which thread has been awakened with event u^r_a. In case that the queue associated with sem is empty, no thread is awakened and the scheduler is informed about that by event u^r_r. Events $b^r_{sem}\times$ and $u^r_{sem}\times$ are triggered when threads terminate with synchronization commands under circumstances similar to b^r_{sem} and u^r_{sem}, respectively.

9.4 Attacks using semaphores

Confidentiality of data might be compromised if commands related to semaphores are freely allowed in programs. To illustrate this, we show an attack in Figure 13. The program contains semaphore variables s_1, s_2, p, and f, and variables h and l to store secret and public data, respectively. The code blocks and unblocks threads that assign to public variables in an order that depends on h. That is, the execution of $l := 1$ is followed by $l := 0$ when $h \geq 0$, and $l := 0$ is followed by $l := 1$ otherwise. Observe that the branching command presents no timing differences. Nevertheless, some information about h is revealed. Restrictions on the use of semaphores are needed in order to avoid such leaks.

9.5 Extended security specification

In Section 4, we state that a program is secure under some scheduler if for any two initial low-equivalence memories, whenever the two runs of the program terminate, then the resulting memories are also low-equivalent. Since semaphores variables are stored in programs memories as any other variables, the low-equivalent relation, as defined previously, is enough to capture the attacker's view of memories even in the presence of semaphores. However, the notion of "configuration cfg *terminates* in configuration cfg'" needs to be adapted. An entire program terminates if there are no blocked threads and no threads to schedule. More precisely, $cfg \Downarrow cfg'$ if $cfg \rightarrow^* cfg'$ where the thread-pool of cfg' is empty and the waiting queue $w(sem)$ is empty for every semaphore sem.

To maintain the assumption that schedulers are not leaky, it is necessary to extend the low-bisimulation defined in Section 4 with the events related to synchronization. The distinguishability level of events b^r_{sem}, u^r_{sem}, $b^r_{sem}\times$, and $u^r_{sem}\times$ is the same as the security level of thread r. Definitions 1, 2, and 3 can be easily extended to consider such events and we therefore omit the details here.

We introduce a low-equivalence relation on queues of waiting threads. We define such relation as $=_L$ where $w_1 =_L w_2$ if for every low semaphore sem, it holds that $w_1(sem) = w_2(sem)$. We are now in condition to present the extended security specification:

$$\frac{\Gamma(hc) \sqsubseteq \Gamma(sem)}{\Gamma \vdash \texttt{signal}(sem) : \Gamma(hc)} \qquad \frac{\Gamma(sem) = \Gamma(hc)}{\Gamma \vdash \texttt{wait}(sem) : \Gamma(hc)}$$

Fig. 14. Typing rules from synchronization primitives

d_1 : $\texttt{signal}(s_l); \texttt{if}\ h \geq 0\ \texttt{then}\ \texttt{wait}(s_l)\ \texttt{else}\ \texttt{skip};$
d_2 : $\texttt{sleep(30)}; \texttt{wait}(s_l); l := 0$
d_3 : $\texttt{sleep(60)}; l := 1; \texttt{signal}(s_l);$

Fig. 15. Waiting on low semaphores in high threads

Definition 7. *Program c is* secure *if for all σ, m_1, m_2, w_1, and w_2 where σ is noninterferent, $m_1 =_L m_2$, and $w_1 =_L w_2$, we have*

$$\langle\!| c, m_1, \sigma, \nu_{init}, w_1 |\!\rangle \Downarrow cfg_1 \ \ \& \ \ \langle\!| c, m_2, \sigma, \nu_{init}, w_2 |\!\rangle \Downarrow cfg_2 \Longrightarrow m(cfg_1) =_L m(cfg_2)$$

9.6 Extended type system

The type system proposed in Section 5 is extended to enforce secure uses of semaphores. As for variables, semaphores have security types (low or $high$) associated with them, which are included in the typing environment Γ. Typing rules for semaphore commands are depicted in Figure 14. The first rule establishes that signals to any semaphore can be performed in low threads. However, signals to public semaphores cannot be sent from high threads. To illustrate why this restriction is imposed, we can think about signals on low semaphores as updates on low variables, which must be avoided inside of high threads. The second rule imposes that threads can only wait on semaphores that matches their security level. Waiting on semaphores at security level ℓ in threads of security level ℓ', where $\ell \neq \ell'$, might affect the timing behavior of threads at security level ℓ and ℓ'. For instance, waiting on high semaphores in low threads might affect low threads timing behavior depending on some secret data and lead to internal timing leaks. Moreover, waiting on low semaphores in high threads might affect, also through internal timing, how assignments to public variables are performed. To illustrate this, we show an example in Figure 15. The code involves high thread d_1, low threads d_2 and d_3, low semaphore s_l, and variables h and l to store secret and public information, respectively. Let us assume a scheduler that picks thread d_1 first and then proceeds to run threads for 15 steps before yielding the control. In this case, d_1 terminates before yielding. After that, depending on the secret, two scenarios are possible. If $h \geq 0$, then d_2 blocks until d_3 completes its execution and produces 0 as the final value of l. If $h < 0$, on the other hand, d_2 is likely to execute $l := 0$ before d_3 runs $l := 1$. The final value of l is then 1, which demonstrates that the program is insecure.

The restrictions enforced by the type system are summarized in Figure 16. The first and second columns describe the use of $\texttt{wait}()$ and $\texttt{signal}()$, respectively. The first and second rows describe the use of semaphores in low and high threads, respectively. In addition, s_l (resp., s_h) means that low (resp., high) semaphores can be safely used.

	`wait()`	`signal()`
low thread	s_l	s_l, s_h
high thread	s_h	s_h

Fig. 16. Secure use of semaphores

9.7 Soundness of the extension

It is straightforward to see that the lemmas in Section 5.2 hold for the extended language. In fact, the requirements on their typing rules can be thought as requirements for assignments of some variables where their security levels have hc as lower bound. This condition is weaker than the one applied in the typing rule for assignments. Consequently, it is not surprising that every lemma holds considering the synchronization primitives `wait` and `signal`.

In order to prove the security of typable commands, we define an operator $N(w)$ that returns, for every semaphore sem, the thread identifiers in $w(sem)$. We then identify the following subpools of blocked threads for a given configuration.

Definition 8. *Given a scheduler memory ν and a function w from semaphores to a list of blocked threads, we define the following subpools of blocked threads: $BL(w, \nu) = \{c_i\}_{i \in t_\circ \cap N(w)}$, $BH(w, \nu) = \{c_i\}_{i \in t_\bullet \cap N(w)}$, and $BEL(w, \nu) = \{c_i\}_{i \in t_e \cap N(w)}$.*

Definition 6 is extended to include the fact that eventually low threads might be blocked on high semaphores. The notion of "eventually match" is now described in terms of tuples. The status, blocked or unblocked, of such threads depends on which components of the tuples they are situated. More precisely, we have the following definition:

Definition 9. *Given a command p, we define the relation* eventually low*, written $\sim_{el,p}$, on tuples of empty or singleton sets of threads as follows:*

$$\emptyset, \emptyset \sim_{el,p,\emptyset} \emptyset, \emptyset; \qquad \emptyset, \{c\} \sim_{el,p,\{n\}} \{d\}, \emptyset; \quad \emptyset, \{c\} \sim_{el,p,\{n\}} \emptyset, \{d\};$$
$$\{c\}, \emptyset \sim_{el,p,\{n\}} \{d\}, \emptyset; \quad \{c\}, \emptyset \sim_{el,p,\{n\}} \emptyset, \{d\}\ ;$$

where $N(c) = N(d) = n$, and there exist commands c' and d' without `unhide` instructions such that $c \in \{c'; \texttt{unhide}, \texttt{unhide}\}$ and $d \in \{d'; \texttt{unhide}, \texttt{unhide}\}$ or $c \in \{c'; \texttt{unhide}; p, \texttt{unhide}; p\}$ and $d \in \{d'; \texttt{unhide}; p, \texttt{unhide}; p\}$.

The following definition indicates that blocked high and low threads are placed in the waiting list of high and low semaphores, respectively.

Definition 10. *Given a typing environment Γ, an scheduler memory ν, and queues of blocked threads, we define $w \rhd v$ iff for any $sem \in dom(w)$ such that $w(sem) = c_{i_1} c_{i_2} \dots c_{i_k}$ where $k \geq 0$, $\{i_1, i_2, \dots, i_k\} \subseteq \nu.t_\bullet \cup \nu.t_e$ whether $\Gamma(sem) = high$, and $\{i_1, i_2, \dots, i_k\} \subseteq \nu.t_\circ$ whether $\Gamma(sem) = low$.*

This leads us to the following soundness theorem, which extends Theorem 1 with invariants R_{10-17} concerning blocked threads.

Theorem 2. *Given a command* p *and the multithreaded configurations* $\langle\!| \vec{c_1}, m_1, \sigma_1, \nu_1, w_1 |\!\rangle$ *and* $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\rangle$ *so that* $R_1(m_1, m_2)$, $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, *the eventually low relationship* $BEL(w_1, \nu_1)$, $EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, $BEL(w_2, \nu_2)$, *written as* R_5 $(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$, $R_6(\vec{c_1}, \nu_1)$, $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, $N(w_1) = BL(w_1, \nu_1) \cup BH(w_1, \nu_1) \cup BEL(w_1, \nu_1)$, *written* $R_{10}(w_1, \nu_1)$, $R_{10}(w_2, \nu_2)$, *sets* $BH(w_1, \nu_1)$, $BL(w_1, \nu_1)$, $BEL(w_1, \nu_1)$, *and* $N(\vec{c_1})$ *are disjoint, written as* $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $BL(w_1, \nu_1) = BL(w_2, \nu_2)$, *written as* $R_{12}(w_1, \nu_1, w_2, \nu_2)$, $(\Gamma[\mathbf{hc} \mapsto \mathit{low}] \vdash c_i : \mathit{low})_{i \in BL(w_1, \nu_1)}$, *written as* $R_{13}(w_1, \nu_1)$, $(\Gamma[\mathbf{hc} \mapsto \mathit{high}, \mathbf{pc} \mapsto \mathit{high}] \vdash c_i : \mathit{high})_{i \in BH(w_1, \nu_1) \cup BH(w_2, \nu_2)}$, *written as* $R_{14}(w_1, \nu_1, w_2, \nu_2)$, $(\Gamma[\mathbf{hc} \mapsto \mathit{high}] \vdash c_i : \mathit{low})_{i \in BEL(w_1, \nu_1) \cup BEL(w_2, \nu_1)}$, *written as* $R_{15}(w_1, \nu_1, w_2, \nu_2)$, $w_1 =_L w_2$, *written as* $R_{16}(w_1, w_2)$, $w_1 \rhd \nu_1$, *written as* $R_{17}(w_1, \nu_1)$, $R_{17}(w_2, \nu_2)$, *then:*

i) if $\langle\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\rangle \xrightarrow{\alpha} \langle\!| \vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i, w'_i |\!\rangle$ *where* α *is high and* $i \in \{1, 2\}$, *then there exists* p' *such that* $R_1(m'_i, m_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}\,'_i, w'_i, \nu'_i, \vec{c}_{3-i}, w_{3-i}, \nu_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, R_7 $(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, *and* $R_9(\sigma'_i, \nu'_i, \sigma_{3-i}, \nu_{3-i})$, $R_{10}(w'_i, \nu'_i)$, $R_{10}(w_{3-i}, \nu_{3-i})$, $R_{11}(w'_i, \nu'_i)$, $R_{11}(w_{3-i}, \nu_{3-i})$, $R_{12}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{13}(w'_i, \nu'_i)$, $R_{14}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{15}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{16}(w'_i, w_{3-i})$, $R_{17}(w'_i, \nu'_i)$, *and* $R_{17}(w_{3-i}, \nu_{3-i})$*;*

ii) if the above case cannot be applied, and given $\langle\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\rangle$ *where* $BEL(w_i, \nu_i) \neq \emptyset$, *then* $R_1(m_i, m_{3-i})$, $R_2(\vec{c}_i, \nu_i)$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}_i, \nu_i)$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}_i, w_i, \nu_i, w_{3-i}, \vec{c}_{3-i}, \nu_{3-i}, p)$, $R_6(\vec{c}_i, \nu_i)$, R_7 $(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, *and* $R_9(\sigma_i, \nu_i, \sigma_{3-i}, \nu_{3-i})$, R_{10} (w_i, ν_i), R_{10} (w_{3-i}, ν_{3-i}), $R_{11}(w_i, \nu_i)$, $R_{11}(w_{3-i}, \nu_{3-i})$, $R_{12}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{13}(w_i, \nu_i)$, $R_{14}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{15}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{16}(w_i, w_{3-i})$, $R_{17}(w_i, \nu_i)$, *and* $R_{17}(w_{3-i}, \nu_{3-i})$*;*

iii) if the above cases cannot be applied, and if $\langle\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\rangle \xrightarrow{\alpha} \langle\!| \vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i, w'_i |\!\rangle$ *where* α *is low and* $i \in \{1, 2\}$, *then* $\langle\!| \vec{c}_{3-i}, m_{3-i}, \sigma_{3-i}, \nu_{3-i}, w_{3-i} |\!\rangle \xrightarrow{\alpha} \langle\!| \vec{c}\,'_{3-i}, m'_{3-i}, \sigma'_{3-i}, \nu'_{3-i}, w'_{3-i} |\!\rangle$ *where there exists* p' *such that* $R_1(m'_i, m'_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_5(\vec{c}\,'_i, \nu'_i, w'_i, \vec{c}\,'_{3-i}, w'_{3-i}, \nu'_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, $R_7(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, *and* $R_9(\sigma'_i, \nu'_i, \sigma'_{3-i}, \nu'_{3-i})$, $R_{10}(w'_i, \nu'_i)$, $R_{10}(w'_{3-i}, \nu'_{3-i})$, $R_{11}(w'_i, \nu'_i)$, $R_{11}(w'_{3-i}, \nu'_{3-i})$, $R_{12}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{13}(w'_i, \nu'_i)$, $R_{14}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{15}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{16}(w'_i, w'_{3-i})$, $R_{17}(w'_i, \nu'_i)$, *and* $R_{17}(w'_{3-i}, \nu'_{3-i})$*;*

Compared with Theorem 1, Theorem 2 has new invariants, described by $R_{10} - R_{17}$, and applies the extended definition of the eventually low relationship in R_5. Intuitively, invariants R_{10} and R_{11} establish that the subpools of blocked threads introduced in Definition 8 form a partition of the blocked threads found in the configuration. Invariant R_{12} determines that the subpools of low blocked threads are the same in both configurations. Invariants $R_{13} - R_{15}$ establish the typing requirements for the subpools of blocked threads. A subpool of blocked threads at some security level is typed under the same circumstances that live threads at that security level. Observe the similarities between $R_6 - R_8$ and $R_{13} - R_{15}$. Invariant R_{16} establishes that threads blocked on low semaphores are the same in both configurations. Invariant R_{17} determines that the blocked threads present in the configuration match the blocked threads registered by the scheduler.

Corollary 2 (Soundness). *If $\Gamma_{\mathrm{hc}}, \mathrm{hc} \mapsto low \vdash c : low$ then c is secure.*

10 Conclusion

We have argued for a tight interaction between threads and the scheduler in order to guarantee secure information flow in multithreaded programs. In conclusion, we revisit the goals set in the paper's introduction and report the degree of success meeting these goals.

Permissiveness A key improvement over previous approaches is a permissive, yet secure, treatment of dynamic thread creation. Even if threads are created in a sensitive context, the flexible scheduling mechanism allows these threads to perform useful computation. This is particularly satisfying because it is an encouraged pattern to perform time-consuming computation (such as establishing network connections) in separate threads [Knu02, Mah04].

Scheduler-independence In contrast to known approaches to internal timing-sensitive approaches, the underlying security specification is robust with respect to a wide class of schedulers. However, the schedulers supported by the definition need to satisfy a form of noninterference that disallows information transfer from threads created in a sensitive context to threads with publicly observable effects. Sections 4 and 8 argue that such scheduler properties are not difficult to achieve.

Realistic semantics The underlying semantics does not appeal to the nonstandard construct `protect`. The semantics, however, features additional `hide`, `unhide`, and `hfork` primitives. In contrast to `protect`, these features are directly implementable, as discussed in Section 8.

Language expressiveness As discussed earlier, a flexible treatment of dynamic thread creation is a part of our model. So is synchronization, as elaborated in Section 9. Note that our typing rules do not force a separated use of low and high semaphores by low and high threads, respectively. For example, signaling on a high semaphore by a low thread is allowed. However, input/output primitives are also desirable features. We expect a natural extension of our model with input/output primitives on channels labeled with security levels, similarly to semaphores that operate on different security levels. For the two-point security lattice, we imagine the following extension of the type system. Low channels would allow low threads to input to low variables and to output low expressions: similarly to low semaphores s that permit low threads to execute both $\mathtt{P}(s)$ and $\mathtt{V}(s)$ operations. High channels would allow high threads to input/output data and allow low threads to output data: similarly to high semaphores that allow high threads s to perform both $\mathtt{P}(s)$ and $\mathtt{V}(s)$ operations and allow low threads to perform $\mathtt{V}(s)$. Formalizing this intuition is subject to our future work.

Practical enforcement We have demonstrated that security can be enforced for both cooperative and preemptive schedulers using a compositional type system. The type system accommodates permissive programming. We have illustrated by an example in Section 7 that the permissiveness of dynamic thread creation is not majorly restricted by the type system. The type system does not involve padding to eliminate timing leaks at the cost of efficiency.

Most recently, together with Barthe and Rezk [BRRS07], we have adapted our type system to an unstructured assembly language. Our future work plans include handling richer low-level languages (such as languages with exceptions and bytecode) and facilitating tool support for them.

Acknowledgments

We wish to thank our colleagues in the ProSec group at Chalmers, partners in the Mobius project, and the anonymous reviewers for helpful feedback. This work was funded by the Swedish research agencies SSF and VR and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[Aga00] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[BC01] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*, pages 382–395. Springer-Verlag, July 2001.

[BC02] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[BRRS07] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symp. on Research in Computer Security*, volume 4734 of *LNCS*, pages 2–18. Springer-Verlag, September 2007.

[CF07] S. N. Freund C. Flanagan. Type inference against races. *Science of Computer Programming*, 64(1):140–165, 2007.

[Coh78] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[Dij02] Edsger W. Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer-Verlag New York, Inc., 2002.

[DM03] U. Drepper and I. Molnar. The native POSIX thread library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, January 2003.

[Eng05] Ralf S. Engelschall. GNU pth - The GNU portable threads. `http://www.gnu.org/software/pth/`, November 2005.

[FG01] R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.

[HVY00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.

[HWS06] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

[HY02] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, January 2002.

[KM07] B. Köpf and H. Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security (IJIS)*, 6(2–3):107–131, March 2007.

[Knu02] J. Knudsen. Networking, user experience, and threads. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/articles/threading/`, 2002.

[Mah04] Q. H. Mahmoud. Preventing screen lockups of blocking operations. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/ttips/screenlock/`, 2004.

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proc. IEEE Symp. on Security and Privacy*, pages 161–166, April 1987.

[McL90] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

[MZZ$^+$06] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2006.

[Pot02] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[RHNS07] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference (ASIAN'06)*, LNCS. Springer-Verlag, 2007.

[Ros95] A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[RS06a] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[RS06b] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2006.

[Rya01] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of*

System Informatics, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[SBN$^+$97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sim03] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet /soft/flowcaml/`, July 2003.

[SM02] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[Smi01] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[Smi03] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

[TRH07] Ta Chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, pages 187–200, July 2007.

[VM01] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[ZM03] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

Appendix

Lemma 1. If $\Gamma \vdash c : \tau$, where $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ or $c = \texttt{while}\ e\ \texttt{do}\ c$, and $\Gamma \vdash e : high$, then $\Gamma(\mathsf{hc}) = high$.

Proof. By inspection of the typing rules for `if` and `while`. □

Lemma 2. If $\Gamma_{\mathsf{hc},\mathsf{pc}}, \mathsf{pc} \mapsto high, \mathsf{hc} \mapsto high \vdash c : high$, then c does not contain `hide` and `unhide`.

Proof. By simple induction on the typing derivation. □

Lemma 3. If $\Gamma_{hc}, hc \mapsto high \vdash c : \tau$ and $\langle c, m \rangle \xrightarrow{\alpha} \langle c', m' \rangle$, then $m =_L m'$ and $\alpha \notin \{\circ, \rightsquigarrow \bullet\}$.

Proof. By induction on the type derivation of c. □

Lemma 4. If $\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash c : high$ and $\langle c, m \rangle \xrightarrow{\alpha} \langle c', m' \rangle$ and $c' \neq \texttt{stop}$, then $\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash c' : high$.

Proof. By case analysis on c and inspection of the typing rules. □

Lemma 5. If $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c : low$ for some given τ_c and $\langle c, m \rangle \xrightarrow{\alpha} \langle c', m' \rangle$, where $c' \neq \texttt{stop}$ and $\alpha \neq \bullet \rightsquigarrow r$, then $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c' : low$.

Proof. By case analysis on c. Observe that the only typable command under the hypothesis of the lemma is the sequential composition. □

Lemma 6. For a given command c such that $\Gamma_{hc}, hc \mapsto low \vdash c : low$, memories m_1 and m_2 such that $m_1 =_L m_2$, and $\langle c, m_1 \rangle \xrightarrow{\alpha} \langle c', m_1' \rangle$; it holds that $\langle c, m_2 \rangle \xrightarrow{\alpha} \langle c', m_2' \rangle$ and $m_1' =_L m_2'$.

Proof. By case analysis on c and by exploring its type derivation. □

Lemma 7. If $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto low \vdash c : low$ for some given τ_c and $\langle c, m \rangle \xrightarrow{\alpha} \langle c', m' \rangle$, where $c' \neq \texttt{stop}$ and $\alpha \neq r \rightsquigarrow \bullet$, then $\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto low \vdash c' : low$.

Proof. By case analysis on c and inspection of the typing rules. □

Lemma 8. If $\Gamma_{hc}, hc \mapsto low \vdash \texttt{hide}; c : low$, then there exist commands c' and p such that $c \in \{c'; \texttt{unhide},\ \texttt{unhide},\ c'; \texttt{unhide}; p,\ \texttt{unhide}; p\}$, where c' has no `unhide` commands.

Proof. By induction on the size of command c. □

Theorem 1. Given a command p and the multithreaded configurations $\langle \vec{c_1}, m_1, \sigma_1, \nu_1 \rangle$ and $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle$ so that $m_1 =_L m_2$, written as $R_1(m_1, m_2)$, $N(\vec{c_1}) = H(\vec{c}_1, \nu_1) \cup L(\vec{c}_1, \nu_1) \cup EL(\vec{c}_1, \nu_1)$, written as $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, sets $H(\vec{c_1}, \nu_1)$, $L(\vec{c_1}, \nu_1)$, and $EL(\vec{c_1}, \nu_1)$ are disjoint, written as $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $L(\vec{c_1}, \nu_1) = L(\vec{c}_2, \nu_2)$, written as $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, written as $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $(\Gamma[hc \mapsto low] \vdash c_i : low)_{i \in L(\vec{c_1}, \nu_1)}$, written as $R_6(\vec{c_1}, \nu_1)$, $(\Gamma[hc \mapsto high, pc \mapsto high] \vdash c_i : high)_{i \in H(\vec{c_1}, \nu_1) \cup H(\vec{c_2}, \nu_2)}$, written as $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $(\Gamma[hc \mapsto high] \vdash c_i : low)_{i \in EL(\vec{c_1}, \nu_1) \cup EL(\vec{c_2}, \nu_2)}$, written as $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, and $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$, written as $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, then:

i) if $\langle \vec{c_i}, m_i, \sigma_i, \nu_i \rangle \xrightarrow{\alpha} \langle \vec{c}\,'_i, m_i', \sigma_i', \nu_i' \rangle$ where α is high and $i \in \{1, 2\}$, then there exists p' such that $R_1(m_i', m_{3-i})$, $R_2(\vec{c}\,'_i, \nu_i')$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}\,'_i, \nu_i')$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}\,'_i, \nu_i', \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}\,'_i, \nu_i', \vec{c}_{3-i}, \nu_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu_i')$, $R_7(\vec{c}\,'_i, \nu_i', \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}\,'_i, \nu_i', \vec{c}_{3-i}, \nu_{3-i})$, and $R_9(\sigma_i', \nu_i', \sigma_{3-i}, \nu_{3-i})$;

ii) if the above case cannot be applied, and if $\langle\!|\vec{c}_i, m_i, \sigma_i, \nu_i|\!\rangle \xrightarrow{\alpha} \langle\!|\vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i|\!\rangle$ where α is low and $i \in \{1,2\}$, then $\langle\!|\vec{c}_{3-i}, m_{3-i}, \sigma_{3-i}, \nu_{3-i}|\!\rangle \xrightarrow{\alpha} \langle\!|\vec{c}\,'_{3-i}, m'_{3-i}, \sigma'_{3-i}, \nu'_{3-i}|\!\rangle$ where there exists p' such that $R_1(m'_i, m'_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_5(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, $R_7(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, and $R_9(\sigma'_i, \nu'_i, \sigma'_{3-i}, \nu'_{3-i})$.

Proof. By case analysis on command/scheduler steps. We are only going to show the proofs for the mentioned commands when the configuration $\langle\!|\vec{c_1}, m_1, \sigma_1, \nu_1|\!\rangle$ makes some progress. We assume that the thread c_r belongs to $\vec{c_1}$. Analogous proofs are obtained when $\langle\!|\vec{c_2}, m_2, \sigma_2, \nu_2|\!\rangle$ makes progress instead. We make a distinction if the system performs an step that produces a low or a high event.

i) High events $\bullet^r_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, and $\uparrow_\bullet r'$ (where $\{r, r'\} \subseteq H(\vec{c_1}, \nu_1) \cup EL(\vec{c_1}, \nu_1)$).

$\alpha_1 = \bullet^r_{\vec{d}}$) By inspecting the semantics for threadpools and the scheduler, we know that $c_r \in H(\vec{c_1}, \nu_1)$ or $c_r \in EL(\vec{c_1}, \nu_1)$ and that $H(\vec{c'_1}, \nu'_1) = H(\vec{c}_1, \nu_1) \cup N(\vec{d})$. $R_1(m'_1, m'_2)$ holds trivially since `hfork` has no changed the memories. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler together with the fact that $N(\vec{d})$ are fresh names for threads. $R_4(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads (only high threads were created). For a similar reason, $R_6(\vec{c_1}', \nu'_1)$ also holds. $R_9(\sigma'_1, \nu'_1, \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.
In order to prove $R_5(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2, p')$, $R_7(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$, and $R_8(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$, we need to split the proof in two more cases: $c_r \in H(\vec{c_1}, \nu_1)$ and $c_r \in EL(\vec{c_1}, \nu_1)$.

$c_r \in H(\vec{c_1}, \nu_1)$) By taking $p' = p$, we have that $R_5(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2, p)$ and proposition $R_8(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ hold because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold; and because the eventually low thread, if there exists one, has made no progress. Finally, $R_7(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 4 to c_r.

$c_r \in EL(\vec{c_1}, \nu_1)$) Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds, we know that the thread with name r belongs to the threadpool $\vec{c_2}$. Moreover, we know that $c_r =$`hfork`$(c, \vec{d})$; c'; `unhide`, $c_r =$ `hfork`$(c, \vec{d})$; `unhide`, $c_r =$ `hfork`$(c, \vec{d})$; c'; `unhide`; p, or c_r = `hfork`$(c, \vec{d})$; `unhide`; p, where c' has no `unhide` commands. Then, $R_5(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2, p')$ holds by taking $p' = p$. $R_8(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ holds by Lemma 5. Finally, $R_7(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have made no progress.

$\alpha_1 = r\rightsquigarrow$) We split the proof in two more cases: $c_r \in H(\vec{c_1}, \nu_1)$ and $c_r \in EL(\vec{c_1}, \nu_1)$.

$c_r \in H(\vec{c_1}, \nu_1)$) $R_1(m'_1, m_2)$ holds by applying Lemma 3. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu'_1)$ also holds. $R_7(\vec{c'_1},$

$\nu_1', \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 4. By taking $p' = p$, we have that $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p')$ and $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ hold because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold; and because the eventually low thread, if there exists one, has made no progress. $R_9(\sigma_1', \nu_1', \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$c_r \in EL(\vec{c_1}, \nu_1)$) $R_1(m_1', m_2)$ holds by applying Lemma 5. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu_1')$ also holds. Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds, we know that $c_r = c'; \texttt{unhide}$, $c_r = \texttt{unhide}$, $c_r = c'; \texttt{unhide}; p$, or $c_r = \texttt{unhide}; p$ for some command c' without `unhide` instructions. However, $c_r \neq \texttt{unhide}; p$ and $c_r \neq \texttt{unhide}$ since $\alpha_1 = r \rightsquigarrow$. The proof proceeds similarly when $c_r = c'; \texttt{unhide}$ or $c_r = c'; \texttt{unhide}; p$. Therefore, we only show the latter case. By inspecting the semantics for commands, we know that $\langle\!| c_r, m_1 |\!\rangle \rightharpoonup \langle\!| c_r', m_1' |\!\rangle$, where $c_r' = c''; \texttt{unhide}; p$ where $\langle\!| c', m_1 |\!\rangle \rightharpoonup \langle\!| c'', m_1' |\!\rangle$ and $c'' \neq \texttt{stop}$ or $c_r' = \texttt{unhide}; p$. By taking $p' = p$, we can conclude that $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p')$ holds by Definition 6. $R_7(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not involve high threads. $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ hold by applying Lemma 5 to c_r. $R_9(\sigma_1', \nu_1', \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r \rightsquigarrow \times$) We need to split the proof in two more cases: $c_r \in H(\vec{c_1}, \nu_1)$ and $c_r \in EL(\vec{c_1}, \nu_1)$.

$c_r \in H(\vec{c_1}, \nu_1)$) $R_1(m_1', m_2)$ holds by applying Lemma 3. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu_1')$ also holds. $R_7(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the thread c_r has finished. By taking $p' = p$, we have that $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p)$ and $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ hold because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold; and because the eventually low thread, if there exists one, has made no progress. $R_9(\sigma_1', \nu_1', \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$c_r \in EL(\vec{c_1}, \nu_1)$) The eventually low thread cannot make progress and finishes immediately. Observe that c_r must be typable as $\Gamma[\mathit{hc} \mapsto \mathit{high}] \vdash c_r : \mathit{low}$ and it must terminate in one step. Therefore, $c_r = \texttt{unhide}$ but this cannot occur since $\alpha_1 = r \rightsquigarrow \times$.

$\alpha_1 = \uparrow_\bullet r'$) By taking $p' = p$, we have that $R_1(m_1', m_2)$, $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu_1')$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p')$, $R_6(\vec{c_1}', \nu_1')$, $R_7(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$, and $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ holds since $R_1(m_1, m_2)$, $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $R_6(\vec{c_1}, \nu_1)$, $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold and because the transition has only modified the variable t_r in the scheduler. $R_9(\sigma_1', \nu_1', \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

ii) Low events : $\circ^{r}_{\vec{d}}$, $r \rightsquigarrow$, $r \rightsquigarrow \times$, $\uparrow_\circ r'$, $r \rightsquigarrow \bullet$, $\bullet \rightsquigarrow r_e$, $r \rightsquigarrow \bullet \times$, and $\bullet \rightsquigarrow r_e \times$ (where $\{r, r'\} \subseteq L(\vec{c_1}, \nu_1)$ and $r_e \in t_{e_1} EL(\vec{c_1}, \nu_1)$).

$\alpha_1 = \circ^{r}_{\vec{d}}$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \texttt{fork}(c, \vec{d})$ or $c_r = \texttt{fork}(c, \vec{d}); c^*$ for some commands c and c^*. We are only going to show the proof for the case when $c_r = \texttt{fork}(c, \vec{d}); c^*$ since the proof for $c_r = \texttt{fork}(c, \vec{d})$ proceeds in a similar way. By inspecting the semantics for threadpools and commands, we have the transition $\langle c_r, m_1 \rangle \overset{\circ_{\vec{d}}}{\rightharpoonup} \langle c; c^*, m_1 \rangle$, and that $\langle \sigma_1, \nu_1 \rangle \overset{\circ^{r}_{\vec{d}}}{\rightharpoondown} \langle \sigma_1', \nu_1' \rangle$. Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \overset{\circ^{r}_{\vec{d}}}{\rightharpoondown} \langle \sigma_2', \nu_2' \rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle c_r, m_2 \rangle \overset{\circ_{\vec{d}}}{\rightharpoonup} \langle c; c^*, m_2 \rangle$. We can therefore conclude that $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle \overset{\circ^{r}_{\vec{d}}}{\rightharpoonup} \langle \vec{c_2}', m_2', \sigma_2', \nu_2' \rangle$.
$R_1(m_1', m_2')$ holds by applying Lemma 6. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since propositions $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds, and by inspecting the semantics for the scheduler together with the fact that $N(\vec{d})$ are fresh names for threads. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 added the same new low threads to both configurations. By taking $p' = p$, we have that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p)$ holds since proposition $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and because the eventually low thread, if exists one, has made no progress. $R_6(\vec{c_1}', \nu_1')$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and by inspecting the Lemma 7. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the low step α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the step α_1. Finally, proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r \rightsquigarrow$) By inspecting the semantics rules for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, $\langle c_r, m_1 \rangle \rightharpoonup \langle c', m_1' \rangle$, and that $\langle \sigma_1, \nu_1 \rangle \overset{r \rightsquigarrow}{\rightharpoondown} \langle \sigma_1', \nu_1' \rangle$. Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \overset{r \rightsquigarrow}{\rightharpoonup} \langle \sigma_2', \nu_2' \rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle c_r, m_2 \rangle \rightharpoonup \langle c', m_2 \rangle$. Therefore, we can conclude that the transition $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle \overset{r \rightsquigarrow}{\rightharpoonup} \langle \vec{c_2}', m_2', \sigma_2', \nu_2' \rangle$ holds.
$R_1(m_1', m_2')$ holds by applying Lemma 6 to c_r. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 6 to c_r. By taking $p' = p$, we have that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p)$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and because the eventually low threads, if they exist, have made no progress. $R_6(\vec{c_1}', \nu_1')$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and by applying Lemma 7 to c_r. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because

the eventually low threads in both configurations, if they exist, have been not modified by the transition α_1. Finally, $R_9(\sigma'_1, \nu'_1, \sigma'_2, \nu'_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r \rightsquigarrow \times$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, $\langle\!| c_r, m_1 |\!\rangle \rightharpoonup \langle\!| \texttt{stop}, m'_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \xrightarrow{r \rightsquigarrow \times} \langle\!| \sigma'_1, \nu'_1 |\!\rangle$. Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \xrightarrow{r \rightsquigarrow \times} \langle\!| \sigma'_2, \nu'_2 |\!\rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c_1}, \nu_1)$, and that $\langle\!| c_r, m_2 |\!\rangle \rightharpoonup \langle\!| \texttt{stop}, m_2 |\!\rangle$. We can therefore conclude that the transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2 |\!\rangle \xrightarrow{r \rightsquigarrow \times} \langle\!| \vec{c_2}', m'_2, \sigma'_2, \nu'_2 |\!\rangle$ holds.
$R_1(m'_1, m'_2)$ holds by applying Lemma 6 to c_r. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}', \nu'_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}', \nu'_2)$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler (observe that the thread c_r has just terminated). $R_4(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 6 to c_r. By taking $p' = p$, we have that $R_5(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2, p)$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and because the eventually low threads, if they exist, have made no progress. $R_6(\vec{c_1}', \nu'_1)$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and $c_r \notin L(\vec{c}'_1, \nu'_1)$. $R_7(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the transition α_1. Finally, $R_9(\sigma'_1, \nu'_1, \sigma'_2, \nu'_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = \uparrow_\circ r'$) By inspecting the semantics for threadpools and the scheduler, we have that $\langle\!| \sigma_1, \nu_1 |\!\rangle \xrightarrow{\uparrow_\circ r'} \langle\!| \sigma'_1, \nu'_1 |\!\rangle$. Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \xrightarrow{\uparrow_\circ r'} \langle\!| \sigma'_2, \nu'_2 |\!\rangle$. We can therefore conclude that the transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2 |\!\rangle \xrightarrow{\uparrow_\circ r'} \langle\!| \vec{c_2}', m'_2, \sigma'_2, \nu'_2 |\!\rangle$ holds.
Let us take $p' = p$. Then, we have that $R_1(m'_1, m'_2)$, $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}', \nu'_2)$, $R_3(\vec{c_1}', \nu'_1)$, $R_3(\vec{c_2}', \nu'_2)$,$R_4(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$, $R_5(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2, p')$, $R_6(\vec{c_1}', \nu'_1)$, $R_7(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$, $R_8(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since $R_1(m_1, m_2)$, $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $R_6(\vec{c_1}, \nu_1)$, $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition has only modified the variable t_r in the scheduler. $R_9(\sigma'_1, \nu'_1, \sigma'_2, \nu'_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r \rightsquigarrow \bullet$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r = \texttt{hide}; c^*$ for some command c^*, $\langle\!| c_r, m_1 |\!\rangle \xrightarrow{\rightsquigarrow \bullet} \langle\!| c'_r, m_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \xrightarrow{r \rightsquigarrow \bullet} \langle\!| \sigma'_1, \nu'_1 |\!\rangle$. We also know that $\langle\!| c_r, m_2 |\!\rangle \xrightarrow{\rightsquigarrow \bullet} \langle\!| c'_r, m_2 |\!\rangle$ since $L(\vec{c_1}) = L(\vec{c_2})$. Moreover, we know that $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \xrightarrow{r \rightsquigarrow \bullet} \langle\!| \sigma'_2, \nu'_2 |\!\rangle$. We can thus conclude that the transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2 |\!\rangle \xrightarrow{r \rightsquigarrow \bullet} \langle\!| \vec{c_2}', m'_2, \sigma'_2, \nu'_2 |\!\rangle$ holds.
We know that $EL(\vec{c_1}) = \emptyset$ because a low thread was scheduled to produce the event $r \rightsquigarrow \bullet$. Then, $EL(\vec{c_2}) = \emptyset$ since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds. By applying Lemma 8 to c_r, we know that $c^* = c'; \texttt{unhide}$, $c^* = \texttt{unhide}$, $c^* = c'; \texttt{unhide}; p^*$, or $c^* = \texttt{unhide}; p^*$, where c' has no $\texttt{unhide}$.
$R_1(m'_1, m'_2)$ holds since $m'_1 = m_1$ and $m'_2 = m_2$. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}', \nu'_2)$,

$R_3(\vec{c_1}', \nu_1')$, $R_3(\vec{c_2}', \nu_2')$, and $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ hold since the following equalities $EL(\vec{c_1}, \nu_1) = EL(\vec{c}_2, \nu_2) = \emptyset$ hold, $(L(\vec{c_i}', \nu_i') = L(\vec{c}_i, \nu_i) \backslash \{c_r\})_{i=1,2}$, and $(EL(\vec{c_i}', \nu_i') = \{c_r\})_{i=1,2}$ hold by inspecting the semantics for threadpools and the scheduler.
In the cases where $c^* = c'; \texttt{unhide}$ or $c^* = \texttt{unhide}$, $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p')$ holds by taking $p' = \texttt{skip}$ (see Definition 6). On the other cases, by taking $p' = p^*$, we know that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p^*)$ holds because the application of Lemma 8 gave us the appropriate p^* that satisfies Definition 6. $R_6(\vec{c_1}', \nu_1')$ holds since $L(\vec{c_1}', \nu_1') = L(\vec{c}_1, \nu_1) \backslash \{c_r\}$ and $R_6(\vec{c_1}, \nu_1)$ hold. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since proposition $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by inspecting the type derivation of c_r. Finally, proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = \bullet \rightsquigarrow r_e$) We know that $r_e \in t_{e_1}$. By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_{r_e} = \texttt{unhide}; c^*$ or $c_{r_e} = \texttt{unhide}$ for some command c^*, $c_{r_e} \in EL(\vec{c_1}, \nu_1)$, $\langle\!| c_{r_e}, m_1 |\!\rangle \overset{\bullet\rightsquigarrow}{\rightharpoondown} \langle\!| c^*, m_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoondown} \langle\!| \sigma_1', \nu_1' |\!\rangle$. We are only going to consider the case when $c_{r_e} = \texttt{unhide}; c^*$ since the proof for $c_{r_e} = \texttt{unhide}$ is analogous. Therefore, we omit the proof when $\alpha_1 = \bullet \rightsquigarrow r_e \times$.
Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoondown} \langle\!| \sigma_2', \nu_2' |\!\rangle$. Because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds, we know that $c^* = p$ and that the thread with name r_e belongs to the threadpool $\vec{c_2}$ as well. Let us call it $c_{r_e}^2$. Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and it is not possible for a thread to make progress by a high computation, we have that $c_{r_e}^2 = \texttt{unhide}; p$. As a consequence of that, it holds that $\langle\!| c_{r_e}^2, m_2 |\!\rangle \overset{\bullet\rightsquigarrow}{\rightharpoondown} \langle\!| p, m_2 |\!\rangle$. Thus, transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoondown} \langle\!| \vec{c_2}', m_2', \sigma_2', \nu_2' |\!\rangle$ holds.
$R_1(m_1', m_2')$ holds trivially since $\texttt{unhide}$ has no changed the memories. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds; and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because after the transition α_1, the threads c_{r_e} and $c_{r_e}^2$ become the thread p. By inspecting the semantics for the scheduler, we have that $EL(\vec{c_1'}, \nu_1') = EL(\vec{c}_2', \nu_2') = \emptyset$. Then, by taking $p' = \texttt{skip}$, it trivially holds that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', \texttt{skip})$. $R_6(\vec{c_1}', \nu_1')$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds ; and by inspecting the type derivation of c_{r_e}. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $EL(\vec{c_1}', \nu_1') = EL(\vec{c}_2', \nu_2') = \emptyset$. Finally, $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r \rightsquigarrow \bullet \times$) We know that $r \in t_{\circ_1}$. The hypothesis in the theorem state that c_r must be typable as $\Gamma[\mathit{hc} \mapsto \mathit{low}] \vdash c_r : \mathit{low}$ by $R_6(\vec{c_1}, \nu_1)$. Observe that when $c_r = \texttt{hide}$ this requirement is violated. Therefore, this event can never occur under the given hypothesis.

□

Corollary 1 (Soundness). If $\mathit{\Gamma}_{\mathbf{hc}}, \mathbf{hc} \mapsto \mathit{low} \vdash c : \mathit{low}$ then c is secure.

Proof. For arbitrary σ, m_1, and m_2 so that $m_1 =_L m_2$ and σ is noninterferent, assume $\langle\!\!| c, m_1, \sigma, \nu_{init} |\!\!\rangle \Downarrow cfg_1$ & $\langle\!\!| c, m_2, \sigma, \nu_{init} |\!\!\rangle \Downarrow cfg_2$. By inductive (in the number of transition steps of the above configurations) application of Theorem 1, we propagate invariant $m_1 =_L m_2$ to the terminating configurations. $\square$

Theorem 2. Given a command p and the multithreaded configurations $\langle\!\!| \vec{c_1}, m_1, \sigma_1, \nu_1, w_1 |\!\!\rangle$ and $\langle\!\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\!\rangle$ so that $R_1(m_1, m_2)$, $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, the eventually low relationship $BEL(w_1, \nu_1)$, $EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, $BEL(w_2, \nu_2)$, written as R_5 $(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$, $R_6(\vec{c_1}, \nu_1)$, $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, $N(w_1) = BL(w_1, \nu_1) \cup BH(w_1, \nu_1) \cup BEL(w_1, \nu_1)$, written $R_{10}(w_1, \nu_1)$, $R_{10}(w_2, \nu_2)$, sets $BH(w_1, \nu_1)$, $BL(w_1, \nu_1)$, $BEL(w_1, \nu_1)$, and $N(\vec{c_1})$ are disjoint, written as $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $BL(w_1, \nu_1) = BL(w_2, \nu_2)$, written as $R_{12}(w_1, \nu_1, w_2, \nu_2)$, $(\mathit{\Gamma}[\mathbf{hc} \mapsto \mathit{low}] \vdash c_i : \mathit{low})_{i \in BL(w_1, \nu_1)}$, written as $R_{13}(w_1, \nu_1)$, $(\mathit{\Gamma}[\mathbf{hc} \mapsto \mathit{high}, \mathbf{pc} \mapsto \mathit{high}] \vdash c_i : \mathit{high})_{i \in BH(w_1, \nu_1) \cup BH(w_2, \nu_2)}$, written as $R_{14}(w_1, \nu_1, w_2, \nu_2)$, $(\mathit{\Gamma}[\mathbf{hc} \mapsto \mathit{high}] \vdash c_i : \mathit{low})_{i \in BEL(w_1, \nu_1) \cup BEL(w_2, \nu_1)}$, written as $R_{15}(w_1, \nu_1, w_2, \nu_2)$, $w_1 =_L w_2$, written as $R_{16}(w_1, w_2)$, $w_1 \rhd \nu_1$, written as $R_{17}(w_1, \nu_1)$, $R_{17}(w_2, \nu_2)$, then:

i) if $\langle\!\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\!\rangle \xrightarrow{\alpha} \langle\!\!| \vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i, w'_i |\!\!\rangle$ where α is high and $i \in \{1, 2\}$, then there exists p' such that $R_1(m'_i, m_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}\,'_i, w'_i, \nu'_i, \vec{c}_{3-i}, w_{3-i}, \nu_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, R_7 $(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}_{3-i}, \nu_{3-i})$, and $R_9(\sigma'_i, \nu'_i, \sigma_{3-i}, \nu_{3-i})$, $R_{10}(w'_i, \nu'_i)$, $R_{10}(w_{3-i}, \nu_{3-i})$, $R_{11}(w'_i, \nu'_i)$, $R_{11}(w_{3-i}, \nu_{3-i})$, $R_{12}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{13}(w'_i, \nu'_i)$, $R_{14}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{15}(w'_i, \nu'_i, w_{3-i}, \nu_{3-i})$, $R_{16}(w'_i, w_{3-i})$, $R_{17}(w'_i, \nu'_i)$, and $R_{17}(w_{3-i}, \nu_{3-i})$;

ii) if the above case cannot be applied, and given $\langle\!\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\!\rangle$ where $BEL(w_i, \nu_i) \neq \emptyset$, then $R_1(m_i, m_{3-i})$, $R_2(\vec{c}_i, \nu_i)$, $R_2(\vec{c}_{3-i}, \nu_{3-i})$, $R_3(\vec{c}_i, \nu_i)$, $R_3(\vec{c}_{3-i}, \nu_{3-i})$, $R_4(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_5(\vec{c}_i, w_i, \nu_i, w_{3-i}, \vec{c}_{3-i}, \nu_{3-i}, p)$, $R_6(\vec{c}_i, \nu_i)$, R_7 $(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, $R_8(\vec{c}_i, \nu_i, \vec{c}_{3-i}, \nu_{3-i})$, and $R_9(\sigma_i, \nu_i, \sigma_{3-i}, \nu_{3-i})$, R_{10} (w_i, ν_i), R_{10} (w_{3-i}, ν_{3-i}), $R_{11}(w_i, \nu_i)$, $R_{11}(w_{3-i}, \nu_{3-i})$, $R_{12}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{13}(w_i, \nu_i)$, $R_{14}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{15}(w_i, \nu_i, w_{3-i}, \nu_{3-i})$, $R_{16}(w_i, w_{3-i})$, $R_{17}(w_i, \nu_i)$, and $R_{17}(w_{3-i}, \nu_{3-i})$;

iii) if the above cases cannot be applied, and if $\langle\!\!| \vec{c_i}, m_i, \sigma_i, \nu_i, w_i |\!\!\rangle \xrightarrow{\alpha} \langle\!\!| \vec{c}\,'_i, m'_i, \sigma'_i, \nu'_i, w'_i |\!\!\rangle$ where α is low and $i \in \{1, 2\}$, then $\langle\!\!| \vec{c}_{3-i}, m_{3-i}, \sigma_{3-i}, \nu_{3-i}, w_{3-i} |\!\!\rangle \xrightarrow{\alpha} \langle\!\!| \vec{c}\,'_{3-i}, m'_{3-i}, \sigma'_{3-i}, \nu'_{3-i}, w'_{3-i} |\!\!\rangle$ where there exists p' such that $R_1(m'_i, m'_{3-i})$, $R_2(\vec{c}\,'_i, \nu'_i)$, $R_2(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_3(\vec{c}\,'_i, \nu'_i)$, $R_3(\vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_4(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_5(\vec{c}\,'_i, \nu'_i, w'_i, \vec{c}\,'_{3-i}, w'_{3-i}, \nu'_{3-i}, p')$, $R_6(\vec{c}\,'_i, \nu'_i)$, $R_7(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, $R_8(\vec{c}\,'_i, \nu'_i, \vec{c}\,'_{3-i}, \nu'_{3-i})$, and $R_9(\sigma'_i, \nu'_i, \sigma'_{3-i}, \nu'_{3-i})$, $R_{10}(w'_i, \nu'_i)$, $R_{10}(w'_{3-i}, \nu'_{3-i})$, $R_{11}(w'_i, \nu'_i)$, $R_{11}(w'_{3-i}, \nu'_{3-i})$, $R_{12}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{13}(w'_i, \nu'_i)$, $R_{14}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{15}(w'_i, \nu'_i, w'_{3-i}, \nu'_{3-i})$, $R_{16}(w'_i, w'_{3-i})$, $R_{17}(w'_i, \nu'_i)$, and $R_{17}(w'_{3-i}, \nu'_{3-i})$;

Proof. By case analysis on command/scheduler steps. We are only going to show the proofs for commands related to synchronization and `unhide` when the configuration

$\langle \vec{c_1}, m_1, \sigma_1, \nu_1 \rangle$ makes some progress. The proof for other commands proceeds similarly as in Theorem 1. We assume that the thread c_r belongs to $\vec{c_1}$. Analogous proofs are obtained when $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle$ makes progress instead.

i) High events related to synchronization : b^r_{sem}, $b^r_{sem}\times$, u^r_{sem}, and $u^r_{sem}\times$ (where $r \in H(\vec{c_1}, \nu_1) \cup EL(\vec{c_1}, \nu_1)$).

$\alpha_1 = b^r_{sem}$)

$c_r \in H(\vec{c_1}, \nu_1)$)

$R_1(m'_1, m_2)$ holds by inspecting the semantics of threadpools and applying Lemma 3 to c_r. By inspecting the threadpool semantics and since c_r has been blocked, we have that $N(\vec{c_1}\,') = N(\vec{c}_1) \backslash \{c_r\}$. Moreover, we have that $N(\vec{c}\,'_1) = H(\vec{c}_1, \nu_1) \cup L(\vec{c}_1, \nu_1) \cup EL(\vec{c}_1, \nu_1) \setminus \{c_r\}$. We also know that $H(\vec{c_1}\,', \nu'_1) = H(\vec{c}_1, \nu_1) \setminus \{c_r\}$ since $r \in t_{\bullet_1}$ and c_r has been blocked. By this last fact and $R_3(\vec{c_1}, \nu_1)$, it holds $R_2(\vec{c_1}\,', \nu'_1)$ as expected. $R_2(\vec{c_2}, \nu_2)$ holds since it already holds by hypothesis. $R_3(\vec{c_1}\,', \nu'_1)$ holds since $R_3(\vec{c_1}, \nu_1)$ holds and $H(\vec{c_1}\,', \nu'_1) = H(\vec{c}_1, \nu_1) \backslash \{c_r\}$. $R_3(\vec{c_2}, \nu_2)$ holds since it already holds by hypothesis. $R_4(\vec{c}\,'_1, \nu'_1, \vec{c}_2, \nu_2)$ holds since low threads are not affected by the transition α_1. By taking $p' = p$, we have that $R_5(\vec{c}\,'_1, w'_1, \nu'_1, \vec{c_2}, w_2, \nu_2, p')$ holds since $R_5(\vec{c}_1, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ holds and because the eventually low thread, if it exists, has made no progress. $R_6(\vec{c_1}\,', \nu'_1)$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and low threads have made no progress. $R_7(\vec{c}\,'_1, \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c}_1, \nu_1, \vec{c_2}, \nu_2)$ holds and $H(\vec{c}\,'_1, \nu'_1) = H(\vec{c}_1, \nu_1) \setminus \{c_r\}$. Proposition $R_8(\vec{c_1}\,', \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low thread, if exists one, has made no progress. Proposition $R_9(\sigma'_1, \nu'_1, \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. By inspecting the semantics of threadpools, we have that $N(w'_1) = N(w_1) \cup \{c_r\}$. By rewriting $N(w)$ according to $R_{10}(w_1, \nu_1)$, we know that $N(w'_1) = BL(w_1, \nu_1) \cup BH(w_1, \nu_1) \cup BEL(w_1, \nu_1) \cup \{c_r\}$. Since $r \in t_{\bullet_1}$, we also know that $BH(w'_1, \nu'_1) = BH(w_1, \nu_1) \cup \{c_r\}$. Consequently, $R_{10}(w'_1, \nu'_1)$ holds. Proposition $R_{10}(w_2, \nu_2)$ holds since it already holds by hypothesis. Proposition $R_{11}(w'_1, \nu'_1)$ holds since $R_{11}(w_1,\ \nu_1)$ holds and because $BH(w'_1, \nu'_1) = BH(w_1, \nu_1) \cup \{c_r\}$. Proposition $R_{11}(\ w_2, \nu_2)$ holds since it holds by hypothesis. Propositions $R_{12}(w'_1,\ \nu'_1, w_2, \nu_2)$ and $R_{13}(w'_1, \nu'_1)$ hold since $R_{12}(w_1, \nu_1, w_2, \nu_2)$ and $R_{13}(w_1, \nu_1)$ hold and because no blocked low threads are affected by the transition α_1. By $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_{14}(w_1, \nu_1, w_2, \nu_2)$, Lemma 4 applied to c_r, and the fact that $BH(w'_1, \nu'_1) = BH(w_1, \nu_1) \cup \{c_r\}$, we obtain that $R_{14}(w'_1, \nu'_1, w_2, \nu_2)$ holds. Proposition $R_{15}(w'_1, \nu'_1, w_2, \nu_2)$ and $R_{16}(w'_1, w_2)$ hold since $R_{15}(w_1, \nu_1, w_2,\ \nu_2)$ and $R_{16}(w_1, w_2)$ hold and because no low semaphores or the eventually low thread, if exists one, are affected by the transition α_1. By inspecting the semantics, $c_r \in H(\vec{c_1}, \nu_1)$, and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, we have that $\Gamma(sem) = high$ and $r \in t_{\bullet_1}$. By these last facts and $R_{17}(w_1, \nu_1)$, we obtain that $R_{17}(w'_1, \nu'_1)$ holds. $R_{17}(w_2, \nu_2)$ holds since it already holds by hypothesis.

$c_r \in EL(\vec{c_1}, \nu_1)$)

Propositions R_{1-4} can be proved in a similar way as when $c_r \in H(\vec{c_1}, \nu_1)$. By hypothesis, we know that $BEL(w_1, \nu_1), EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$,

$BEL(w_2, \nu_2)$. By inspecting the semantics, we also know that $t_{e_1} = \{c_r\}$. By inspecting Definition 9, we have that $\emptyset, \{c_r\} \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, $BEL(w_2, \nu_2)$. We need to do case analysis to determine if $EL(\vec{c_2}, \nu_2) = \emptyset$ or $BEL(w_2, \nu_2) = \emptyset$. Both cases proceed in a similar way and therefore we omit when $BEL(w_2, \nu_2) = \emptyset$. Consequently, we have that $\emptyset, \{c_r\} \sim_{el,p,t_{e_1}} \emptyset, \{d_r\}$ where there exist commands c' and d' without `unhide` instructions such that $c_r \in \{c'; \texttt{unhide}, \texttt{unhide}\}$ and $d_r \in \{d'; \texttt{unhide}, \texttt{unhide}\}$ or $c_r \in \{c'; \texttt{unhide}; p, \texttt{unhide}; p\}$ and $d_r \in \{d'; \texttt{unhide}; p, \texttt{unhide}; p\}$. Since the triggered event is b^r_{sem}, we can deduce that $c_r \in \{c'; \texttt{unhide}\}$ or $c_r \in \{c'; \texttt{unhide}; p\}$. By inspecting the threadpool semantics, we have that $\langle\!| c_r, m_1 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c'_r, m'_1 |\!\rangle$, where $c'_r \neq \texttt{stop}$. Consequently, we know that $c'_r \in \{c''; \texttt{unhide}, \texttt{unhide}\}$ or $c'_r \in \{c''; \texttt{unhide}; p, \texttt{unhide}; p\}$ where $\langle\!| c', m_1 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c'', m'_1 |\!\rangle$. Let us take $p' = p$. Since $t'_{w_1} = t_{w_1} \cup \{r\}$ and $t_{e_1} = \{r\}$, we have that $BEL(w'_1, \nu'_1) = \{c'_r\}$, $EL(\vec{c_1}', \nu'_1) = \emptyset$, and that $\{c'_r\}, \emptyset \sim_{el,p',t_{e_1}} \emptyset, \{d_r\}$ holds. Therefore, $R_5(\vec{c_1}', w'_1, \nu'_1, \vec{c_2}, \nu_2, w_2, p')$ holds. Propositions $R_6(\vec{c_1}', \nu'_1)$ and $R_7(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$ hold since $R_6(\vec{c_1}, \nu_1)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold and because low and high threads have made no progress. Proposition $R_8(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and $EL(\vec{c_1}', \nu'_1) = \emptyset$. Propositions R_{9-13} are proved similarly as when $c_r \in H(\vec{c_1}, \nu_1)$. Proposition $R_{14}(w'_1, \nu'_1, w_2, \nu_2)$ holds since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ holds and because high threads have made no progress. By $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, applying Lemma 5 to c_r, and $R_{15}(w_1, \nu_1, w_2, \nu_2)$, we obtain that $R_{15}(w'_1, \nu'_1, w_2, \nu_2)$ holds. Proposition $R_{16}(w'_1, w_2)$ holds since $R_{16}(w_1, w_2)$ holds and because no low semaphores or the eventually low thread, if it exists, are affected by the transition. By inspecting the semantics, $c_r \in EL(\vec{c_1}, \nu_1)$, and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, we have that $\Gamma(sem) = high$ and $r \in t_{e_1}$. By these last facts and $R_{17}(w_1, \nu_1)$, we obtain that $R_{17}(w'_1, \nu'_1)$ holds. Proposition $R_{17}(w_2, \nu_2)$ holds since it already holds by hypothesis.

$\alpha_1 = b^r_{sem}\times$)

The proof proceeds similarly as when $\alpha_1 = b^r_{sem}$.

$\alpha_1 = u^r_{sem}$)

By inspecting the semantics of threadpools, this event can be produced by two rules in Figure 12. Which rule is applied depends on the existence of threads on the waiting list $w_1(sem)$ when executing `signal`, which is captured by the scheduler events u^r_r and u^r_a. The proof proceeds similarly in both cases. Therefore, we omit the case when the scheduler triggers the event u^r_r.

$c_r \in H(\vec{c_1}, \nu_1)$)

$R_1(m'_1, m_2)$ holds by inspecting the semantics of threadpools and applying Lemma 3 to c_r. By semantics, the thread c_a has been awakened and place into the threadpool. Since $R_{17}(w_1, \nu_1)$ holds and $\Gamma(sem) = high$ by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, we have that $a \in t_{\bullet_1} \cup t_{e_1}$ and consequently $R_2(\vec{c}\,'_1, \nu'_1)$ holds. Proposition $R_2(\vec{c_2}, \nu_2)$ holds since it holds by hypothesis. Proposition $R_3(\vec{c}\,'_1, \nu'_1)$ holds by $R_3(\vec{c_1}, \nu_1)$ and $R_{11}(w_1, \nu_1)$. Proposition $R_3(\vec{c_2}, \nu_2)$ holds since it holds already by hypothesis. Propositions R_{4-6} can be proved in a similar way as when $\alpha_1 = b^r_{sem}$. To prove proposition $R_7(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$, we need to consider if $a \in t_{\bullet_1}$. If that is the case, it is proved by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$

and $R_{14}(w_1, \nu_1, w_2, \nu_2)$. Otherwise, it holds since it already holds by hypothesis. To prove proposition $R_8(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, we need to consider if $a \in t_{e_1}$. If that is the case, it is proved by $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$. Otherwise, it holds since it already holds by hypothesis. By inspecting the semantics, we know that $N(w_1') = N(w_1) \setminus \{c_a\}$ and that, depending if $a \in t_{\bullet_1}$ or $a \in t_{e_1}$, we have that $BH(w_1', \nu_1') = BH(w_1, \nu_1) \setminus \{c_a\}$ or $BEL(w_1', \nu_1') = BEL(w_1, \nu_1) \setminus \{c_a\}$, respectively. Consequently, we obtain that $R_{10}(w_1', \nu_1')$ holds. Proposition $R_{10}(w_2, \nu_2)$ since it already holds by hypothesis. Proposition $R_{11}(w_1', \nu_1')$ holds since $R_{11}(w_1, \nu_1)$ holds and c_a has been move from one subpool of threads to another. Proposition $R_{12}(w_2, \nu_2)$ holds since it already holds by hypothesis. Proposition $R_{13}(w_1', \nu_1')$ holds since $R_{13}(w_1, \nu_1)$ holds and no threads have been blocked by the transition α_1. Propositions $R_{14}(w_1', \nu_1', w_2, \nu_2)$ and $R_{15}(w_1', \nu_1', w_2, \nu_2)$ hold since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$ hold and because c_a has been removed from the subpool of threads $BH(w_1', \nu_1')$ or $BEL(w_1', \nu_1')$ by the transition α_1. Since $R_{17}(w_1, \nu_1)$ holds and $\Gamma(sem) = high$ by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, it holds that $w_1' =_L w_1$ and thus $R_{16}(w_1', w_2)$ holds. Proposition $R_{17}(w_1', \nu_1')$ holds since $R_{17}(w_1, \nu_1)$ holds and c_a has been removed from the waiting list of sem. Proposition $R_{17}(w_2, \nu_2)$ holds since it already holds by hypothesis.

$c_r \in EL(\vec{c_1}, \nu_1)$)

The proof of R_{1-4} proceeds as when $c_r \in H(\vec{c_1}, \nu_1)$. Proposition R_5 is proved as when $\alpha_1 = b^r_{sem}$ and $c_r \in EL(\vec{c_1}, \nu_1)$. The rest of the propositions are proved similarly as when $c_r \in H(\vec{c_1}, \nu_1)$.

$\alpha_1 = u^r_{sem}\times$)

The proof proceeds similarly as when $\alpha_1 = u^r_{sem}$.

ii) In this case, all the propositions are valid since they are valid already by hypothesis. Observe that no step in the semantics is performed.

iii) Low events related to synchronization : b^r_{sem}, $b^r_{sem}\times$, u^r_{sem}, and $u^r_{sem}\times$ (where $r \in L(\vec{c_1}, \nu_1)$ and $r_e \in EL(\vec{c_1}, \nu_1)$).

$\alpha_1 = b^r_{sem}$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \texttt{wait}(sem); c'$ for some command c'. By inspecting the semantics for threadpools and commands, we have the transition $\langle c_r, m_1 \rangle \xrightarrow{b(sem)} \langle c_r', m_1 \rangle$, and that $\langle \sigma_1, \nu_1 \rangle \xrightarrow{b^r} \langle \sigma_1', \nu_1' \rangle$. Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \xrightarrow{b^{sem}} \langle \sigma_2', \nu_2' \rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle c_r, m_2 \rangle \xrightarrow{b(sem)} \langle c_r', m_2 \rangle$. We can therefore conclude that $\langle \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 \rangle \xrightarrow{b^r_{sem}} \langle \vec{c_2}', m_2', \sigma_2', \nu_2', w_2' \rangle$. $R_1(m_1', m_2')$ holds by applying Lemma 6 to c_r. Propositions $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since propositions $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_{10}(w_1, \nu_1)$ and $R_{10}(w_2, \nu_2)$ hold and since $L(\vec{c}'_1, \nu_1') = L(\vec{c}_1, \nu_1) \setminus \{c_r'\}$ by inspecting the semantics for threadpools. Proposition $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 block the same low thread on both configurations.

By taking $p' = p$, we have that $R_5(\vec{c_1}', w_1', \nu_1', \vec{c_2}', w_2', \nu_2', p)$ holds since proposition $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, \nu_2, w_2, p)$ holds and because the eventually low thread, if exists one, has made no progress. $R_6(\vec{c_1}', \nu_1')$ holds by $R_6(\vec{c_1}, \nu_1)$ and the fact that $c_r' \notin L(\vec{c_1}', \nu_1')$. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds because $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds because $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the transition α_1. Proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. By inspecting the semantics of threadpools, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, and $R_{12}(w_1, \nu_1, w_2, \nu_2)$, we have that $N(w_1') = N(w_2') = N(w_1) \cup \{c_r'\}$, $N(\vec{c_1}') = N(\vec{c_2}') = N(\vec{c_1}) \backslash \{c_r\}$, and $BL(w_1', \nu_1') = BL(w_2', \nu_2') = BL(w_1, \nu_2) \cup \{c_r'\}$. By these last facts, $R_{10}(w_1, \nu_2)$, $R_{10}(w_2, \nu_2)$, $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $R_{12}(w_1, \nu_1, w_2, \nu_2)$, we have that it holds $R_{10}(w_1', \nu_2')$, $R_{10}(w_2', \nu_2')$, $R_{11}(w_1', \nu_1')$, $R_{11}(w_2', \nu_2')$, $R_{12}(w_1', \nu_1', w_2', \nu_2')$. $R_{13}(w_1', \nu_1')$ holds since $R_{13}(w_1, \nu_1)$ holds and by Lemma 7. Proposition $R_{14}(w_1', \nu_1', w_2', \nu_2')$ and $R_{15}(w_1', \nu_1', w_2', \nu_2')$ holds since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$ holds and because transition α_1 does not affect high and eventually low threads, if they exists. Since $c_r = wait(sem); c'$, $c_r \in L(\vec{c_1}, \nu_1)$, $R_6(\vec{c_1}, \nu_1)$, and typing rules, we have that $\Gamma(sem) = low$. By this fact and $R_{16}(w_1, w_2)$, By inspecting the semantics, $c_r \in L(\vec{c_1}, \nu_1)$, and $R_6(\vec{c_1}, \nu_1)$, we have that $\Gamma(sem) = low$, $r \in t_{\circ_1} = t_{\circ_2}$. By these last facts, $R_{17}(w_1, \nu_1)$, and $R_{17}(w_2, \nu_2)$, we obtain that $R_{17}(w_2', \nu_2')$ and $R_{17}(w_2', \nu_2')$ hold.

$\alpha_1 = b^r_{sem}\times$) It proceeds in a similar way as when $\alpha_1 = b^r_{sem}$.

$\alpha_1 = u^r_{sem}$)

By inspecting the semantics of threadpools, this event can be produced by two rules in Figure 12. Which rule is applied depends on the waiting list $w(sem)$ when executing `signal`, which is captured by the scheduler events u^r_r and u^r_a. The proof proceeds similarly in both cases. Therefore, we omit the case when the scheduler triggers the event u^r_r.

By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \texttt{signal}(sem); c'$ for some command c'. By inspecting the semantics for threadpools and commands, we have the transition $\langle\!| c_r, m_1 |\!\rangle \overset{u(sem)}{\rightharpoonup} \langle\!| c_r', m_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \overset{u^r_a}{\rightarrow} \langle\!| \sigma_1', \nu_1' |\!\rangle$ for some a. Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \overset{u^r_a}{\rightarrow} \langle\!| \sigma_2', \nu_2' |\!\rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle\!| c_r, m_2 |\!\rangle \overset{u(sem)}{\rightharpoonup} \langle\!| c_r', m_2 |\!\rangle$. We can therefore conclude that $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\rangle \overset{u^r_{sem}}{\longrightarrow} \langle\!| \vec{c_2}', m_2', \sigma_2', \nu_2', w_2' |\!\rangle$. Proposition $R_1(m_1', m_2')$ holds by applying Lemma 6 to c_r. By $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and inspecting the semantics of threadpools and the scheduler, we have that $L(\vec{c_1}', \nu_1') = L(\vec{c}_2', \nu_2') = L(\vec{c}_1, \nu_1) \cup \{c_a\}$. Therefore, we have that $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds. Moreover, $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $\Gamma(sem) = low$ by typing rules, and $R_{17}(w_1, \nu_1)$ hold. By taking $p' = p$, we have that $R_5(\vec{c_1}', w_1', \nu_1', \vec{c_2}', w_2', \nu_2', p)$ holds since proposition $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, \nu_2, w_2, p)$ holds and because the eventually low thread, if exists one, has made no progress. Proposition $R_6(\vec{c_1}', \nu_1')$ holds by $R_6(\vec{c_1}, \nu_1)$, inspecting the semantics of threadpools, and $R_{13}(\vec{c_1}, \nu_1)$. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds because $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds because $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the transition α_1. Proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. Since $r \in t_{\circ_1}$ and $R_6(\vec{c_1}, \nu_1)$, we obtain that $\Gamma(sem) = low$. By these facts, $R_{12}(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_{16}(w_1, w_2)$, and $R_{17}(w_1, \nu_1)$, we can conclude that $BL(\vec{c_1}', \nu_1') = BL(\vec{c_2}', \nu_2') = BL(\vec{c}_1, \nu_1) \setminus \{c_a\}$. By applying this fact together with $R_{10}(w_1, \nu_1)$ and $R_{10}(w_2, \nu_2)$, we obtain that $R_{10}(w_1', \nu_1')$ and $R_{10}(w_2', \nu_2')$ hold. Propositions $R_{11}(w_1', \nu_1')$, $R_{11}(w_2', \nu_2')$, and $R_{12}(w_1', \nu_1', w_2', \nu_2')$ hold since $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, and $R_{12}(w_1, \nu_1, w_2, \nu_2)$ hold and since $BL(\vec{c_1}', \nu_1') = BL(\vec{c_2}', \nu_2') = BL(\vec{c}_1, \nu_1) \setminus \{c_a\}$ by inspecting the semantics of threadpools. $R_{13}(w_1', \nu_1')$ holds since $R_{13}(w_1, \nu_1)$ holds an a low thread has been awakened. $R_{14}(w_1', \nu_1', w_2', \nu_2')$ and $R_{15}(w_1', \nu_1', w_2', \nu_2')$ holds since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$ holds and because transition α_1 does not affect high and eventually low threads, if they exists. $R_{16}(w_1', w_2')$ holds since $R_{16}(w_1, w_2)$ holds and since $w_1'(sem) = w_1(sem) \setminus \{c_a\}$ by inspecting the semantics of threadpools. $R_{17}(w_1', \nu_1')$ and $R_{17}(w_2', \nu_2')$ hold since $R_{17}(w_1, \nu_1)$ and $R_{17}(w_2, \nu_2)$ hold and because $t'_{w_1} = t_{w_1} \setminus \{a\}$ and $t'_{w_2} = t_{w_2} \setminus \{a\}$ by semantics of the scheduler.

$\alpha_1 = u^r_{sem}\times$) It proceeds in a similar way as when $\alpha_1 = u^r_{sem}$.

$\alpha_1 = \bullet \rightsquigarrow r_e$) We know that $r_e \in t_{e_1}$. By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_{r_e} = \texttt{unhide}; c^*$ or $c_{r_e} = \texttt{unhide}$ for some command c^*, $c_{r_e} \in EL(\vec{c_1}, \nu_1)$, $\langle\!| c_{r_e}, m_1 |\!\rangle \overset{\bullet\rightsquigarrow}{\rightharpoonup} \langle\!| c^*, m_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoondown} \langle\!| \sigma_1', \nu_1' |\!\rangle$. We are only going to consider the case when $c_{r_e} = \texttt{unhide}; c^*$ since the proof for $c_{r_e} = \texttt{unhide}$ is analogous. Therefore, we omit the proof when $\alpha_1 = \bullet \rightsquigarrow r_e \times$.

Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoondown} \langle\!| \sigma_2', \nu_2' |\!\rangle$. Because $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ holds, we know that $c^* = p$ and that the thread with name $r_e \in L(\vec{c_2}, \nu_2)$ or $r_e \in BL(w_2, \nu_2)$. Since $ii)$ cannot be applied, we obtain that $r_e \in L(\vec{c_2}, \nu_2)$. Let us call $c^2_{r_e}$ the thread with name r_e in $\vec{c_2}$. Since $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ holds and it is not possible for a thread to make progress by a high computation, we have that $c^2_{r_e} = \texttt{unhide}; p$. As a consequence of that, it holds that $\langle\!| c^2_{r_e}, m_2 |\!\rangle \overset{\bullet\rightsquigarrow}{\rightharpoonup} \langle\!| p, m_2 |\!\rangle$. Therefore, the transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoonup} \langle\!| \vec{c_2}', m_2', \sigma_2', \nu_2', w_2' |\!\rangle$ holds.

$R_1(m_1', m_2')$ holds trivially since unhide has no changed the memories. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds; and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because after the transition α_1, the threads c_{r_e} and $c^2_{r_e}$ become the thread p. By inspecting the semantics for the scheduler, we have that $EL(\vec{c_1'}, \nu_1') = EL(\vec{c_2'}, \nu_2') = \emptyset$. By $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ and since we cannot apply $ii)$, we also know that $BEL(w_1, \nu_1) = BEL(w_2, \nu_2) = \emptyset$ and consequently $BEL(w_1', \nu_1') = BEL(w_2', \nu_2') = \emptyset$ since no thread is blocked by the transition α_1. Then, by taking $p' = \texttt{skip}$, it trivially holds that $R_5(\vec{c_1}', w_1', \nu_1', \vec{c_2}', w_2', \nu_2', \texttt{skip})$. $R_6(\vec{c_1}', \nu_1')$ holds since $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2},$

$w_2, \nu_2, p)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds ; and by inspecting the type derivation of c_{r_e}. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $EL(\vec{c_1}', \nu_1') = EL(\vec{c_2}', \nu_2') = \emptyset$. Finally, $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. Propositions R_{10-17} hold since they hold already by hypothesis and since transition α_1 does not affect blocked threads.

□

Corollary 2 (Soundness). If $\Gamma_{hc}, hc \mapsto low \vdash c : low$ then c is secure.

Proof. For arbitrary σ, m_1, and m_2 so that $m_1 =_L m_2$ and σ is noninterferent, assume $\langle\!| c, m_1, \sigma, \nu_{init} |\!\rangle \Downarrow cfg_1$ & $\langle\!| c, m_2, \sigma, \nu_{init} |\!\rangle \Downarrow cfg_2$. Observe that, by assuming terminating configurations, it is not possible to apply case $ii)$ of Theorem 2.
By inductive (in the number of transition steps of the above configurations) application of Theorem 2, we propagate invariant $m_1 =_L m_2$ to the terminating configurations. □