Securing Interaction between Threads and the Scheduler

Alejandro Russo 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, scheduler dependence, and efficiency overhead for 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 [25]. 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 [30]. In this light, it is hardly surprising that Jif [18] and Flow Caml [27], 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 [31].

There are 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 [26]. 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.

- *Standard semantics* Following the philosophy of *extensional security* [17], we argue for security defined in terms of standard semantics, as opposed to security-instrumented semantics. If there are some non-standard 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 publically-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.

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), and implementation issues (Section 8) before we conclude the paper (Section 9).

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* [4, 9, 33]) 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* [5] 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:

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

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

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

The lock primitives ensure that the undesired interleaving of c_1 and c_2 is prevented.

Unfortunately, synchronization primitives offer no general solution. The source of the leak in program $d_1 \parallel d_2$ is *internal timing* [32]. 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 [25] for other, less related, work.

2.2. Possibilistic security

Smith and Volpano [30] 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 [16]. Work by Honda et al. [11, 12] and Pottier [19] 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 [8] for an overview) and CSP (see [22] for an overview).

2.3. Scheduler-specific security

Volpano and Smith [32] 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 live 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 $\mathtt{protect}(c)$ is executed atomically, *by definition* of its semantics. Such a primitive can be used to secure program $d_1 \parallel d_2$ as:

$$\begin{array}{ll} d_1: & \mathtt{protect}(\mathtt{if}\ h > 0\ \mathtt{then}\ \mathtt{sleep}(100)\ \mathtt{else}\ \mathtt{skip}); \\ & l := 1 \\ d_2: & \mathtt{sleep}(50);\ l := 0 \end{array}$$

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. A synchronization-based implementation would face some non-trivial 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:

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

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 $\mathtt{sleep}(\mathtt{n})$ 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 $\mathtt{sleep}(100)$, 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 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 [28] 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 [2, 3] 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 [29] 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:

$$\begin{array}{l} e_1 : l := 0 \\ e_2 : l := 1 \\ e_3 : \mathtt{if}\ h > 0\ \mathtt{then}\ \mathtt{fork}(\mathtt{skip}, \mathtt{skip})\ \mathtt{else}\ \mathtt{skip} \end{array}$$

This program is considered secure according to [29]. 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 [14, 15].

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 [26] 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 [1, 26] might arbitrarily change the efficiency of the resulting code (and possibly result in a diverging program). In the present work, we assume a weaker attacker and aim for a more permissive security condition and analysis.

External timing-sensitive security has been extended to languages with semaphores [23] and message passing [24].

2.5. Security via low determinism

Inspired by Roscoe's *low-view determinism* [20] for security in a CSP setting, Zdancewic and Myers [34] 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. Analyses enforcing low-view determinism are inherently non-compositional because the parallel composition with a thread assigning to low variables is not generally secure.

Most recently, Huisman et al. [13] have suggested a temporal logic-based characterization of low-view determinism security. This characterization enables high-precision security enforcement by known model-checking techniques.

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).

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

Figure 1. Command syntax

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 form $\langle\!|c, m|\!\rangle \xrightarrow{\alpha} \langle\!|c', m'|\!\rangle$ 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 $\rightsquigarrow\!\bullet$ which communicates to the scheduler to treat the thread as 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. 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, an `unhide` in an exception handler may refer to several `hide` primitives under a `try` statement.

Commands $\texttt{fork}(c, \vec{d})$ and $\texttt{hfork}(c, \vec{d})$ dynamically spawn a collection of threads $\vec{d}$ 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, generally, a pro-

$$\langle \mathtt{skip}, m \rangle \rightharpoonup \langle \mathtt{stop}, m \rangle$$

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

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

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

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

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

$$\frac{\langle e, m \rangle \downarrow \mathtt{True}}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m \rangle \rightharpoonup \langle c; \mathtt{while}\ e\ \mathtt{do}\ c, m \rangle}$$

$$\frac{\langle e, m \rangle \downarrow \mathtt{False}}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m \rangle \rightharpoonup \langle \mathtt{stop}, m \rangle}$$

$$\langle \mathtt{hide}, m \rangle \xrightharpoonup{\rightsquigarrow\bullet} \langle \mathtt{stop}, m \rangle$$

$$\langle \mathtt{unhide}, m \rangle \xrightharpoonup{\bullet\rightsquigarrow} \langle \mathtt{stop}, m \rangle$$

$$\langle \mathtt{fork}(c, \vec{d}), m \rangle \xrightharpoonup{\circ_{\vec{d}}} \langle c, m \rangle$$

$$\langle \mathtt{hfork}(c, \vec{d}), m \rangle \xrightharpoonup{\bullet_{\vec{d}}} \langle c, m \rangle$$

Figure 2. Semantics for commands

gram σ that forms a *scheduler configuration* $\langle \sigma, \nu \rangle$ together with a memory ν. 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 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 have 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. For simplicity, we require that scheduler transitions are deterministic. 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 it satisfies these basic assumptions and runs infinitely long. We discuss an example of a scheduler that conforms to these assumptions in Section 4.

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`. Rule for event $r \rightsquigarrow$ keeps track of a non-terminal 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.

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 $\langle \vec{c}, m, \sigma, \nu \rangle \xrightarrow{\alpha} \langle \vec{c'}, m', \sigma', \nu' \rangle$ where α ranges over the same set of events as in the semantics for schedulers.

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

$$\frac{q > 0 \qquad q' = q - 1}{\langle\!|\sigma, \nu|\!\rangle \overset{r\leadsto}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle} \qquad \frac{q > 0 \qquad q' = 0 \qquad \forall\alpha \in \{\bullet, \circ\}.t'_\alpha = t_\alpha \backslash \{r\}}{\langle\!|\sigma, \nu|\!\rangle \overset{r\leadsto\times}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle}$$

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

$$\frac{q > 0 \qquad q' = q - 1 \qquad s' = \bullet \qquad t'_\circ = t_\circ \backslash \{r\} \qquad t'_e = \{r\}}{\langle\!|\sigma, \nu|\!\rangle \overset{r\leadsto\bullet}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle}$$

$$\frac{q > 0 \qquad q' = 0 \qquad s' = \circ \qquad t'_\circ = t_\circ \cup \{r\} \qquad t'_e = \emptyset}{\langle\!|\sigma, \nu|\!\rangle \overset{\bullet\leadsto r}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle}$$

$$\frac{q > 0 \qquad q' = 0 \qquad s' = \bullet \qquad \forall\alpha \in \{\bullet, \circ\}.t'_\alpha = t_\alpha \backslash \{r\} \qquad t'_e = \emptyset}{\langle\!|\sigma, \nu|\!\rangle \overset{r\leadsto\bullet\times}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle}$$

$$\frac{q > 0 \qquad q' = 0 \qquad s' = \circ \qquad \forall\alpha \in \{\bullet, \circ\}.t'_\alpha = t_\alpha \backslash \{r\} \qquad t'_e = \emptyset}{\langle\!|\sigma, \nu|\!\rangle \overset{\bullet\leadsto r\times}{\rightharpoondown} \langle\!|\sigma', \nu'|\!\rangle}$$

Figure 3. 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 $r\leadsto$. If c_r terminates in a transition without labels, then scheduler event $r\leadsto\times$ is required for synchronization in order to update the threadpool information in the scheduler memory. If c_r terminates with $\leadsto\bullet$ (resp., $\bullet\leadsto$) then synchronization with $r\leadsto\bullet\times$ (resp., $\bullet\leadsto r\times$) 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 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 accommodate 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 $high \not\sqsubseteq low$.

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 need 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.

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

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

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

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

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

$$\frac{\langle \sigma, \nu\rangle \overset{\uparrow_\alpha r'}{\rightharpoonup} \langle \sigma', \nu'\rangle}{\langle c_1 \dots c_n, m, \sigma, \nu\rangle \overset{\uparrow_\alpha r'}{\rightharpoonup} \langle c_1 \dots c_n, m, \sigma', \nu'\rangle}\ \alpha \in \{\circ, \bullet\}, r' \in \{1, \dots, n\}$$

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

Figure 4. Semantics for threadpools

We will illustrate 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 $m_1|_L = m_2|_L$. 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 $\circ^r_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\circ r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r and r' are low threads) operate on low threads and are therefore low events. On the other hand, events $\bullet^r_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, and $\uparrow_\bullet r'$ (where r and r' are high threads) are high.

With security partition defined on scheduler events, we specify the indistinguishability of scheduler configurations via *low-bisimulation*.

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 $\langle\sigma_i, \nu_i\rangle \overset{\alpha}{\rightharpoonup} \langle\sigma'_i, \nu'_i\rangle$ 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 $\langle\sigma_i, \nu_i\rangle \overset{\alpha}{\rightharpoonup} \langle\sigma'_i, \nu'_i\rangle$ where α is low and $i \in \{1, 2\}$, then $\langle\sigma_{3-i}, \nu_{3-i}\rangle \overset{\alpha}{\rightharpoonup} \langle\sigma'_{3-i}, \nu'_{3-i}\rangle$ and $\langle\sigma'_i, \nu'_i\rangle\ R\ \langle\sigma'_{3-i}, \nu'_{3-i}\rangle$.*

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:

```
t∘ := [c]; t• := []; r := c; s := 0; turn := 0;
while (True) do {
 q := M; run(r);
 while (q > 0) do {
  receive
   ∘^r_d⃗ :   t∘ := append(t∘, N(d⃗));
   •^r_d⃗ :   t• := append(t•, N(d⃗));
   r⇝ :     skip;
   r⇝× :    t∘ := remove(r, t∘); t• := remove(r, t•);
            q := 0;
   r⇝• :    t∘ := remove(r, t∘); t• := remove(r, t•);
            t• := append(t•, [r]); s := 1;
   •⇝r :    t∘ := append(t∘, [r]);
            t• := remove(r, t•); s := 0; q := 0;
   r⇝•× :   t∘ := remove(r, t∘); t• := remove(r, t•);
            s := 1; q := 0;
   •⇝r× :   t∘ := remove(r, t∘); t• := remove(r, t•);
            s := 0; q := 0;
  end receive;
  q := q − 1
 };
 turn := (turn + 1)%2;
 if ((turn = 1) or (s = 1))
 then {r := head(t•); t• := append(tail(t•), [r])}
 else {r := head(t∘); t∘ := append(tail(t∘), [r])}
}
```

Figure 5. Round-robin scheduler

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

Figure 5 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 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* [25] 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.

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 systems

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 6 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 [5, 33]. 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

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

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

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

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

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

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

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

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

Figure 6. Security type system

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 track 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 6, we obtain a standard type system for securing information flow in sequential programs (cf. [33]). 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. We only give proof sketches due to the lack of space. Extended proofs are available in the full version of the paper.

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* $\Gamma(hc) = high$.

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

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

Lemma 2 *If* $\Gamma_{hc}, hc \mapsto low \vdash \texttt{hide}; c : low$, *then there exist commands* c' *and* p *such that* $c = c'; \texttt{unhide}; p$, *where* c' *has no* `unhide` *commands, or* $c = \texttt{unhide}; p$.

Proof. By induction on the size of command c. $\square$

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 *We define the relation* eventually low, *written $\sim_{el,p}$, on empty or singleton sets of threads as follows:*

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

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* [10] 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 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 $R_4(\vec{c_1},\nu_1,\vec{c_2},\nu_2)$, $EL(\vec{c_1},\nu_1) \sim_{el,p} EL(\vec{c_2},\nu_2)$, 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:*

- *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 $R_9(\sigma'_i,\nu'_i,\sigma_{3-i},\nu_{3-i})$;*
- *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. The interesting cases are `fork` and `hfork`, where the dynamic creation of threads is handled; and `hide` and `unhide`, where the notion of eventually low is used. These cases are elaborated in the full version of the paper. Briefly, the *hc* mechanism of the type system together with the low-bisimilarity $\langle\sigma_1,\nu_1\rangle \sim_L \langle\sigma_2,\nu_2\rangle$ allows us to propagate the invariant according to the high and low cases. $\square$

Corollary 1 (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 $\langle c, m_1, \sigma, \nu_{init}\rangle \Downarrow cfg_1$ & $\langle c, m_2, \sigma, \nu_{init}\rangle \Downarrow cfg_2$. 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. $\square$

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 are as follows:

$$\langle \texttt{yield}, m\rangle \xrightarrow{\not\rightarrow} \langle \texttt{stop}, m\rangle$$

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

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

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

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 to enforcing security under cooperative scheduling [21]. 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 [21] 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 7. 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 (including the class and seat details), p_h contains secret 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_h, which is secret. Variable s_h is assigned the (secret) status (e.g., $BASIC$ or $GOLD$) of passenger p_h. The value of s_h is then stored in o_h.

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

Figure 7. Ticket purchase code

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 assigns the time stamp of the purchase to (public) variable d_l and 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 (d_l) after the instructions that branches on secrets. Nevertheless, a minor modification of the program (which can, generally, 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. Implementation issues

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 [7] and the NPTL [6] 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. We have modified this library to allow the implementation of the primitives `hide` and `unhide` as well as a noninterferent scheduler based on the round-robin policy from Section 4. The scheduler consists of one list of threads for each security level, in this case, low and high. The 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).

The NPTL library, on the other hand, is more complex than the previous one. It maps threads in user space to threads in kernel space by using low-level primitives in the code. Nevertheless, it is possible to apply a similar procedure to that we have applied to 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. 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 [14, 15].

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.

Standard semantics The underlying semantics does not appeal to the nonstandard `protect` construct. The semantics, however, feature 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. Input/output and synchronization are also desirable features. We expect a natural extension of our model with input/output primitives on channels labeled with security levels, as well as synchronization primitives (such as 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. Low semaphores s would 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. High semaphores would 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. Our future work plans include adapting the type system to unstructured languages (such as languages with exceptions and bytecode) and facilitating tool support for it.

Acknowledgments

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

References

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

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

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

[4] 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.

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

[6] U. Drepper and I. Molnar. The native posix thread library for linux. `http://people.redhat.com/drepper/nptl-design.pdf`, Jan. 2003.

[7] R. S. Engelschall. Gnu pth - the gnu portable threads. `http://www.gnu.org/software/pth/`, Nov. 2005.

[8] 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.

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

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

[11] 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.

[12] 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, Jan. 2002.

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

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

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

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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, Sept. 2002.

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

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

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

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

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

[30] 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, Jan. 1998.

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

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

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

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