Compositional Non-Interference for Concurrent Programs via Separation and Framing

Aleksandr Karbyshev[1], Kasper Svendsen[2], Aslan Askarov[1], and Lars Birkedal[1]

[1] Aarhus University, Denmark
[2] University of Cambridge, UK

Abstract. Reasoning about information flow in a concurrent setting is notoriously difficult due in part to timing channels that may leak sensitive information. In this paper, we present a compositional and flexible type-and-effect system that guarantees non-interference by disallowing potentially insecure races that can be exploited through internal timing attacks. In contrast to many previous approaches, which disallow all races on public variables, we use an explicit scheduler model to give a more permissive security definition and type system, which allows benign races on public variables. To achieve compositionality, we use the idea of resources from separation logic, both to locally specify and reason about whether accesses may be racy and to bound the security level of data that may be learned through scheduling.

1 Introduction

Non-interference [15] is an important security property. Informally, a program satisfies non-interference if its publicly observable (*low*) outputs are independent of its private (*high*) inputs. In spite of the vast body of research on non-interference, reasoning about information flow control and enforcing non-interference for imperative concurrent programs remains a difficult problem. One of the main problems is prevention of information flows that originate from interaction of the scheduler with individual threads, also known as *internal timing leaks*.

Example 1. Consider the following program [44][3].

$$\begin{array}{ll} \mathsf{fork}(\mathsf{delay}(50);\ l := 1); & \textit{// Thread 1} \\ \mathsf{fork}(\mathsf{if}\ h\ \mathsf{then\ skip\ else\ delay}(100);\ l := 2); & \textit{// Thread 2} \end{array}$$

In this program, h is a high variable and l is intended to be a low variable. But the order of the two assignments to l depends on the branch that is picked by Thread 2. As a result, under many schedulers, the resulting value of $l = 1$ reveals the value of h being true to a low observer.

[3] $\mathsf{delay}(n)$ is used as an abbreviation for $\mathsf{skip}; \ldots; \mathsf{skip}$ n times, i.e., it models a computation that takes n reduction steps.

It may appear that the problem in the above example is that Thread 2 races to the low assignment after branching on a secret. The situation is actually worse. Without explicit assumptions on the scheduling of threads, a mere *presence* of a high branching in the pool of concurrently running threads is problematic.

Example 2. Consider the following program, which forks three threads.

```
fork(delay(50); l := 1);                          // Thread 1
fork(if h then skip else delay(100));             // Thread 2
fork(l := 2)                                      // Thread 3
```

In this program, every individual thread is secure, in the sense that it does not leak information about high variables to a low observer. Additionally, pairwise parallel composition of any of the threads is secure, too, including a benign race fork($l := 1$); fork($l := 2$). Even if we assume that the attacker fully controls the scheduler, the final value of l will be determined only by the scheduler of his choice. However, for the parallel execution of all the three threads, if the attacker can influence the scheduler, it can leak the secret value of h through public l.

In this paper, we present a compositional and flexible type-and-effect system that supports compositional reasoning about information flow in concurrent programs, with minimal assumptions on the scheduler. Our type system is based on ideas from separation logic; in particular, we track ownership of variables. An assignment to an exclusively-owned low variable is allowed as long as it does not create a thread-local information flow violation, regardless of the parallel context. Additionally, we introduce a notion of a *labeled scheduler resource*, which allows us to distinguish and accept benign races as secure.[4] A racy low assignment is allowed as long as the thread's scheduler resource is low; the latter, in its turn, prevents parallel composition of the assignment with high threads, avoiding potential scheduler leaks. This flexibility allows our type system to accept pairwise parallel compositions of threads from Example 2, while rightfully rejecting the composition of all three threads.

Following the idea of ownership transfer from separation logic, our type system allows static transfer of resource ownership along synchronization primitives. This enables typing of programs that use synchronization primitives to avoid races, as illustrated in the following example.

Example 3. Consider the following modification of Example 2.

```
fork(delay(50); l := 1; send(c));        // Thread 1
fork(if h then skip else delay(100));    // Thread 2
recv(c);                                 // recover exclusive ownership of variable l
fork(l := 2)                             // Thread 3
```

[4] One could argue that programs should not have any races on assignments at all; but in general we will want to allow races on some shareable resources (e.g., I/O) and that is why we study a setup in which we do try to accommodate benign races to assignments.

In this program, Thread 1 sends a message on channel c. Since the main program synchronizes on the c channel (by receiving on channel c), Thread 3 is not forked until after the assignment $l := 1$ in Thread 1 has happened. Hence, the synchronization ensures that there is no race on l and the program is therefore secure, even in the presence of the high branching in the concurrent Thread 2.

Note that unconstrained transfer of resources creates an additional covert channel that needs to be controlled. Section 3 describes how our type system prevents implicit flows via resource transfer.

One might expect that synchronization can also be used to allow races after high threads are removed from the scheduler. That is, however, problematic, as illustrated by the following example.

Example 4. Consider the following program.

```
fork(if h then s1 else s2; send(c));          // Thread 1
recv(c);
fork(l := 1);                                 // Thread 2
fork(l := 2)                                  // Thread 3
```

The program forks off three threads and uses send(c) and recv(c) on a channel c to postpone forking of Thread 2 and 3 until after Thread 1 has finished. Here it is possible for the high thread (Thread 1) to taint the scheduler and thus affect its choice of scheduling between Threads 2 and 3 after Thread 1 has finished. This could, e.g., happen if we have an adaptive scheduler and s_1 and s_2 have different workloads. Then the scheduler will be adapted differently depending on whether h is true or false and therefore the final value of l may reveal the value of h.

To remedy this issue, we introduce a special *rescheduling* operation that resets the scheduler state, effectively removing all possible taint from past high threads.

Example 5. Consider the following variation of Example 4:

```
fork(if h then s1 else s2; send(c));     // Thread 1
recv(c);
reschedule;                              // reset the scheduler state
fork(l := 1);                            // Thread 2
fork(l := 2)                             // Thread 3
```

The reschedule operation resets the scheduler state and therefore no information about the high variable h is leaked from the high thread and this program is thus secure.

The above example illustrates that reschedule allows us to remove scheduler taint from high threads and thus accept programs with benign races as secure after high threads have finished executing.

Contributions This paper proposes a new *compositional* model for enforcing information flow security in imperative concurrent programs. The key components of the model are:

– A fine-grained compositional[5] type-and-effect system that prevents internal timing leaks by tracking when races may occur and whether the scheduler state could be tainted with confidential information. The type-and-effect system allows us to verify programs with benign races as secure.
– A novel programming construct for resetting the scheduler state.
– A proof technique for termination-insensitive notion of security under possible low nondeterminism.

We emphasize that our model is independent of the choice of scheduler; the only restriction on the runtime system is that it should implement the reschedule operation for resetting the scheduler state. This is a very mild restriction. Compared to other earlier work that also allows for scheduler independence and benign low races, our type-and-effect system is, to the best of our knowledge, much more expressive in the sense that it allows to verify more programs as secure.

The choice of termination-insensitive security condition as the target condition is deliberate for we only consider batch-style executions. We believe that our results can be extended to progress-insensitive security [2] using standard techniques. Despite that termination-insensitive security conditions leak arbitrary information [3], these leaks occur only via unary encoding of the secret in the trace and are relatively slow, especially when the secret space is large, compared to fast internal timing channels that we aim to close. We do not consider termination (or progress)-sensitivity because it is generally difficult to close all possible termination and crashing channels that may be exploited by the adversary, including resource exhaustion, without appealing to system-level mechanisms that also mitigate external timing channels. We discuss this more in detail in Section 5. Finally, note that in this paper we only address leaks through interactions with the scheduler (i.e., the *internal* timing leaks). Preventing *external* leaks is an active area of research and is out of scope of the paper.

Outline The remainder of this paper is organized as follows. In Section 2, we formally define the concurrent language and our security model. In Section 3, we present the type system for establishing security of concurrent programs. For reasons of space, an overview of the soundness proof and the detailed proof can be found in the accompanying appendix. We discuss related work in Section 5. Finally, in Section 6, we conclude and discuss future work.

2 Language and Security Model

We begin by formally defining the syntax and operational semantics of a core concurrent imperative language. The syntax is defined by the grammar below and

[5] We use a standard notion of compositionality for separation-style type systems, see comments to Theorem 1.

includes the usual imperative constructs, loops, conditionals and fork. Thread synchronization is achieved using channels which support a non-blocking send primitive and a blocking receive. In addition, the syntax also includes our novel reschedule construct for resetting the scheduler.

$$\begin{array}{rl}
v \in \mathit{Val} ::= & () \mid n \mid \mathsf{tt} \mid \mathsf{ff} \\
e \in \mathit{Exp} ::= & x \mid v \mid e_1 = e_2 \mid e_1 + e_2 \\
s \in \mathit{Stm} ::= & \mathsf{skip} \mid s_1; s_2 \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{while}\ e\ \mathsf{do}\ s \\
\mid & \mathsf{fork}(s) \mid \mathsf{send}(ch) \mid \mathsf{recv}(ch) \mid \mathsf{reschedule} \\
K \in \mathit{ECtx} ::= & \bullet \mid K; s
\end{array}$$

Here x and ch range over finite and disjoint sets of variable and channel identifiers, respectively. The sets are denoted by Var and Chan, respectively.

The operational semantics is defined as a small-step reduction relation over configurations of the form $\mathit{sf}, S, T, M, \rho$ consisting of a scheduling function sf, a scheduler state S, a thread pool T, a message pool M and a heap ρ. A scheduling function sf takes a scheduler state, a thread pool, a message pool and a heap as arguments and returns a new scheduler state and a thread identifier of the next thread to be scheduled [33, 30]. A thread pool T is a partial function from thread identifiers to sequences of statements, a message pool is a function from channel names to natural numbers, each representing a number of signals available on the respective channel, and a heap is a function from variables to values. We model a thread as a stack of statements, pushing whenever we encounter a branch and popping upon termination of branches. The semantic domains are defined formally in Figure 1.

$$\begin{array}{ll}
T \in \mathit{TPool} \overset{\mathit{def}}{=} \mathit{TId} \overset{\mathit{fin}}{\rightharpoonup} \mathit{seq}\ \mathit{Stm} & \mathit{sf} \in \mathit{Schd} \overset{\mathit{def}}{=} \mathcal{S} \times \mathit{TPool} \times \mathit{MPool} \times \mathit{Heap} \to \mathcal{S} \times \mathit{TId} \\
M \in \mathit{MPool} \overset{\mathit{def}}{=} \mathit{Chan} \to \mathbb{N} & \Psi \in \mathit{ReSchd} \overset{\mathit{def}}{=} \mathit{Schd} \times \mathit{MPool} \times \mathit{Heap} \to \mathit{Schd} \times \mathcal{S} \times \mathit{TId} \\
\rho \in \mathit{Heap} \overset{\mathit{def}}{=} \mathit{Var} \to \mathit{Val} &
\end{array}$$

Fig. 1. Semantic domains.

$$\frac{T, M, \rho \rightarrowtriangle^{t,a} T', M', \rho' \qquad \mathit{sf}(S, T, M, \rho) = (S', t) \qquad a \neq \mathbf{rs}(\cdot)}{\mathit{sf}, S, T, M, \rho \longrightarrow_{\Psi} \mathit{sf}, S', T', M', \rho'}$$

$$\frac{T, M, \rho \rightarrowtriangle^{t,a} T', M', \rho' \qquad \Psi(\mathit{sf}, M, \rho) = (\mathit{sf}', S', t') \qquad a = \mathbf{rs}(t')}{\mathit{sf}, S, T, M, \rho \longrightarrow_{\Psi} \mathit{sf}', S', T', M', \rho'}$$

Fig. 2. Global reduction relation.

The reduction relation is split into a *local* reduction relation that reduces a given thread and a *global* reduction relation that picks the next thread to be scheduled. The global reduction relation is defined in terms of the local reduction relation, written $T, M, \rho \rightarrowtriangle^{t,a} T', M', \rho'$, which reduces the thread t in thread pool T, emitting action a during the reduction. The global reduction relation only distinguishes between reschedule actions and non-reschedule actions. To reduce reschedule actions, the global reduction relation refers to a rescheduling function Ψ, which computes the next scheduler and scheduler state. The global reduction relation, written $sf, S, T, M, \rho \longrightarrow_{\Psi} sf', S', T', M', \rho'$, is indexed by a rescheduling function Ψ, which takes as argument the current scheduling function, message pool and heap and returns a new scheduling function and scheduler state. The global reduction relation is defined formally in Figure 2.

The local reduction relation is defined over configurations consisting of a thread pool, a message pool and a heap (Figure 3). It is defined in terms of a statement reduction relation, $s, h \to_a s'$ that reduces a statement s to s' and emits an action a describing the behavior of the statement on the state. We use evaluation contexts, K, to refer to the primitive statement that appears in a reducible position inside a larger statement. We use $K[s]$ to denote the

$$\frac{T(t) = K[s] :: \mathit{stk} \qquad s, \rho \to_a s'}{T, M, \rho \twoheadrightarrow^{t,a} [\![a]\!]_A(T[t \mapsto K[s'] :: \mathit{stk}], M, \rho, t)} \qquad \frac{T(t) = \mathsf{skip} :: \mathit{stk}}{T, M, \rho \twoheadrightarrow^{t,\epsilon} T[t \mapsto \mathit{stk}], M, \rho}$$

Fig. 3. Local reduction relation.

substitution of statement s in evaluation context K. Actions include a no-op action, ϵ, a branch action, $\mathbf{b}(e, s)$, an assignment action, $\mathbf{a}(x, v)$, a fork action, $\mathbf{f}(s)$, send and receive actions, $\mathbf{s}(ch)$, $\mathbf{r}(ch)$, a wait action for blocking on a receive $\mathbf{w}(ch)$, a reschedule action, $\mathbf{rs}(t)$, and a wait action for blocking on a reschedule, $\mathbf{wa}$. Formally,

$$a \in \mathit{Act} ::= \epsilon \mid \mathbf{b}(e, s) \mid \mathbf{a}(x, v) \mid \mathbf{f}(s) \mid \mathbf{s}(ch) \mid \mathbf{r}(ch) \mid \mathbf{w}(ch) \mid \mathbf{wa} \mid \mathbf{rs}(t)$$

The behavior of an action a on the state is given by the function $[\![a]\!]_A$ defined in Figure 4. The function tgen is used to generate a fresh thread identifier for newly forked threads. It thus satisfies the specification $\mathit{tgen}(T) \notin \mathit{dom}(T)$. We assume tgen is a fixed global function, but it is possible to generalize the semantics and allow the rescheduling function to also pick a new thread identifier generator. $\mathit{active}(T)$ denotes the set of active threads in T, i.e., $\mathit{active}(T) = \{t \in \mathit{dom}(T) \mid T(t) \neq \varepsilon\}$. The statement reduction relation is defined in Figure 5.

$$\begin{aligned}
[\![\epsilon]\!]_A(T, M, \rho, t) &= (T, M, \rho) \\
[\![\mathbf{b}(e, s)]\!]_A(T, M, \rho, t) &= (T[t \mapsto s :: T(t)], M, \rho) \\
[\![\mathbf{a}(x, v)]\!]_A(T, M, \rho, t) &= (T, M, \rho[x \mapsto v]) \\
[\![\mathbf{f}(s)]\!]_A(T, M, \rho, t) &= (T[\mathit{tgen}(T) \mapsto s], M, \rho) \\
[\![\mathbf{s}(ch)]\!]_A(T, M, \rho, t) &= (T, M[ch \mapsto M(ch) + 1], \rho) \\
[\![\mathbf{r}(ch)]\!]_A(T, M, \rho, t) &= \text{if } M(ch) > 0 \text{ then } (T, M[ch \mapsto M(ch) - 1], \rho) \text{ else } \bot \\
[\![\mathbf{w}(ch)]\!]_A(T, M, \rho, t) &= \text{if } M(ch) = 0 \text{ then } (T, M, \rho) \text{ else } \bot \\
[\![\mathbf{rs}(t')]\!]_A(T, M, \rho, t) &= \text{if } |\mathit{active}(T)| = 1 \text{ then } ([t' \mapsto T(t)], M, \rho) \text{ else } \bot \\
[\![\mathbf{wa}]\!]_A(T, M, \rho, t) &= \text{if } |\mathit{active}(T)| > 1 \text{ then } (T, M, \rho) \text{ else } \bot
\end{aligned}$$

Fig. 4. Semantics of actions.

Note that semantics of events is deterministic. For example, $\mathbf{r}(ch)$-transition can only be executed if $M(ch) > 0$, while $\mathbf{w}(ch)$ can only be emitted if $M(ch) > 0$ (symbol $\bot$ in the definition means "undefined"). Note that $\mathsf{reschedule}$ only reduces globally once all other threads in the thread pool have reduced fully and that it further removes all other threads from the thread pool upon reducing and assigns a new thread identifier to the only active thread. This requirement ensures that once $\mathsf{reschedule}$ reduces and resets the scheduler state then other threads that exist prior to the reduction of $\mathsf{reschedule}$ cannot immediately taint the scheduler state again. The reschedule reduction step is deterministic: the value of t is bound in the respective rule in Figure 2 by function Ψ.

Example 6. To illustrate the issue, consider the following code snippet. This program branches on a confidential (high) variable h and then spawns one of two threads with the sole purpose of tainting the scheduler with the state of h. It also contains a race on a public (low) variable l, which occurs after the rescheduling.

$$\begin{array}{l}\mathsf{if}\ h > 0\ \mathsf{then}\ \mathsf{fork}(\mathsf{skip})\ \mathsf{else}\ \mathsf{fork}(\mathsf{skip};\ \mathsf{skip});\\ \mathsf{reschedule};\\ \mathsf{fork}(l := 0);\ l := 1\end{array}$$

If $\mathsf{reschedule}$ could reduce and reset the scheduler state before the forked thread had reduced, then the forked thread could reduce between $\mathsf{reschedule}$ and the assignment and therefore affect which of the two racy assignments to l would win the race. Our operational semantics therefore only reduces $\mathsf{reschedule}$ once all other threads have terminated, which for the above example ensures that the forked thread has already fully reduced, and cannot taint the scheduler state after $\mathsf{reschedule}$ has reset it.

$$\mathsf{while}\ e\ \mathsf{do}\ s, \rho \to_\epsilon \mathsf{if}\ e\ \mathsf{then}\ (s; \mathsf{while}\ e\ \mathsf{do}\ s)\ \mathsf{else}\ \mathsf{skip}$$

$$\begin{array}{rl} \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \rho \to_{\mathbf{b}(e,s_1)} \mathsf{skip} & \text{if } [\![e]\!](\rho) = \mathsf{tt} \\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \rho \to_{\mathbf{b}(e,s_2)} \mathsf{skip} & \text{if } [\![e]\!](\rho) = \mathsf{ff} \\ x := e, \rho \to_{\mathbf{a}(x,v)} \mathsf{skip} & \text{where } v = [\![e]\!](\rho) \end{array}$$

$$\begin{array}{rr} \mathsf{skip}; s, \rho \to_\epsilon s & \mathsf{recv}(ch), \rho \to_{\mathbf{w}(ch)} \mathsf{recv}(ch) \\ \mathsf{fork}(s), \rho \to_{\mathbf{f}(s)} \mathsf{skip} & \mathsf{recv}(ch), \rho \to_{\mathbf{r}(ch)} \mathsf{skip} \\ \mathsf{send}(ch), \rho \to_{\mathbf{s}(ch)} \mathsf{skip} & \mathsf{reschedule}, \rho \to_{\mathbf{wa}} \mathsf{reschedule} \\ & \mathsf{reschedule}, \rho \to_{\mathbf{rs}(t)} \mathsf{skip} \end{array}$$

Fig. 5. Statement reduction.

2.1 Security model

In this section we introduce our formal security model for confidentiality. This is formalized as a non-interference property, requiring that attackers cannot learn anything about confidential inputs from observing public outputs.

To express this formally, we assume a bounded $\sqcup$-semilattice $\mathcal{L}$ of security levels for classifying the confidentiality levels of inputs and outputs. We say that level ℓ_1 *flows into* ℓ_2 if $\ell_1 \sqsubseteq \ell_2$. In examples we typically assume $\mathcal{L}$ is a bounded lattice with distinguished top and bottom elements, denoted H and L, and referred to as high and low, respectively. Given a security typing Γ that assigns security levels to all program variables and channel identifiers, we consider two heaps ρ_1 and ρ_2 indistinguishable at attacker level ℓ_A if the two heaps agree for all variables with a security level below or equal to the attacker security level:

$$\rho_1 \sim_{\Gamma}^{\ell_A} \rho_2 \overset{def}{=} \forall x \in \mathit{Var}.\, \Gamma(x) \sqsubseteq \ell_A \Rightarrow h_1(x) = h_2(x)$$

Likewise, we consider two message pools M_1 and M_2 indistinguishable at attacker level ℓ_A if they agree on all channels with security level below or equal to the attackers security level:

$$M_1 \sim_{\Gamma}^{\ell_A} M_2 \overset{def}{=} \forall ch \in \mathit{Chan}.\, \Gamma(ch) \sqsubseteq \ell_A \Rightarrow M_1(ch) = M_2(ch)$$

Non-interference expresses that attackers cannot learn confidential information by requiring that executions from attacker indistinguishable initial message pools and heaps should produce attacker indistinguishable terminal message pools and heaps, when executed from the same initial scheduler state and scheduling function. Since scheduling and rescheduling functions have complete access to the machine state, including confidential variables and channels, we restrict attention to schedulers and reschedulers that only access attacker-observable variables and channels. We say that a scheduler sf is an ℓ-scheduler iff it does not distinguish message pools and heaps that are ℓ-indistinguishable:

$$\begin{aligned}\ell\text{-}\mathit{level}(sf) \Leftrightarrow\ & \forall S, T, M_1, M_2, \rho_1, \rho_2. \\ & M_1 \sim_{\Gamma}^{\ell} M_2 \wedge \rho_1 \sim_{\Gamma}^{\ell} \rho_2 \Rightarrow sf(S, T, M_1, \rho_1) = sf(S, T, M_2, \rho_2)\end{aligned}$$

Likewise, a rescheduling function is an ℓ-rescheduler iff it does not distinguish message pools and heaps that are ℓ-indistinguishable and only returns ℓ-schedulers:

$$\begin{aligned}\ell\text{-}\mathit{level}(\Psi) \Leftrightarrow\ & \forall sf, M_1, M_2, \rho_1, \rho_2.\, \ell\text{-}\mathit{level}(\pi_1(\Psi(sf, M_1, \rho_1))) \wedge \\ & (M_1 \sim_{\Gamma}^{\ell} M_2 \wedge \rho_1 \sim_{\Gamma}^{\ell} \rho_2 \Rightarrow \Psi(sf, M_1, \rho_1) = \Psi(sf, M_2, \rho_2))\end{aligned}$$

where π_1 is a projection to the first component of the triple.

Definition 1 (Security). *A thread pool T satisfies* non-interference at attacker level ℓ_A and security typing Γ *iff all fully-reduced executions from ℓ_A-related initial heaps (starting with empty message pools) reduce to ℓ_A-related terminal heaps, for all ℓ_A-level schedulers sf and reschedulers Ψ:*

$$\begin{aligned}& \forall \rho_1, \rho_2, \rho_1', \rho_2' \in \mathit{Heap}.\, \forall M_1', M_2' \in \mathit{MPool}.\, \forall S, S_1', S_2' \in \mathcal{S}.\, \forall T_1', T_2'.\, \forall sf, sf_1', sf_2'. \\ & \quad \ell_A\text{-}\mathit{level}(sf) \wedge \ell_A\text{-}\mathit{level}(\Psi) \wedge \rho_1 \sim_{\Gamma}^{\ell_A} \rho_2 \wedge \mathit{final}(T_1') \wedge \mathit{final}(T_2') \wedge \\ & \quad sf, S, T, \lambda ch.0, \rho_1 \longrightarrow_{\Psi}^{*} sf_1', S_1', T_1', M_1', \rho_1' \wedge \\ & \quad sf, S, T, \lambda ch.0, \rho_2 \longrightarrow_{\Psi}^{*} sf_2', S_2', T_2', M_2', \rho_2' \Rightarrow M_1' \sim_{\Gamma}^{\ell_A} M_2' \wedge \rho_1' \sim_{\Gamma}^{\ell_A} \rho_2'\end{aligned}$$

where $\mathit{final}(T) \overset{def}{=} \forall t \in \mathit{dom}(T).\, T(t) = \varepsilon$.

This non-interference property can be specialized in the obvious way from thread pools to programs by considering a thread pool with only the given program.

In our security model, we focus on standard end-to-end security, i.e., the attacker is allowed to observe low parts of the initial and final heaps. The security definition quantifies over all possible schedulers, which in particular means that the attacker is allowed to choose any scheduler.

To develop some intuition about our security model, let's consider a few basic examples. The program $\mathsf{fork}(x := 1); x := 2$ is *secure* for any attacker level ℓ_A, because in any two executions from the same initial scheduler state and ℓ_A-equivalent initial message pools and heaps, the scheduler must schedule the assignments in the same order. This follows from the assumption that the scheduler cannot distinguish ℓ_A-equivalent message pools and heaps.

If prior to a race on a low variable a thread branches on confidential information, then we can construct a scheduler that leaks this information. To illustrate, consider the following variant of Example 1 from the Introduction:

```
fork(if h then skip else (skip; skip))        // Thread 1
fork(l := 1);                                 // Thread 2
fork(l := 2)                                  // Thread 3
```

If we take the scheduler state to be a natural number corresponding to the number of statements reduced so far, then we can construct a scheduler that first reduces Thread 1 and then schedules Thread 2 if Thread 1 was fully reduced in two steps and Thread 3 if Thread 1 was fully reduced in three steps. Therefore, this program is *not* secure.

3 Type system

In this section we present a type-and-effect system for establishing non-interference. The type-and-effect system is inspired by separation logic [36] and uses ideas of ownership and resources to track whether accesses to variables and channels may be racy and to bound the security level of the data that may be learned through observing how threads are scheduled. Statements are typed relative to a pre- and postcondition, where the precondition describes the resources necessary to run the statement and the postcondition the resources owned after executing the statement. The statement typing judgment has the following form:

$$\Gamma \mid \Delta \mid pc \vdash \{P\}\ s\ \{Q\}$$

Here P and Q are resources and pc is an upper bound on the security level of the data that can be learned through knowing the control of the program up to this point. Context Γ defines security levels for all program variables and channel identifiers and Δ defines a static resource specification for every channel identifier. We will return to these contexts later. Expressions are typed relative to a precondition and the expression typing judgment has the following form: $\Gamma \vdash \{P\}\ e : \ell$. Here ℓ is an upper bound on the security level of the data computed by e. Resources are described by the following grammar:

$$P, Q ::= emp \mid P * Q \mid x_\pi \mid ch_\pi \mid schd_\pi(\ell) \mid \lceil P \rceil^\ell$$

where $\pi \in \mathbb{Q} \cap (0, 1]$. The *emp* assertion describes the empty resource that does not assert ownership of anything. The $P * Q$ assertion describes a resource that can be split into two disjoint resources, P and Q, respectively. This assertion is inspired by separation logic and is used to reason about separation of resources.

Variable resources, written x_π, express fractional ownership of variable x with fraction $\pi \in \mathbb{Q} \cap (0, 1]$. We use these to reason locally about whether accesses to a given variable might cause a race. Ownership of the full fraction $\pi = 1$ expresses that we own the variable exclusively and can therefore access the variable without fears of causing a race. Any fraction less than 1 only expresses partial ownership and accessing the given variable could therefore cause a race. These variable resources can be split and recombined using the fraction. We express this using the resource entailment judgment, written $\Gamma \vdash P \Rightarrow Q$, which asserts that resource P can be converted into resource Q. We write $\Gamma \vdash P \Leftrightarrow Q$ when resource P can be converted into Q and Q can be converted into P. Splitting and recombination of variable resources comply with the rule: If $\pi_1 + \pi_2 \leq 1$ then $\Gamma \vdash x_{\pi_1+\pi_2} \Leftrightarrow x_{\pi_1} * x_{\pi_2}$. This can for instance be used to split an exclusive permission into two partial permissions that can be passed to two different threads and later recombined back into the exclusive permission.

The other kind of crucial resources, $schd_\pi(\ell)$, where $\pi \in \mathbb{Q} \cap (0, 1]$, allows us to track the scheduler level (also called the scheduler taint). A *labeled scheduler resource*, $schd_\pi(\ell)$, expresses that the scheduler level currently cannot go above ℓ. This is both a guarantee we give to the environment and something we can rely on the environment to follow. This guarantee ensures that level of information that can be learned by observing how threads are scheduled is bounded by the scheduler level. Again, we use fractional permissions to split the scheduler resource between multiple threads: If $\pi_1 + \pi_2 \leq 1$ then $\Gamma \vdash schd_{\pi_1+\pi_2}(\ell) \Leftrightarrow schd_{\pi_1}(\ell) * schd_{\pi_2}(\ell)$. If we own the scheduler resource exclusively, then no one else is relying on the scheduler level staying below a given security level and we can thus change the scheduler rely-guarantee level to a higher security level: If $\ell_1 \sqsubseteq \ell_2$ then $\Gamma \vdash schd_1(\ell_1) \Rightarrow schd_1(\ell_2)$. In general it is not secure to lower the upper bound on the scheduler level in this way, even if we own the scheduler resource exclusively. Instead, we must use `reschedule` to lower the scheduler level. We will return to this issue in a subsequent section.

State and control flow Before introducing the remaining resources, let's look at the typing rules for assignments and control flow primitives, to illustrate how we use these variable and scheduler resources. The type-and-effect system features two assignment rules, one for non-racy assignments and one for potentially racy assignments (T-Asgn-Excl and T-Asgn-Racy, respectively, in Figure 6). If we own a variable resource exclusively, then we can use the typing rule for non-racy assignments and we do not have to worry about leaking information

$$\frac{}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{skip}\ \{P\}}\ \textsc{T-Skip}$$

$$\frac{\Gamma \mid \Delta \mid pc \vdash \{P\}\ s_1\ \{R\} \qquad \Gamma \mid \Delta \mid pc \vdash \{R\}\ s_2\ \{Q\}}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ s_1; s_2\ \{Q\}}\ \textsc{T-Seq}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{P\}\ e : \ell \\ P \equiv R * schd_\pi(\ell_s) \qquad \ell \sqsubseteq \ell_s \qquad \Gamma \mid \Delta \mid pc \sqcup \ell \vdash \{P\}\ s_i\ \{Q\} \quad \mathit{for}\ i \in \{1,2\}\end{array}}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{Q\}}\ \textsc{T-If}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{P\}\ e : \ell \\ P \equiv R * schd_\pi(\ell_s) \qquad \ell \sqsubseteq \ell_s \qquad \Gamma \mid \Delta \mid pc \sqcup \ell \vdash \{P\}\ s\ \{P\}\end{array}}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{while}\ e\ \mathsf{do}\ s\ \{P\}}\ \textsc{T-While}$$

$$\frac{\Gamma \vdash \{P\}\ e : \ell \qquad \Gamma \vdash P \Rightarrow x_1 \qquad pc \sqcup \ell \sqsubseteq \Gamma(x)}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ x := e\ \{P\}}\ \textsc{T-Asgn-Excl}$$

$$\frac{\Gamma \vdash \{P\}\ e : \ell \qquad P \equiv R * schd_{\pi_s}(\ell_s) \qquad \Gamma \vdash P \Rightarrow x_\pi \qquad pc \sqcup \ell \sqcup \ell_s \sqsubseteq \Gamma(x)}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ x := e\ \{P\}}\ \textsc{T-Asgn-Racy}$$

Fig. 6. Typing rules for assignments and control flow statements.

through scheduling. However, if we only own a partial variable resource for a given variable, then any access to the variable could potentially introduce a race and we have to ensure information learned from scheduling is allowed to flow into the given variable. The typing rule for potentially racy assignments (T-Asgn-Racy) thus requires that we own a scheduler resource, $schd_\pi(\ell_s)$, that bounds the information that can be learned through scheduling, and requires that ℓ_s may flow into $\Gamma(x)$. Both assignment rules naturally also require that the security level of the assigned expression and the current pc-level is allowed to flow into the assigned variable. The assigned expression is typed using the expression

$$\overset{\textsc{T-Sub}}{\frac{\Gamma \vdash \{P\}\ e : \ell_1 \qquad \ell_1 \sqsubseteq \ell_2}{\Gamma \vdash \{P\}\ e : \ell_2}} \qquad \overset{\textsc{T-Const}}{\frac{}{\Gamma \vdash \{P\}\ v : \ell}} \qquad \overset{\textsc{T-Var}}{\frac{\Gamma \vdash P \Rightarrow x_\pi}{\Gamma \vdash \{P\}\ x : \Gamma(x)}}$$

$$\frac{\Gamma \vdash \{P\}\ e_1 : \ell \qquad \Gamma \vdash \{P\}\ e_2 : \ell}{\Gamma \vdash \{P\}\ e_1 + e_2 : \ell}\ \textsc{T-Add} \qquad \frac{\Gamma \vdash \{P\}\ e_1 : \ell \qquad \Gamma \vdash \{P\}\ e_2 : \ell}{\Gamma \vdash \{P\}\ e_1 = e_2 : \ell}\ \textsc{T-Eq}$$

Fig. 7. Typing rules for expressions.

typing judgment, $\Gamma \vdash \{P\}\ e : \ell$, using the rules from Figure 7. This judgment computes an upper-bound ℓ on the security-level of the data computed by the expression and ensures that P asserts at least partial ownership of any variables accessed by e. Hence, exclusive ownership of a given variable x ensures both the absence of write-write races to the given variable, but also read-write races, which can also be exploited to leak confidential information through scheduling.

$$\frac{\Gamma \mid \Delta \mid pc \vdash \{P\}\ s\ \{Q\}}{\Gamma \mid \Delta \mid pc \vdash \{P * R\}\ s\ \{Q * R\}}\ \textsc{T-Frame}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow P_2 \qquad \Gamma \mid \Delta \mid pc_2 \vdash \{P_2\}\ s\ \{Q_2\} \qquad \Gamma \vdash Q_2 \Rightarrow Q_1 \qquad pc_1 \sqsubseteq pc_2}{\Gamma \mid \Delta \mid pc_1 \vdash \{P_1\}\ s\ \{Q_1\}}\ \textsc{T-Conseq}$$

Fig. 8. Structural typing rules.

The typing rules for conditionals and loops (T-If and T-While) both require ownership of a scheduler resource with a scheduler level ℓ_s and this scheduler level must be an upper bound on the security level of the branching expression. The structural rule of consequence (T-Conseq in Figure 8) allows to strengthen preconditions and weaken postconditions. In particular, in conjunction with resource implication rules (Figure 9), it allows to raise the level of scheduler resource, which is necessary to type branching on high-security data.

$$\frac{}{\Gamma \vdash P \Rightarrow P * emp} \qquad \frac{}{\Gamma \vdash P * Q \Rightarrow Q * P} \qquad \frac{}{\Gamma \vdash (P * Q) * R \Leftrightarrow P * (Q * R)}$$

$$\frac{}{\Gamma \vdash P * Q \Rightarrow P} \qquad \frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash Q \Rightarrow R}{\Gamma \vdash P \Rightarrow R} \qquad \frac{\Gamma \vdash P \Rightarrow Q}{\Gamma \vdash P * R \Rightarrow Q * R}$$

$$\frac{\pi_1 + \pi_2 \leq 1}{\Gamma \vdash x_{\pi_1} * x_{\pi_2} \Leftrightarrow x_{\pi_1 + \pi_2}} \qquad \frac{\pi_1 + \pi_2 \leq 1}{\Gamma \vdash schd_{\pi_1}(\ell) * schd_{\pi_2}(\ell) \Leftrightarrow schd_{\pi_1 + \pi_2}(\ell)}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\Gamma \vdash schd_1(\ell_1) \Rightarrow schd_1(\ell_2)}$$

Fig. 9. Resource implication rules.

Spawning threads When spawning a new thread, the spawning thread is able to transfer some of its resources to the newly created thread. This is captured by the T-Fork rule given below, which transfers the resources described by P from the spawning thread to the spawned thread.

$$\frac{\Gamma \mid \Delta \mid pc \vdash \{P\}\ s\ \{Q\}}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{fork}(s)\ \{emp\}}\ \textsc{T-Fork}$$

Naturally, the newly spawned thread inherits the pc-level of the spawning thread. Upon termination of the spawned thread, the resources still owned by the spawned thread are lost. To transfer resources back to the spawning thread or other threads requires synchronization using channels.

Synchronization From the point of view of resources, synchronization is about transferring ownership of resources between threads. When sending a message on a channel, we relinquish ownership of some of our resources, which become associated with the message until it is read. Conversely, when reading from a channel the reader may take ownership of a part of the resource associated with the message it reads. The Δ context defines a static specification for every channel identifier that describes the resources we wish to associate with messages on the given channel. If $\Delta(ch) = P$, then we must transfer resource P when sending a message on channel ch. However, when receiving a message from channel ch, we might only be able to acquire part of P, depending on whether our receive may race with other receives to acquire the resources and how our pc-level relates to the pc-level of the sender of the message and to the potential scheduler taint.

To capture this formally, our type-and-effect system contains channel resources, written ch_π, erased resources, written $\lceil P \rceil^\ell$, and channel security levels, $\Gamma(ch)$. Like variable resources, channel resources allow us to track whether a given receive operation on a channel might race with another receive on the same channel using a fraction π. To receive on a channel ch requires fractional ownership of the corresponding channel resource. The channel resource can be split and recombined freely: $\Gamma \vdash ch_{\pi_1+\pi_2} \Leftrightarrow ch_{\pi_1} * ch_{\pi_2}$, with the full fraction, $\pi = 1$, indicating an exclusive right to receive on the given channel. The erased resource, $\lceil P \rceil^\ell$, is used to erase variable and channel resources in P with security levels that are not greater than or equal to the security level ℓ. To illustrate how we use these features to type send and receive commands, let us start by considering an example that is *not* secure, and that should therefore *not* be typeable.

We start with the simpler case of non-racy receives. In the case of non-racy receives, we have to prevent ownership transfer of low variables from a high security context to a lower security context. This is illustrated by the program

$$\begin{aligned}&\mathsf{fork}(\mathsf{if}\ h\ \mathsf{then}\ \mathsf{send}(a)\ \mathsf{else}\ \mathsf{send}(b));\\ &\mathsf{fork}(\mathsf{recv}(a);\ l := 1;\ \mathsf{send}(b));\\ &\mathsf{fork}(\mathsf{recv}(b);\ l := 2;\ \mathsf{send}(a))\end{aligned}$$

This code snippet spawns a thread which sends a message on either channel a or b depending on the value of the confidential variable h. Then the program spawns two other threads that wait until there is an available message on their channel, before they write to l and message the other thread that it may proceed. This code snippet is insecure, because if h is initially true, then the public variable l will contain the value 2 upon termination and if h is initially false, then l will contain the value 1.

$$\frac{pc \sqsubseteq \Gamma(ch)}{\Gamma \mid \Delta \mid pc \vdash \{\Delta(ch)\}\ \mathsf{send}(ch)\ \{emp\}}\ \textsc{T-Send}$$

$$\frac{P \equiv R * schd_{\pi_s}(\ell_s) \qquad pc \sqsubseteq \Gamma(ch) \sqsubseteq \ell_s \qquad \Gamma \vdash P \Rightarrow ch_1}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{recv}(ch)\ \{P * \lceil \Delta(ch) \rceil^{\Gamma(ch)}\}}\ \textsc{T-Recv-Excl}$$

$$\frac{P \equiv R * schd_{\pi_s}(\ell_s) \qquad pc \sqsubseteq \Gamma(ch) = \ell_s \qquad \Gamma \vdash P \Rightarrow ch_\pi}{\Gamma \mid \Delta \mid pc \vdash \{P\}\ \mathsf{recv}(ch)\ \{P * \lceil \Delta(ch) \rceil^{\Gamma(ch)}\}}\ \textsc{T-Recv-Racy}$$

Fig. 10. Typing rules for synchronization primitives.

To type this program, the idea would be to transfer exclusive ownership of the public variable l along channels a and b. However, our type system prevents this by erasing the resources received along channels a and b at the high security level, because the first thread may send messages on a and b in a high security context (i.e., with a high pc-level).

Formally, the typing rules for send and for exclusive receives are given by T-Send and T-Recv-Excl in Figure 10. The send rule requires that the security level of the channel is greater than or equal to the sender's pc-level and the exclusive receive rule erases the resources received from the channel using the security-level of the channel. This means that the second and third threads do not get exclusive ownership of the l variable and that we therefore cannot type the subsequent assignments. The exclusive receive rule also requires fractional ownership of the scheduler resource and that the bound on the taint on the scheduler level is greater than or equal to the channel security level when receiving on a channel. This condition is related to the use of reschedule and we will return to this condition later.

Example 7. To illustrate how to use these rules for ownership transfer, consider the following variant of the examples from the introduction.

$$\begin{aligned} ex_7 \overset{def}{=} & \ \mathsf{fork}(\mathsf{if}\ h\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2);\ \textit{/* high computation */} \\ & \mathsf{fork}(l := 1;\ \mathsf{send}(c)); \\ & \mathsf{recv}(c);\ l := 2 \end{aligned}$$

It forks off a thread that does a high computation and potentially taints the scheduler with confidential information. The main thread also forks off a new thread that performs a write to public variable l, before itself writing to l. However, a communication through channel c in between these two assignments ensure that they are not racy and therefore do not leak private information for any chosen scheduling. We can, for instance, type this example as follows:

$$\Gamma \mid \Delta \mid L \vdash \{c_1 * l_1 * h_1 * schd_1(L)\}\ ex_7\ \{c_1 * l_1 * schd_{\frac{1}{2}}(H)\}$$

where Γ and Δ are defined as follows: $\Gamma(l) = \Gamma(c) = L$, $\Gamma(h) = H$, and $\Delta(c) = l_1$.

This typing requires the main thread to pass exclusive ownership of l to the second thread upon forking, which is then passed back on channel c. Since we only send and receive on channel c in a low context, we can take the channel security level to be low for c. When the main thread receives a message on c it thus takes ownership of $\lceil l_1 \rceil^{\Gamma(c)}$ and since $\Gamma(c) = L$, it follows that $\Gamma \vdash \lceil l_1 \rceil^{\Gamma(c)} \Rightarrow l_1$. The main thread thus owns the variable resource for l exclusively when typing the second assignment.

We use the resource implication rules in Figure 11 to reason about erased resources, by pulling resources out of the erasure. For instance, if the security level of a variable x is greater than or equal to the erasure security level, then we can pull it out of the erasure: if $\ell \sqsubseteq \Gamma(x)$ then $\Gamma \vdash \lceil x_\pi \rceil^\ell \Rightarrow x_\pi$; and likewise for channel resources: if $\ell \sqsubseteq \Gamma(ch)$ then $\Gamma \vdash \lceil ch_\pi \rceil^\ell \Rightarrow ch_\pi$. Resources that cannot be pulled out of the erasure cannot be used for anything; owning $\lceil x_\pi \rceil^\ell$ where $\Gamma(x) \not\sqsubseteq \ell$ is thus equivalent to owning emp. The full set of erasure implication rules is given in Figure 11. Notice that scheduler resources never get erased: $\Gamma \vdash \lceil schd_\pi(\ell_s) \rceil^\ell \Rightarrow schd_\pi(\ell_s)$. Moreover, the resource erasure is idempotent and distributes over the star operator.

$$\frac{\ell \sqsubseteq \Gamma(x)}{\Gamma \vdash \lceil x_\pi \rceil^\ell \Rightarrow x_\pi} \qquad \frac{\ell \sqsubseteq \Gamma(ch)}{\Gamma \vdash \lceil ch_\pi \rceil^\ell \Rightarrow ch_\pi} \qquad \frac{}{\Gamma \vdash \lceil schd_\pi(\ell_s) \rceil^\ell \Rightarrow schd_\pi(\ell_s)}$$

$$\frac{}{\Gamma \vdash \lceil \lceil P \rceil^\ell \rceil^\ell \Rightarrow \lceil P \rceil^\ell} \qquad \frac{}{\Gamma \vdash \lceil P_1 * P_2 \rceil^\ell \Rightarrow \lceil P_1 \rceil^\ell * \lceil P_2 \rceil^\ell}$$

Fig. 11. Erasure implication rules

Racy synchronization In the case of racy receives, where we have multiple threads racing to take ownership of a message on the same channel, we have to restrict which resources the receivers can take ownership of even further. This is best illustrated with another example of an insecure program. The following is a variant of the earlier insecure program, but instead of sending a message on a channel in a high context it sends a message on a channel in a low context after the scheduler has been tainted and the scheduler level has been raised to high.

```
if h then skip else (skip; skip);
send(c);
fork(recv(c); l := 1; send(c));
recv(c); l := 2; send(c)
```

With a suitably chosen scheduler, the initial value of the confidential variable h could decide which of the two racy receives will receive the initial message on c and thereby leak the initial value of h through the public variable l. We thus have to ensure that this program is *not* typeable. Our type system ensures that this is the case by requiring the scheduler level to equal the channel security level when performing a potentially racy receive. In the case of the example above, the scheduler level gets high after the high branching and is still high when we type check the two receives; since they are racy we are forced to set the security level of channel c to high — see the typing rule T-Recv-Racy for racy receives in Figure 10 — which ensures we cannot transfer ownership of the public variable l on c. This in turn ensures that we cannot type the assignments to l as exclusive assignments and therefore that the example is not typeable.

Reschedule Recall that if we own the scheduler resource exclusively, then we can freely raise the upper bound on the security level of the scheduler, since no other threads are relying on any upper bound. In general, it is not sound to lower this upper bound, unless we can guarantee that the current scheduler level is less than or equal to the new upper bound. This is exactly what the reschedule statement ensures. The typing rule for reschedule (T-Resched given below) thus requires exclusive ownership of the scheduler resource and allows us to change this upper bound to any security level we wish. To ensure soundness, we only allow $\mathsf{reschedule}$ to be used when the pc-level is $\bot_{\mathcal{L}}$, the bottom security level of the semilattice of security elements.

$$\frac{}{\Gamma \mid \Delta \mid \bot_{\mathcal{L}} \vdash \{schd_1(\ell_1)\}\ \mathsf{reschedule}\ \{schd_1(\ell_2)\}}\ \textsc{T-Resched}$$

Example 8. To illustrate how the typing rule for $\mathsf{reschedule}$ is used, consider the following code snippet from the introduction section:

$$\begin{aligned} ex_8 \overset{def}{=} &\ \mathsf{if}\ h\ \mathsf{then\ skip\ else\ (skip;\ skip)};\\ &\ \mathsf{reschedule};\\ &\ \mathsf{fork}(l := 0);\ l := 1 \end{aligned}$$

Recall that this snippet is secure, since reschedule resets the scheduler state before the race on l. We can, for instance, type this example as follows:

$$\Gamma \mid \Delta \mid L \vdash \{l_1 * h_1 * schd_1(L)\}\ ex_8\ \{l_{\frac{1}{2}} * schd_{\frac{1}{2}}(L)\}$$

with $\Gamma(l) = L$ and $\Gamma(h) = H$.

To type this example we first raise the upper bound on the scheduler level from low to high, so that we can branch on confidential h. Then we use T-Resched to reset it back to low after $\mathsf{reschedule}$. At this point we split both the scheduler and variable resource for variable l into two, keep one part of each for the main thread and give away one part of each to the newly spawned thread. The two assignments to l are now typed by T-Asgn-Racy rule.

Example 9. To illustrate why we only allow $\mathsf{reschedule}$ to be used at pc-level $\bot_{\mathcal{L}}$, consider the following example, which branches on the confidential variable h before executing $\mathsf{reschedule}$ in both branches.

$$\begin{aligned} &\mathsf{fork(if}\ h\ \mathsf{then\ (reschedule;\ skip)\ else\ (reschedule;\ skip;\ skip))};\\ &\mathsf{fork}(l := 0);\ l := 1 \end{aligned}$$

Despite doing a reschedule in both branches, the subsequent statements in the two branches immediately taint the scheduler with information about h again, after the scheduler has been reset. This example is thus not safe.

In the full version of the paper, the reader will find several more intricate examples justifying the constraints of the rules.

Precision of the type system Notice that mere racy reading or writing from/to variables does not taint the scheduler. For example, programs

$$\begin{aligned} &\mathsf{fork}(l := 1);\ \mathsf{fork}(m := l);\ \mathsf{fork}(h := 0);\ h := 1\\ &\mathsf{fork}(l := 0);\ h := h + 1;\ l := 1\\ &\mathsf{if}\ l\ \mathsf{then}\ h := 0\ \mathsf{else}\ h := 1; (\mathsf{fork}(l := 0); l := 1) \end{aligned}$$

where l, m are low variables and h is a high variable, are all secure in the sense of Definition 1 and are typable. Indeed, there is no way to exploit scheduling to leak the secret value h in either of these programs. The scheduler may get tainted only if a high branch or receiving from a high channel is encountered, since the number of computation steps for the remaining computation (and hence its scheduling) may depend on a secret value as, for example, in the program $\mathsf{while}\ h\ \mathsf{do}\ h := h - 1;\ (\mathsf{fork}(l := 0);\ l := 1)$. This example is rejected by our type system. To re-enable low races in the last example, rescheduling must be used:

$$\mathsf{while}\ h\ \mathsf{do}\ h := h - 1;\ \mathsf{reschedule};\ (\mathsf{fork}(l := 0);\ l := 1)$$

The last example is secure and accepted by the type system.

Limitations of our type system include imprecisions such as when both branches of a secret-dependent if-statement take the same number of steps, e.g., $\mathsf{if}\ h\ \mathsf{then\ skip\ else\ skip};(\mathsf{fork}(l := 0); l := 1)$, and standard imprecisions of flow-insensitive type-based approaches to information flow that reject programs such as in $\mathsf{if}\ h\ \mathsf{then}\ l := 0\ \mathsf{else}\ l := 0$ or in $(\mathsf{if}\ h\ \mathsf{then}\ l := 0\ \mathsf{else}\ l := 1);\ l := 42$.

Language extensions We believe that the ideas of this section can be extended to richer languages using standard techniques [32, 51, 17]. In particular, to handle a language with procedures we would use a separate environment to record types for procedures, similarly to what is done in, e.g., [34]. (In loc. cit. they did not cover concurrency; however, we take inspiration from [12] which presents a concurrent separation logic for a language with procedures and mutable stack variables.) Specifications for procedures would involve quantification over variables and security levels.

4 Soundness

Let T be a thread pool and let $\overline{P}$, $\overline{Q}$ map every thread identifier to $t \in \mathit{dom}(T)$ to a resource. We write $\Gamma \mid \Delta \vdash \{\overline{P}\}\ T\ \{\overline{Q}\}$ if $\overline{P}(t)$ and $\overline{Q}(t)$ are typing resources for every thread $T(t)$ with respect to Γ and Δ. We say that resource R is *compatible* if implication $\Gamma \vdash \circledast_{x \in \mathit{Var}} x_1 * \circledast_{ch \in \mathit{Chan}} ch_1 * \mathit{schd}_1(L) \Rightarrow R$ is provable.

Theorem 1 (Soundness). *Let $\Gamma \mid \Delta \vdash \{\overline{P}\}\ T\ \{\overline{Q}\}$ such that the composition of all the resources in $\overline{P}$ is compatible, then T satisfies non-interference for all attacker levels ℓ_A.*

Notice that the theorem quantifies universally over all attacker levels ℓ_A, hence, one typing is sufficient to guarantee security against all possible adversaries.

As a direct corollary from the theorem, we obtain a *compositionality* property for our type-and-effect system: Given two programs s_1, s_2 typable with preconditions P_1 and P_2, respectively, if $P_1 * P_2$ is compatible then the parallel composition of the two programs is typable with precondition $P_1 * P_2$.

Our soundness proof is inspired by previous non-interference results proved using a combination of erasure and confluence[6] for erased programs, but requires a number of novel techniques related to our reschedule construct, scheduler resources and support for benign races. A proof of Theorem 1 can be found in the full version of the paper.

[6] a property which guarantees that a given program can be reduced in different orders but yields the same result (up to a suitable equivalence relation).

5 Related work

The problem of securing information flow in concurrent programs has received widespread attention. We review the relevant literature along the following three dimensions:

(1) *Scheduler-(in)dependence.* Sabelfeld and Sands [41] argue for importance of scheduler independence because in practice it may be difficult to accommodate for precise scheduler behavior under all circumstances, and attackers aware of the scheduler specifics can use that knowledge to their advantage, also known as refinement attacks. However, designing a scheduler independent enforcement technique that is also practical comes at a price of additional restrictions. To this extent, a number of approaches gain permissiveness via scheduler support. This is manifested either as an assumption on a particular scheduling algorithm, i.e., round-robin, or scheduler awareness of security levels of the individual threads.

(2) *Permissiveness w.r.t. low races.* We are interested in seeing which of the approaches support benign low non-determinism and permit low races. We believe this is an important factor from a practical perspective, because an approach capable of handling low races has the potential of scaling to practical settings where parallel access, without extra synchronization overhead, to a single attacker-observable resource, such as network I/O, is desirable.

(3) *Termination-(in)sensitivity.* In sequential programs, ignoring leaks via program divergence is often a pragmatic choice, because the attacker is limited in how much information can be learned via the termination channel [3]. Can this pragmatic argument be carried over to a concurrent setting? On the one hand, malicious code with privileges to spawn threads may efficiently leak an N-bit secret by creating N threads and assigning every thread to leak a specific secret bit via the thread's termination behavior [48]. Motivated by this, many approaches reject programs that may potentially diverge depending on a secret. On the other hand, while it is possible to use techniques from literature on program termination to improve precision of the enforcement [29], a pragmatic attacker can instead use provably-terminating programs that take as much time as it is necessary for them to make their observations. So, for malicious code, one really needs to focus on the timing. But controlling timing behavior is difficult already in sequential programs, because many runtime aspects that have no source-level representation are in play, including hardware caches [50], memory management [35], or lazy evaluation [11].

Another reason for our attention on termination-(in)sensitivity is that it is our experience that technical restrictions that impose termination (or timing)-sensitivity often simplify soundness proofs. Without such restrictions, proving soundness for a (weaker) termination-insensitive definition can be more laborious.

Figure 12 presents a high-level summary of the related work. The figure is by no means exhaustive and lists only a few representative works; we discuss the other related papers below. Observe how the literature is divided across two diametric quadrants. Approaches that prioritize scheduler independence are conservative in their treatment of low races. Approaches that do permit low races require specific scheduler support are confined to particular classes of schedulers.

	Scheduler-dependent or restricted to particular scheduler classes	Scheduler-independent
Low races allowed	**TI:** [30] **TS:** [9, 38, 25, 39, 7, 24, 45, 4, 10]	**TI:** [14] (whole-program), $\star$ **TS:** [41] (+timing-sensitive)
Low races forbidden	-	**TI:** [49, 16, 46] **TS:** [16, 26]

Fig. 12. Summary of the related work w.r.t. permissiveness of the language-based enforcement and scheduler dependence. **TI** stands for *termination-insensitive*; **TS** stands for *termination-sensitive*.

We discuss these quadrants in detail, followed by the discussion of rely-guarantee style reasoning for concurrent information flow and rescheduling.

5.1 Scheduler-independent approaches

Observational determinism The approach of preventing races to individual locations is initiated in the work on observational determinism by Zdancewic and Myers [49] (which itself draws upon the ideas of McLean [27] and Roscoe [37]). Subsequent efforts on observational determinism include the work by Huisman et al. [16] and by Terauchi [46]. Here, Huisman et al. identify an issue in the Zdancewic and Myers' definition of security — they construct a leaky program within the intended attacker model, i.e., not exploiting termination or timing, that is accepted by the definition (though it is ruled out by the type system). They also propose a modified definition and show how to enforce that using self-composition [8]. Terauchi's paper presents a capability system with an inference algorithm for enforcing a restricted version of the Zdancewic and Myers' definition.

Out of these, the work by Terauchi [46] is the closest to ours because of the use of fractional permissions, but there are important differences in the treatment of the low races and the underlying semantic condition. Terauchi's type system is motivated by the design goal to reject racy programs of the form $l := 0 \parallel l := 1$. This is done through tracking fractional permissions on so-called abstract locations that represent a set of locations whose identity cannot be separated statically. Our type system uses fractional permissions in a similar spirit, but has additional expressivity, (even without the scheduler resource), because Terauchi's typing also rules out programs such as $l_1 := 0 \parallel l_2 := 1$, even when l_1 and l_2 are statically known to be non-aliasing. This is because the type system has a restriction that groups *all low variables* into a single abstract location. While this restriction is a necessity if the attacker is assumed to observe the order of individual low assignments, this effectively forces synchronization of all low-updating threads, regardless of whether the updates are potentially racy or not. We do not have such a restriction in our model.

We suspect that lifting this restriction in the Terauchi's system to accommodate a more permissive attacker model such as ours may be difficult without further changes to the type system, because their semantic security condition, being a variant of the one by Zdancewic and Myers [49], requires trace equivalence up to prefixing (and stuttering) for all locations in the set of the abstract low location. Without the typing restriction, the definition would appear to have the same semantic issue discovered by Huisman et al. [16]; the issue does not manifest itself with the restriction.

Note that adapting the security condition proposed by Huisman et al. [16] into a language-based setting also appears tricky. The paper [16] presents both termination-insensitive and termination-sensitive variants of their take on observational determinism. The key changes are the use of infinite traces instead of finite ones and requiring trace equivalence instead of prefix-equivalence (up to stuttering). Terauchi expresses their concerns w.r.t. applicability of this definition ([46], Appendix A). We think there is an additional concern w.r.t. termination-insensitivity. Because the TI-definition requires equivalence of infinite low traces it rejects a program such as

$$l := 1; \mathsf{while}\ secret = 1\ \mathsf{do\ skip};\ l := 2; \mathsf{while}\ secret = 2\ \mathsf{do\ skip}$$

This single-threaded program is a variant of a brute-force attack that is usually accepted by termination-insensitive definitions [3] and language-based techniques for information flow. We, thus, agree with the Terauchi's conclusion [46] that enforcing such a condition via a type-based method without being overly conservative may prove difficult.

By contrast, our approach builds upon the technique of explicit refiners [33, 30], which allows non-determinism as long as it is not influenced by secrets, and does not exhibit the aforementioned semantic pitfalls.

Whole program analysis can be used to enforce concurrent non-interference with a high precision. Giffhorn and Snelting [14] use a PDG-based whole program analysis to enforce relaxed low-security observational determinism (RLSOD) in Java programs. RLSOD is similar to our security condition in that it allows low-nondeterminism as long as it does not depend on secrets.

Strong security Sabelfeld and Sands [41] present a definition of *strong security* that is a compositional semantic condition for a natural class of schedulers. The compositionality is attained by placing timing-sensitivity constraints on individual threads. This condition serves as a foundation for a number of works [22, 13, 19]. To establish timing sensitivity, these approaches often rely on program transformation [1, 28, 6, 19]. A common limitation of the transformation-based techniques is that they do not apply to programs with high loops. Another concern is their general applicability, given the complexity of modern runtimes. A recent empirical study by Mantel and Starostin [23] investigates performance and security implications of these techniques, but as an initial step in this direction the paper [23] has a number of simplifying assumptions, such as disabled JIT optimizations and non-malicious code.

5.2 Scheduler-dependent approaches

Scheduler-dependent approaches vary in their assumptions on the underlying scheduler. Boudol and Castellani [9] study system and threads model where the scheduler code is explicit in the program source; a typing discipline regulates the secure interaction of the scheduler with the rest of the program [5].

Security-aware schedulers [38, 7] track security levels of the program counters of each thread, and provide the interface that timing of high computations is not revealed to the low ones; this interface is realized by suspending all low threads when there is an alive high thread.

A number of approaches assume a particular scheduling strategy, typically round-robin [39, 30, 45]. Mantel and Sudbrock [24] define a class of *robust* schedulers as the schedulers where "the scheduling order of low threads does not depend on the high threads in a thread pool" [24]. The class of robust schedulers appears to be large enough to include a number of practical schedulers, including round-robin. Other works rely on nondeterministic [44, 40, 8, 25, 21, 4] or probabilistically uniform [43, 47, 10] behavior.

5.3 Rely-guarantee style reasoning for concurrent information flow and rescheduling

Rely-guarantee style reasoning Mantel et al. [26] develops a different rely-guarantee style compositional approach for concurrent non-interference in flow-sensitive settings. In this approach, permissions to read or write variables are expressed using special *data access modes*; a thread can obtain an exclusive read access or an exclusive write access via the specific mode. Note that the modes are different from fractional permissions, because, e.g., an exclusive write access to a variable does not automatically grant the exclusive read access. The modes also do not have a moral equivalent of the scheduler resource. Instead, the paper [26] suggests using an external may-happen-in-parallel global analysis to track their global consistency. Askarov et al. [4] give modes a runtime representation, and use a hybrid information flow monitor to establish concurrent non-interference. Li et al. [20] use rely-guarantee style reasoning to reason about information flows in a message-passing distributed settings, where scheduler cannot be controlled. Murray et al. [31] use mode-based reasoning in a flow-sensitive dependent type system to enforce timing-sensitive value-dependent non-interference for shared memory concurrent programs.

Rescheduling The idea of barrier synchronization to recover permissiveness of language-based enforcement appears in papers with possibilistic scheduling [25, 4]. The rescheduling however does more than simple barrier synchronization—it also explicitly resets the scheduler state, which is crucial to avoid refinement attacks. The reason that simple barrier synchronization is insufficient is that despite synchronization at the barrier point, the scheduler state could be tainted by what happens before threads reach the barrier. For example, if the scheduler is implemented so that, after the barrier, the threads are scheduled to run in the order they have arrived to the barrier then there is little to be gained from the barrier synchronization.

Operationally, the reschedule is implementable in a straightforward manner, which is much simpler than security-aware schedulers [38, 7]. We note that rescheduling allows programmers to explore the space of performance/expressivity without losing security. A program that type checks without reschedule, because there are no dangerous race conditions, does not need to suffer from the performance overhead of the rescheduling. Programmers only need to add the reschedule instruction if they wish to re-enable low races after the scheduler was tainted. In that light, rescheduling is no less practical than the earlier mentioned barrier synchronization [4].

While on one hand the need to reschedule appears heavy-handed, we are not aware of other techniques that re-enable low races when the scheduler can be tainted. How exactly the scheduler gets tainted depends on the scheduler implementation/model. Presently, we assume that any local control flow that depends on secrets may taint the scheduler. This conservative assumption can naturally be relaxed for more precise/realistic scheduler models. Future research efforts will focus on refining scheduler models to reduce the need for rescheduling and/or automatic placement of rescheduling to lessen the burden on programmers. The latter can utilize techniques from the literature on the automatic placement of declassifications [18].

5.4 This work in the context of Figure 12

Developing a sound compositional technique for concurrent information flow that is scheduler-independent, low-nondeterministic, and termination-insensitive at the same time—a point marked by the star symbol in Figure 12—is a tall order, but we believe we come close. Our only non-standard operation is reschedule that we argue has a simple operational implementation and can be introduced to many existing runtimes.

6 Conclusion and Future Work

In the paper, we have presented a new compositional model for enforcing information flow security against internal timing leaks for concurrent imperative programs. The model includes a compositional fine-grained type-and-effect system and a novel programming construct for resetting a scheduler state. The type system is agnostic in the level of adversary, which means that one typing judgment is sufficient to ensure security for all possible attacker level. We formulate and prove the soundness result for the type system.

In future work, we wish to support I/O; our proof technique appears to have all the necessary ingredients for that. Moreover, we wish to investigate a generalization of our concurrency model to an X10-like [42, 30] setting where instead of one scheduler, we have several coarse-grained scheduling partitions.

Acknowledgments Thanks are due to Andrei Sabelfeld, Deepak Garg and the anonymous reviewers for their comments on this paper. This research was supported in part by the ModuRes Sapere Aude Advanced Grant, DFF projects no. 4181-00273 and no. 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU), and in part by Aarhus University Research Foundation.

Bibliography

[1] Johan Agat. Transforming out timing leaks. In *POPL*, 2000.

[2] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, 2009.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[4] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In *CSF*, 2015.

[5] Gilles Barthe and Leonor Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *FMSE*, 2004.

[6] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electr. Notes Theor. Comput. Sci.*, 153(2), 2006.

[7] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[8] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21 (6), 2011.

[9] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2), 2002.

[10] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. On improvements of low-deterministic security. In *POST*, 2016.

[11] Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Secure IT Systems*, 2013.

[12] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper - automatic verification for fine-grained concurrency. In *ESOP*, 2017.

[13] Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *FOSSACS*, 2005.

[14] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *Int. J. Inf. Sec.*, 14(3), 2015.

[15] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Security and Privacy*, 1982.

[16] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *CSFW*, 2006.

[17] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, 2006.

[18] Dave King, Susmit Jha, Divya Muthukumaran, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. Automating security mediation placement. In *ESOP*, 2010.

[19] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Sec.*, 6(2-3), 2007.

[20] Ximeng Li, Heiko Mantel, and Markus Tasch. Taming message-passing communication in compositional reasoning about confidentiality. In *APLAS*, 2017.

[21] Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In *ESOP*, 2007.

[22] Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11 (4), 2003.

[23] Heiko Mantel and Artem Starostin. Transforming out timing leaks, more or less. In *ESORICS*, 2015.

[24] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *ESORICS*, 2010.

[25] Heiko Mantel, Henning Sudbrock, and Tina Kraußer. Combining different proof techniques for verifying information flow security. In *LOPSTR*, 2006.

[26] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF*, 2011.

[27] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.

[28] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005.

[29] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-sensitive security. In *CCS*, 2012.

[30] Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *OOPSLA*, 2012.

[31] Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *CSF*, 2016.

[32] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, 1999.

[33] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *CSFW*, 2006.

[34] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[35] Mathias V. Pedersen and Aslan Askarov. From trash to treasure: Timing-sensitive garbage collection. In *Security and Privacy*, 2017.

[36] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[37] A. W. Roscoe. CSP and determinism in security modelling. In *Security and Privacy*, 1995.

[38] Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler. In *CSFW*, 2006.

[39] Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In *ASIAN*, 2006.

[40] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *PSI*, 2001.

[41] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, 2000.

[42] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, January 2012.

[43] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *CSFW*, 2003.

[44] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998.

[45] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[46] Tachio Terauchi. A type system for observational determinism. In *CSF*, 2008.

[47] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.

[48] Dennis M. Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *POPL*, 2000.

[49] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, 2003.

[50] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[51] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference (extended abstract). In *FAST*, 2004.