Thesis for the Degree of Doctor of Philosophy

Language Support for Controlling Timing-Based Covert Channels

Alejandro Russo

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2008

Language Support for Controlling Timing-Based Covert Channels
Alejandro Russo

ISBN: 978-91-7385-171-8
© Alejandro Russo, 2008

Technical Report nr: 47D
ISSN: 0346-718X (Doktorsavhandlingar vid Chalmers tekniska högskola)
Ny serie nr: 2852
Department of Computer Science and Engineering
Research group: Computer Security

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers Reproservice
Göteborg, October 2008

Abstract

The problem of controlling information flow in multithreaded programs remains an important open challenge. A major difficulty for tracking information flow in concurrent programs is due to the *internal timing covert channel*. Information is leaked via this channel when secrets affect the timing behavior of a thread, which, via the scheduler, affects the interleaving of public events. This channel is particularly dangerous because, in contrast to external timing, the attacker does not need to observe the actual execution time of programs.

This thesis introduces a novel treatment of the *interaction between threads and the scheduler*. As a result, a permissive security specification and a compositional security type system are obtained. The type system guarantees security for a wide class of schedulers and provides a flexible treatment of dynamic thread creation and synchronization. The approach relies on the modification of the scheduler in the run-time environment.

In some scenarios, the modification of the run-time environment might not be an acceptable requirement. For such scenarios, the thesis presents two transformations that eliminate the need for modifying the scheduler while avoiding internal timing leaks. The first transformation is given for programs running under cooperative schedulers. It states that threads must not yield control inside of computations that branch on secrets. The second transformation closes internal timing channel when the scheduler is preemptive and behaves as round-robin. It spawns dedicated threads, whenever computation may affect secrets, and carefully synchronizes them.

This dissertation also presents two libraries for information-flow security in Haskell. The first proposed library supports multithreaded code and evaluates the implementations of some of the ideas described above to avoid internal timing leaks. This implementation includes an online-shopping case study. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice and shows how the library can help avoiding internal timing leaks. Up to the publication date, this is the first tool that provides information-flow security in multithreaded programs and the first implementation of a case study that involves concurrency and information-flow policies. The second library, in constrast, is designed for sequential programs and includes a novel treatment for inteded release of information (declassification).

This thesis is based on the work contained in the following papers:

1) The paper *Securing Interaction between Threads and the Scheduler*, with Andrei Sabelfeld. In the Special Issue of Journal of Logic and Algebraic Programming dedicated to the Nordic Workshop on Programming Theory (NWPT'07), Elsevier Editorial. This chapter is an extension of the paper *Securing Interaction between Threads and the Scheduler*, with Andrei Sabelfeld. In Proceedings of the 19th IEEE Computer Security Foundations Workshop, Venice, Italy, July 5-7, 2006.
2) An extended version of *Security of Multithreaded Programs by Compilation*, with Gilles Barth, Tamara Rezk, and Andrei Sabelfeld. In Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS), Dresden, Germany, September 24-26, 2007, LNCS, Springer-Verlag, September 2007.
3) An extended version of the paper *Security for Multithreaded Programs under Cooperative Scheduling*, with Andrei Sabelfeld. In Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia, June 27-30, 2006. LNCS, Springer-Verlag.
4) *Closing Internal Timing Channels by Transformation*, with Andrei Sabelfeld, John Hughes, and David Naumann. In Proceedings of the 11th Annual Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006. LNCS, Springer-Verlag.
5) *A Library for Secure Multi-threaded Information Flow in Haskell*, with Tsa-chung Tsai, and John Hughes. In Proceedings of the 20th IEEE Computer Security Foundations Symposium, Venice, Italy, July 6-8, 2007. IEEE Computer Society Press.
6) An extended version of *A Library for Light-weight Information-Flow Security in Haskell*, with Koen Claessen and John Hughes. In Proceedings of the ACM SIGPLAN 2008 Haskell Symposium, Victoria, British Columbia, Canada.

My main contributions to these papers are:

1) Formalization of the semantics and the type-system together with my co-author. I performed the formal proofs and elaboration of the motivation example presented in the paper as well as the section regarding implementation issues. Contributions to the Section 3.4 and writing of Section 9.
2) I elaborated most of Section 6. That includes formalization of an assembly concurrent language, intermediate high-level typing rules, compilation function, examples, formal proofs of lemmas and theorems described in Sections 6.2 and 6.3, which are sections that I wrote.
3) Formalization of the semantics and the type-system together with my co-author. I performed the formal proofs.
4) Formalization of the transformation. I was also involved in the formalization of the semantics. I elaborated and wrote Section 5.
5) I did some initial programming of the library. I wrote the whole paper except for Section 1. I formalized the typing-rules encoded by our library. I proposed how to preserve the sub-typing invariants required by references, the solution for concurrent settings, and the case study.
6) I did most of the programming of the library with valuable additions of my co-authors. I wrote Section 5 and 6. I considerably contributed to the writing of Sections 2, 3 and 7. I elaborated the semantics, typing rules, and proofs of lemmas and theorems.

Contents

CHAPTER
1

Introduction

Computer systems are nowadays involved in most of the activities performed by our modern society. For instance, companies, banks, and governments heavily rely on computers for their everyday tasks. In fact, these activities usually demands moving, and possibly exchanging, information between components of the same or different computing systems. In some scenarios, when information flows from one place to another, confidentiality of data is an issue difficult to neglect. For example, after sending our credit card number to a web site to perform some shopping, such number must not be publicly available or accessible to unauthorized entities. Likewise, computer frequently download programs that are obtained from different sources (Internet, Bloothoot, USB sticks, etc) with no guarantees that our confidential information is preserved as such when running those programs. For these reasons, it is important that software manufactures consider security aspects when designing software as well as mechanism to enforce them. So far, some solutions to security problems have been provided, e.g anti-virus programs, network firewalls, program monitors, cryptographic techniques, intrusion detection systems, and access control mechanisms. However, they are still unable to enforce *end-to-end* [SRC84] security policies as confidentiality of data.

1 Confidentiality of data

Security requirements are often represented as *security policies*. These policies describe what are acceptable behaviors of computer systems. Confidentiality of data can be seen as a particular kind of such policies. *Information-flow policies*, a particular kind of *confidentiality policies*, describe how data is propagated once access is granted.
Information-flow policies can be formalized by attaching security levels to computational entities and data in the system, and defining how information can flow between different security levels. For instance, it possible to define the following information-

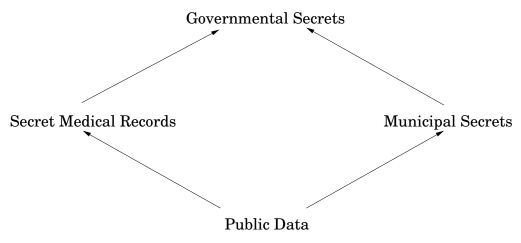

Fig. 1. Security lattice

flow policy: no object can read data from a higher security level and no data can be written in an object with lower security level. These two conditions are known as *"no reads up"* and *"no writes down"*, respectively [BL73]. To formalize this policy, a *lattice on security levels* is used [Den76]. This lattice defines what are the valid flows of information between different security levels. The ordering relation in the lattice, written $\sqsubseteq$, represents the allowed flows of information. In general, $l_1 \sqsubseteq l_2$ indicates that information of security level l_1 can flow into entities of security level l_2. Figure 1 shows an example of a security lattice with four elements: Governmental Secrets, Secret Medical Records, Municipal Secrets, and Public Data, where Public Data $\sqsubseteq$ Secret Medical Records, Public Data $\sqsubseteq$ Municipal Secrets, Secret Medical Records $\sqsubseteq$ Governmental Secrets, and Municipal Secret $\sqsubseteq$ Governmental Secrets. The information can only flow into higher positions in the lattice. In some cases, it is necessary to downgrade some information regarding secrets. *Declassification policies* expresses downgrading of information in a controlled manner and they are currently subject of active research [SS05].

2 Language-based information-flow security

Information-flow analysis studies whether an attacker can obtain confidential information by observing how the input of a system affect its output. Information can be disclosed by different mechanisms or channels. This thesis follows the line of *language-based information-flow security* [SM03]. Analysis regarding how information flows inside of programs is usually performed statically before programs are run. This analysis mainly inspects the code of programs in order to determine, or guarantee, that *end-to-end* securities policies, as confidentiality, are fulfilled.
Confidentiality policies could be precisely characterized by using program semantics. Moreover, they can be provably enforced by traditional mechanisms as type systems. *Non-interference* is a well known end-to-end property to ensure that confidentiality of data is preserved by programs. The property establishes that a variation in the confidential input of a program does not produce any variation of its public output. The attacker model defines what the attacker can observe about the execution of programs. For the non-interference property, the attacker can only inspect the public input and output states. Formally, a program starts in an input state $s = (s_h, s_l)$, where s_h and s_l are respectively the values of secret and public variables initialized with some values. If a program terminates, it results in an output state $s' = (s'_h, s'_l)$, where s'_h and s'_l are

final values for secret and public variables, respectively. The semantics of the program, written $[\![C]\!]$, is a function $[\![C]\!] : S \to S \cup \{\bot\}$ that maps input to output states or input states to $\bot$ for non-terminating programs. Variations in the input can be captured by the equivalence relation $=_L$. Two states are low-equivalent, written $s =_L s'$, iff their public values are the same, i.e., $s_l = s'_l$. The notion of non-interference can then be expressed as:

$$\forall s_1, s_2 \in S. s_1 =_L s_2 \ \wedge [\![C]\!]s_1 \neq\bot \wedge [\![C]\!]s_2 \neq\bot \Rightarrow [\![C]\!]s_1 =_L [\![C]\!]s_2 \qquad (1)$$

The definition above ignores non-terminating executions of programs. This kind of definition is known as a *termination-insensitive* security specification. In some cases, attackers can still deduce confidential information by just observing if a program terminates or not. To consider this kind of leaks due to termination, the definition of non-interference can be extended as follows:

$$\forall s_1, s_2 \in S. s_1 =_L s_2 \Rightarrow [\![C]\!]s_1 =_L [\![C]\!]s_2 \vee ([\![C]\!]s_1 =\bot \ \wedge\ [\![C]\!]s_2 =\bot) \qquad (2)$$

Observe that either both executions of C diverge or terminate with the same public output. Security conditions that take into account leaks due to termination are called *termination-sensitive* security specifications. Definitions 1 and 2 are respectively referred as *termination-insensitive* and *termination-sensitive* non-interference properties.

2.1 Types of flows

Language-based information-flow techniques deal with mechanisms used by programming languages to convey information. These mechanisms usually include assignments and branching instructions. Confidentiality of data can be preserved if programs are free of illegal *explicit and implicit* flows [DD77]. On one hand, explicit flows can leak information by assigning confidential values to public variables. For instance, the program $l := h$ leaks the secret value of h by assigning it directly to the public variable l. Implicit flows, on the other hand, use control constructs in the language to leak information. As an example, the program

$$\texttt{if}\ h > 0\ \texttt{then}\ l := 1\ \texttt{else}\ l := 2 \qquad (3)$$

leaks if $h > 0$ or not by using the construct if-then-else. Even though there is no direct assignment of secret values into public variables, the final value of l depends on the secret value h. Indeed, leaks through implicit flows can be magnified in order to leak whole values of secrets. To illustrate that, we show the following example

```
$b := 0$ ;
$l := 0$ ;
while $b < 32$ do
    if not($h$ mod $2 = 0$) then $l := l + 2^b$ else skip ;      (4)
    $b := b + 1$ ;
    $h := h$ div $2$;
```

where h is the only secret value in the program, which is internally represented as a 32-bits integer. Functions `mod` and `div` represent module and division operations between integers. Intuitively, the program goes into a loop that inspects each bit of h and leaks, through an implicit flows, such a bit. Although explicit flows are not present in the program, the whole value of h has been copied into l.

pc	**instruction**
public	`if` h > 0
secret	`then` $l := 1$;
secret	`else` $l := 2$;
public	$l' := 1$;

Fig. 2. Detection of illegal implicit flows

Prevention of illegal explicit and implicit flows On one hand, explicit flows can be easily prevented by simply enforcing the "*no reads up*" policy. One way to do that is by associating a security level to each variable and, for each assignment of the form $v := e$, it is established that the target variable (at the left hand side of the assignment) should have security level greater than any security level associated with variables appearing in e. Implicit flows, on the other hand, use the fact that the execution of some instructions depends on some secret values, which makes their detection more complex. In program (3), for instance, the execution of instructions $l := 1$ and $l := 2$ depends on the value of h. To deal with these kinds of flows, each instruction in the program is associated with the highest security level on which the execution of that instruction depends on. This security level is usually referred as *pc* since it models the security level of the *program counter*. Implicit flows are then avoided by enforcing the security level of updated variables to be not lower than *pc*, which can be seen as enforcing the "*no writes down*" policy. To illustrate the use of *pc*, we present an example in Figure 2 similar to (3) but with the addition of the public variable l' and the security level of the program counter in each line. The initial security level for the *pc* is *public*. Then, since the conditional depends on h, the *pc* is raised to the security level *secret* for every instruction inside of the construct if-then-else. The *pc* is set back to *public* for the last line in the program since it is executed regardless the value of h. An information-flow analyzer should then reject the program in Figure 2 because public variable l is updated under a *pc* with security level *secret*.

$$C ::= \texttt{skip} \mid x := e \mid C; C \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ e\ \texttt{do}\ C$$

Fig. 3. Command syntax

$$\frac{}{\vdash e : high} \qquad \frac{h \notin \mathit{Vars}(e)}{\vdash e : low} \qquad \frac{}{pc \vdash \texttt{skip}} \qquad \frac{}{pc \vdash h := e} \qquad \frac{\vdash e : low}{low \vdash l := e}$$

$$\frac{\vdash e : pc \qquad pc \vdash C_1 \qquad pc \vdash C_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2} \qquad \frac{\vdash e : pc \qquad pc \vdash C}{pc \vdash \texttt{while}\ e\ \texttt{do}\ C} \qquad \frac{pc \vdash C_1 \qquad pc \vdash C_2}{pc \vdash C_1; C_2}$$

$$\frac{high \vdash C}{low \vdash C}$$

Fig. 4. Security type system

The analysis described above can be implemented using a static type system [VSI96]. To demonstrate that, we start by considering the simple imperative language presented in Figure 3. This language has skip, assignment, sequential composition, conditional, and while-loop constructs. As with the examples given so far, we assume, for simplicity, only two levels of confidentiality: public (low) and secret ($high$). More formally, these two security levels form a two-point lattice where $low \sqsubseteq high$ and $high \not\sqsubseteq low$. As before, we write h and l for variables storing secret and public information, respectively. We assume that expressions e are formed by applying total arithmetic operations to constants and variables. Adapted from [SM04], Figure 4 presents a type system to enforce the non-interference policy (the typing rules are, indeed, equivalent to the ones found in [VSI96]). Expressions type and *pc* can be either low or $high$. The typing rules for expressions establish that any expression (including h and l itself) can have type $high$. Expressions e of type low must not contain any occurrence of h in it, which is captured by the condition $h \notin \mathit{Vars}(e)$. Commands `skip` and $h := e$ are typable under any *pc*. Command $l := e$ is typable when expression e and *pc* have type low. Demanding that e has type low enforces the "*no reads up*" policy, while having a *pc* that is low enforces "*no writes up*". In fact, $[high] \vdash C$ ensures that there are no assignments to public variables in C, which justifies that the rules for if-then-else and loops establish the *pc* as the security level of the guards. Consequently, if an if-then-else (or loop) has a guard involving h, then the *pc* is set to $high$ for the commands involving in such control construct. The rule for sequential composition preserves the *pc*. The last rule is known as *subsumption* rule and establishes that if a command is typable with *pc* as $high$, it is also typable with *pc* as low. This rule allows to reset the program counter to low after control constructs whose guards involve h.
The next theorem connects semantics of programs and the described type system in order to show that well typed programs satisfy the *termination-insensitive* non-interference property.

Theorem 1. *For any program C, memories $s_1, s_2 \in S$ such that $low \vdash C$, $s_1 =_L s_2$, $[\![C]\!]s_1 \neq \perp$, and $[\![C]\!]s_2 \neq \perp$, then $[\![C]\!]s_1 =_L [\![C]\!]s_2$.*

Attackers can still deduce information about secrets through observing the (non) termination of programs. Well typed programs might include loops whose guards involve secrets. Clearly, these kind of loops might introduce the possibility for having non-terminating executions depending on secret data and thus producing leaks. However, the typing rule for loops in Figure 4 can be modified to exclude such loops as follows.

```
b := 0 ;
while b < 32 do
    if not(h mod 2 = 0) then sleep(2) else skip ;
    b := b + 1 ;
    h := h div 2;
sleep(1);
print(".");
```

Fig. 5. External timing leaks

$$\frac{\vdash e : \mathit{low} \qquad \mathit{low} \vdash C}{\mathit{low} \vdash \texttt{while}\ e\ \texttt{do}\ C}$$

With this modification in mind, we can prove the *termination-sensitive* non-interference property for well typed programs.

Theorem 2. *For any program C, memories $s_1, s_2 \in S$ such that $\mathit{low} \vdash C$ and $s_1 =_L s_2$, then $[\![C]\!]s_1 =_L [\![C]\!]s_2$ or $[\![C]\!]s_1 = \bot$ and $[\![C]\!]s_2 = \bot$.*

3 Covert channels

Besides explicit and implicit flows, programming languages can present other mechanisms to leak information that were not originally designed for that purpose. These kinds of mechanisms are referred as *covert channels* [Lam73]. Which covert channels are a concern depends on what attackers can observe from programs. For example, smartcards are commonly inserted into untrusted terminals. Terminals provide the energy requirements to perform computations inside of the card. Therefore, it might be possible for such terminals to infer some information about smartcards by monitoring the supplied power, which clearly constitutes a covert channel [MDS99]. Other examples regarding covert channels are described below.

3.1 External timing

For this covert channel, we assume that attackers can observe the timing behavior of programs by using an arbitrarily precise stopwatch. To illustrate how this channel can be exploited, we provide a program in Figure 5, where variable h stores the only secret value and there are no public variables. Function `sleep(n)` makes the program sleep for n seconds. Observe that the program does not have any explicit or implicit flows. The program essentially travels bit-by-bit the binary representation of the value stored in h and prints a dot on the screen. Then, the output of the program is unsurprisingly 32 dots and each dot is associated to a bit of the secret. Although the output of the program is always the same, the attacker can measure, with an stopwatch, how many seconds have been elapsed between each printed dot and thus deduce the value of each bit of the secret. Call to `sleep(1)` is performed to facilitate the observation when the analyzed bit is zero. It is worth to remark that function `sleep` is not essential to produce these kinds of leaks. In fact, it is enough to replace sleep(2) and sleep(1) with a sequence of instructions C_a and C_b such that C_a and C_b takes approximately one and two seconds to run, respectively.

Transforming out timing leaks Agat [Aga00a] proposes a code transformation that pads programs with dummy computations in order to close timing leaks. The transformation can be formalized as a type system of the form: $C \hookrightarrow C' : S_l$ where C is the original program, C' is the result of the transformation, and S_l is the *low slice* of C'. The low slice S_l is different from C' in that commands of C' involving secret variables are replaced by dummy commands. The transformation guarantees that $\forall s_1, s_2 \in S. [\![C']\!] s_1 =_L [\![S_l]\!] s_2$ and that commands C' and S_l last the same amount of time when executed. The core rule for the transformation is the following:

$$\frac{C_1 \hookrightarrow C_1' : S_{l_1} \qquad C_2 \hookrightarrow C_2' : S_{l_2} \qquad \vdash e : \mathit{high} \qquad \mathit{al}(S_{l_1}) = \mathit{al}(S_{l_1}) = \mathit{false}}{\texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ C_1'; S_{l_2}\ \texttt{else}\ S_{l_1}; C_2' : \texttt{if-skip}(S_{l_1}; S_{l_2})}$$

Predicate $al(C)$ is true iff there are assignments to public variables. Since assignments to public variables present in C_i are also present in S_{l_i}, this predicate helps to avoid implicit flows. Command if$-$skip(C) acts timing-wise as an if-then-else but only having branch C to execute. The rule performs cross-coping of the low slices in order to balance out the execution time of both branches. This approach has been adapted for transforming out timing leaks in languages with concurrency [SS00] and with features as semaphores or message passing [Sab01, SM02].

3.2 Internal timing

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

$$C_1 : \ h := 0;\ l := h \qquad\qquad C_2 : \ h := \mathit{secret}$$

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. These kinds of leaks can be prevented by just avoiding explicit flows in concurrent programs as shown in Figure 4. Unfortunately, there are other covert channels present in concurrent programs.
The *internal timing* covert channel reveals information about secrets by affecting, depending on secrets, the timing behavior of threads that may affect—via the scheduler—the interleaving of assignments to public variables. For instance, consider another pair of thread commands:

$$\begin{array}{l} C_1 : \ (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep(100)}\ \texttt{else}\ \texttt{skip});\ l := 1 \\ C_2 : \ \texttt{sleep(50)};\ l := 0 \end{array} \qquad (5)$$

These threads are clearly secure in isolation because 1 is always the outcome for l in C_1, and 0 is always the outcome for l in C_2. However, when C_1 and C_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. Observe that it is the interleaving of the threads that introduces leaks. To illustrate that, we assume a scheduler that picks thread C_1 first and then proceeds to run a thread for 70 steps before giving the control to another one. If $h > 0$ then C_1 will run for 70 steps and, while being in the middle of $\texttt{sleep}(100)$, the control will be given to thread C_2, which will run till completion. The scheduler will schedule C_1 again, and C_1 will finish its execution. The final value of l is clearly 1. On the other hand, if $h \leq 0$, C_1 will finish within 70 steps and the control will be then given to C_2, which will finish its execution. The final value of l in this case is 0, which demonstrates that the program is insecure. Differently from *external* timing leaks, this covert channels can be exploited by less powerful attackers since it is not necessary for them to have a stopwatch to deduce secret information.

Primitive protect Existing approaches to specifying and enforcing information-flow security that deal with internal timing leaks often present non-standard semantics, lack of compositionality, inability to handle dynamic threads, scheduler dependence, and efficiency overhead for code that results from security-enforcing transformations. These drawbacks arise from features of the proposed approaches or by the fact that they consider more powerful attackers, i.e. with stopwatches. Particularly, Volpano and Smith propose a special primitive called `protect` in order to remove internal timing leaks [VS99]. This can be applied to any command that contains no loops. A protected command $\texttt{protect}(c)$ is executed atomically, *by definition* of its semantics. Such a primitive can be used to secure program $C_1 \parallel C_2$ as:

$$\begin{aligned} C_1 :\ & \texttt{protect}(\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\\ & l := 1\\ C_2 :\ & \texttt{sleep}(50);\ l := 0 \end{aligned}$$

Internal timing leaks are removed if every computation that branches on secrets is wrapped by $\texttt{protect}()$ commands. The timing difference is then not visible to the scheduler because of the atomic semantics of `protect`. The `protect` primitive is, however, non-standard. 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 $C_1 \parallel C_2$, a possible implementation of `protect` could attempt locking all other threads while execution is inside of the `if` statement. 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 C_2 can still be scheduled, which might result in blocking and updating the waiting list for the lock related with C_2.

```
function $leak\_bit$(integer $h$)
array $a$[SIZE];
$t$ := clock();
for $j = 0$ to REPEAT
do $h_1 := 0$;
   for $i = 0$ to SIZE $- 1$
   do $h_1 := h_1 + h$;
      $h_2 := a[h_1]$;
   end
end
return (clock() $- t$);
```

```
function $leak$(integer $h$)
for $i = 0$ to $31$
do $t_0 := leak\_bit(0)$;
   $t_1 := leak\_bit(1)$;
   $t_b := leak\_bit(h$ mod $2)$;
   if $(t_b + t_b > t_0 + t_1)$ then $l := l + 2^i$;
   $h := h$ div $2$;
end
return $l$;
```

Fig. 6. Cache attacks

3.3 Cache attacks

As a special case of timing covert channels, the presence of cache might affect the timing behavior of programs. It is then possible to exploit that in order to leak secrets. This covert channel has been previously noticed in the setting of operating systems [Vle90]. Cache attacks are feasible for attacker with or without stopwatches. In other words, it is possible to write programs that leak information through external or internal timing covert channels by exploiting how instructions or data are cached.

Function $leak\_bit$ in Figure 6 shows a data cache attack that reveals one bit of the argument by an external timing leak. We assume that the argument of the function, h, is either 0 or 1. Function clock() returns the actual time and provides attackers with a stopwatch. Constant SIZE depends on the size of the data cache of the underlying computer architecture. This constant should be bigger than the size of the cache times the number of bits used to represent integers [1]. The attack essentially runs, inside of a for-loop, an instruction that refers to the same data SIZE times if h is 0. Otherwise, the loop runs instructions which refer to SIZE different data locations in memory. In order to do that, variable h_1 acquires the value 0 in each iteration of the loop when h is 0 and values between 0 and SIZE-1 otherwise. Consequently, instruction $h_2 := a[h_1]$ would always run $h_2 := a[0]$ when h is 0 and $h_2 := a[0]$, $h_2 := a[1], \ldots, h_2 := a[\texttt{SIZE} - 1]$, otherwise. It is possible to deduce the value of h by inspecting the time that takes for the loop to run. Constant REPEAT just repeats the loop where the cache attack is performed in order to improve precision when measuring time [2].

The attack presented in function $leak\_bit$ can be magnified in order to leak, for example, whole secrets rather than just one bit. Function $leak$ in Figure 6 takes a 32-bit integer and leaks every bit of it. The function firstly calls $leak\_bit$ with arguments 1 and 0 in order to determine how much time this function takes for each argument, which are represented by the variable t_0 and t_1, respectively. Then, it again calls function $leak\_bit$

[1] For an AMD 64 3200+ processor running on a Linux machine, the attack succeeds for SIZE $=$ 5000000 and REPEAT $= 10$.

[2] For an AMD 64 3200+ processor running on a Linux machine, the attack succeeds for REPEAT $= 10$.

but with the actual secret h. The leak is produced by comparing t_0 and t_1 with the time that $leak\_bit$ takes to return a value when applied to h.
The presence of instruction cache provides another covert channels to leak secrets. It is possible to reveal information by executing different parts of a program depending on some secret data. To demonstrate this, we consider the following piece of code.

```
call $huge\_method_1()$;
if $h > 0$ then call $huge\_method_2()$; else skip ;
call $huge\_method_1()$;
```

We assume that methods $huge\_method$ and $huge\_method_2$ contains enough instructions to fill up the instruction cache in the computer system. Then, if $h \leq 0$, the second call to $huge\_method_1$ will execute faster than the first one since the instruction are cached. However, if $h > 0$, instructions stored in the cache will be replace by instruction corresponding to $huge\_method_2$, and therefore the second call to $huge\_method_1$ will run at approximately the same speed as the first call to that method. This attack can easily be magnified in order to leak whole secret values. Agat suggests to run interpreted programs instead of compiled ones in order to deal with this covert channel even though it affects performance [Aga00b]. By doing so, programs are now data for the interpreter and instruction cache attacks are lifted to data cache attacks.
The presence of cache can also be exploited to leak secrets through internal timing leaks. To illustrate that, consider the function $leak\_bit'$, which contains the same code as $leak\_bit$ except for those lines where function $clock()$ is called. We rewrite the attack shown in (5) as follows.

$$C_1 : (\texttt{if}\ h > 0\ \texttt{then leak\_bit}'(\texttt{1})\ \texttt{else skip});\ l := 1$$
$$C_2 : \texttt{leak\_bit}'(\texttt{0});\ l := 0$$

As in (5), there are differences in the timing behavior of the treads depending on the secret. The race to assign the public variable l is determined by h. Observe that function $leak\_bit'$ takes longer when applied to 1 than to 0.

Security via low determinism In general, approaches considering timing leaks usually model units of time as reduction steps in the semantics [SV98, VS98, Aga00a, SS00, Sab01, MS01, RS06, RHNS06]. However, one unit of time might not correspond to one reduction step in the semantics when running programs. One reason for that is the presence of instruction cache. For instance, depending on the state of the cache, the number of instructions run by unit of time might change. This subtle difference between modeling time in the semantics and real computers with cache might compromise confidentiality of data by performing cache attacks.
Inspired by Roscoe's *low-view determinism* [Ros95] for security in a CSP setting, Zdancewic and Myers [ZM03] develop an approach to information flow in concurrent systems. According to this approach, a program is secure if its publicly-observably results are deterministic and unchanged regardless of secret inputs. This avoids refinement attacks from the outset. Specifically, this definition rules out internal timing leaks

```
$auth_l := False;$
$username_l := input();$
$password_l := input();$
$secret_h := get\_password(username_l);$
if $(secret_h == username_l)$ then $auth := True;$ else $auth := False$ ;
```

Fig. 7. Code for authentication

and cache attacks. However, low-view determinism security rejects intuitively secure programs (such as $l := 0 \parallel l := 1$), and thus introduces the risk of rejecting useful programs. Analysis enforcing low-view determinism are inherently noncompositional since the parallel composition with a thread assigning to low variables is not generally secure. Huisman et al. [HWS06] have suggested a temporal logic-based characterization of low-view determinism security. This characterization enables high-precision security enforcement by known model-checking techniques.

Boudol and Castellani [BC01, BC02] propose a type system that rejects assignment to public variables after branching on secret. Although restrictive, this approach rules out internal timing leaks as well as cache attacks.

4 Declassification

Non-interference is a security policy that specifies the absence of information flows from secret to public data. However, some applications release some information as part of their intended behavior. Consider, for example, the code in Figure 7. It implements a very simple routine for authentication. The security level for each variable is indicated with the subindexes l and h for public and secret data, respectively. Variable $access_l$ determines if the user is successfully authenticated. Function `input` asks the user for an input string. Variables $username_l$ and $password_l$ store the username and password provided by the user. Function $get\_password$ retrieves the password for a given username. Variable $secret_h$ stores the password of the user. The program contains an illegal implicit flow and therefore it does not satisfy non-interference. Depending on $secret_h$, the public variable $auth_l$ is assigned to $True$ or $False$. Indeed, the program is leaking information! Non-interference does not provide means to distinguish between intended releases of information and those ones produced by malicious code, programming errors, or vulnerability attacks. It is needed to relax the notion of non-interference to consider *declassification* policies or intended ways to leak information.

Declassification policies have been recently classified in different dimensions [SS05]. Each dimension represents aspects of declassification. Aspects correspond to *what*, *when*, *where*, and by *whom* data is released. In general, type-systems to enforce different declassification policies include different features, e.g rewriting rules, type and effects, and external analysis [ML00, SM04, CM04].

In the following sections, we briefly illustrate one approach for each dimension of declassification. For further information, readers can refer to [SS05].

`if` $(h = 0)$ `then` $l := 1$ `else` $l := 2$

Fig. 8. Secure program

`if` $(h < 0)$ `then` $l := 1$ `else` $l := 2$

Fig. 9. Unsecure program

4.1 What dimension

In a series of papers[CHM04, CHM05a, CHM05b], Clark et. al. develop a quantitative approach to information flow security. Based on information theory [SW63], the authors introduce definitions to capture the amount of information leaked by a variable inside of deterministic programs. These definitions are based on the notion of *entropy*. Before exposing the insights of this approach, we start by reviewing some basic concepts of information theory relevant for this section.
A random variable X is a total function $X : D \to R$, where D and R are finite sets and D is equipped with a probability distribution. We note $R(X)$ to the range over the set of values which X may take. We note $p(x)$ to the probability that X acquires value x. The entropy of a random variable X is denoted by $H(X)$ and is defined as follows:

$$\mathcal{H}(X) = - \sum_{x \in R(X)} p(x) \log(\frac{1}{p(x)})$$

The base of $\log$ can be chosen freely but it is conventional to use base 2. If $p(x) = 0$ then $p(x) \log(\frac{1}{p(x)})$ is defined as 0. Intuitively, an event that occurs with a non-zero probability p has an uncertainty, or surprise value, of $\log(\frac{1}{p})$. Observe that uncertainty is inversely proportional to likelihood.
Conditional entropy, written $\mathcal{H}(X|Y)$, measures the uncertainty in X given knowledge Y, where X and Y are random variables. It is defined as follows.

$$\mathcal{H}(X|Y) = \mathcal{H}(X, Y) - \mathcal{H}(Y)$$

Random variables can be used to represent variables at certain program points. It is then possible to reason about probabilities that variables acquire certain values at certain points of the programs. Given a program variable X (or set of program variables), we respectively write X^i and X^o as the random variables corresponding to the initial and final values of X. We define L^i and L^o as the random variables corresponding to the initial and final values of public variables. Random variable H^i corresponds to the initial values of variables storing secret data. Finally, we define the amount of information revealed by variable X, written $\mathcal{L}(X)$, as follows.

$$\mathcal{L}(X) = \mathcal{H}(H^i|L^i) - \mathcal{H}(H^i|X^o, L^i)$$

Expression $\mathcal{H}(H^i|L^i)$ captures the uncertainty of knowing secret input by knowing public inputs. Similarly, expression $\mathcal{H}(H^i|X^o, L^i)$ captures the uncertainty of knowing secret inputs by knowing public inputs and the final value of X. The difference between these two quantities reveals how much information is leaked by observing the final value of X. In this setting, programs are defined as non-interferent iff $\mathcal{L}(L) = 0$.

```
aBid := getEnvelopA() ;
bBid := getEnvelopB() ;
print("The winner is:") ;
if (aBid ≥ bBid) then print("A");
                 else print("B");
```

Fig. 10. Sealed auction example

```
aBid := getEnvelopA() ;
bBid := aBid + 1 ;
print("The winner is:") ;
if (aBid ≥ bBid) then print("A");
                 else print("B");
```

Fig. 11. Unsecure sealed auction example

To illustrate how the definition of $\mathcal{L}$ works with programs, we borrow an example by Clark et. al. [CHM04]. We assume a scenario where the secret is a 4-bit number, stored in variable h, and the information to be declassified (or revealed intentionally) from h is only the absolute value of it, which is stored in the public variable l. So, being in a state where $l = \texttt{abs}(h)$, we have the program in Figure 8. At first glance, it seems to be an illegal implicit flow. However, to see if the program reveals more than the absolute value of h, we apply definition $\mathcal{L}(l)$ at this point of the program. Assuming a uniform distribution for h, we have that $\mathcal{L}(l) = \mathcal{H}(h^i|l^i) - \mathcal{H}(h^i|l^o, l^i) = 0$. This result indicates that the program does not reveal anything new. Observe that the program reveals if h is zero or not, which is already revealed when declassifying the absolute value of h. In contrast, if we have the program shown in Figure 9, we obtain that $\mathcal{L}(l) = 1$, which clearly indicates that the program reveals one more bit than what is allowed to be declassified. In fact, the program reveals the sign of h. The authors recently develop a syntax directed type system that safely approximates the amount of information leak in programs [CHM07].

4.2 When dimension

As a motivating example for handling this dimension, we can consider the scenario described in [CM04] of a sealed auction where each bidder submits a single secret bid in a sealed envelop. Once all bids are submitted, the envelopes are opened and the bids are compared. The highest bidder wins. One security policy that is important for this program is that no bidder knows any of the other bids until all the bids have been submitted. Program in Figure 10 simulates this process for two bidders: A and B. Functions *getEnvelopA* and *getEnvolepB* obtain the bids corresponding to users A and B, respectively. It is possible to incorrectly implement the auction protocol by mistake or intentionally. For instance, in Figure 11, we present a program that does not fulfill the mentioned security policy. This program inspects user A's bid ($bBid := aBid + 1$) before user B submits his (her) own in order to make user B the winner.
Broberg and Sands [BS06] introduce some sort of boolean flags called *flow locks* that determine when information can flow between variables. The idea is that storage locations, written ℓ, are guarded by flow locks which represent policies related to events or conditions that must be satisfied in order to have access to the data in ℓ. For instance, $\ell_{A \wedge submittedB \Rightarrow B}$ [3] indicates that the value stored in ℓ can be read by the principal A and

[3] The notation here differs from the one presented in [BS06] in the sense that a sequence of flow lock policies are separated by semicolon instead of the logical operator $\wedge$.

```
aBid := getEnvelopA() ;
bBid := getEnvelopB() ;
print("The winner is:") ;
if (aBid ≥ bBid) then print("A");
                 else print("B");
```

```
function getEnvelopA()
bid_{A∧submittedB⇒B} := readChanA() ;
open submittedA ;
return bid;

function getEnvelopB()
bid_{B∧submittedA⇒A} := readChanB() ;
open submittedB ;
return bid;
```

Fig. 12. Secure sealed auction example

by the principal B provided that the flow lock $submittedB$ is open. This simple mechanism is able to represent a number of recently proposed information flow paradigms for declassification. Instructions `open` and `close` are provided in order to open and close flow locks. Programs do not depend on the lock state and a type-system is provided to statically check that policies specified by flow locks are fulfilled.

In Figure 12, we show a secure implementation of the auction system with flow locks where functions $getEnvelopA$ and $getEnvelopB$ are slightly changed. The system involves principals A and B, which represent the users involved in the auction. In function $getEnvelopA$, variable bid is annotated with $A \wedge submittedB \Rightarrow B$, which makes user B able to read user A's bid as long as $submittedB$ is open. In function $getEnvelopB$, instruction `open` $submittedB$ opens the lock after user B's bid is obtained. Similarly, in function $getEnvelopB$, variable bid is annotated with $B \wedge submittedA \Rightarrow A$ and $submittedA$ is open after user A's bid is obtained. As a result, values stored in $aBid$ and $bBid$ are accessible only after both users have submitted their bids. In fact, program in Figure 11 is rejected by Broberg and Sand's type system since instruction $bBid := aBid + 1$ tries to access the value stored in $aBid$ when the flow lock $submittedB$ is closed.

To illustrate why flow locks may need to be closed, we take the example one step further by thinking of a bidding system that allows users to bid more than once. In this case, functions $getEnvelopA$ and $getEnvelopB$ are called several times inside of a loop and flow locks related to $aBid$ and $bBid$ must be closed between each iteration. Otherwise, all the flow locks are open at the second call of those functions, which allows bids to be released at any time. It is not difficult to imagine this implementation by considering executing instructions `close` $submittedA$ and `close` $submittedB$ in between each iteration of the loop.

It is still possible to write programs that wrongly implement the auction system. For instance, we can write a program that makes user A the winner all the time by just replacing the `if-then-else` in Figure 12 by `print("A")`. However, user A is going to be the winner because the program is not implemented correctly, but not because it does not fulfill the security policies specified by the flow locks. Correctness of programs are stronger properties than those ones captured by using flow locks.

4.3 Who dimension

In the *Decentralized Label Model* (DLM) [ML97, ML98, ML00] data is labeled with a set of principals who owns the information and indicate who can read such data. Labeling of data in this manner makes the approach particularly suitable for scenarios of mutual distrust. While executing a program, the code is also authorized to act on behalf of some set of principals known as *authority*. An example of a label is $\{u_o : u_1, u_2\}$, which indicates a security policy where principals u_o, u_1, and u_2 can read the data while only u_o, the owner, can modify it. It is also possible that a piece of data is associated to several security policies. For instance, label $\{u_o : u_1, u_2 \,; u'_o : u'_1, u'_2\}$ indicates two owners (u_o and u'_o) and six readers ($u_o, u_1, u_2, u'_o, u'_1, u'_2$). The intuitive idea of the approach is that labels must be obeyed as data flows through different parts of programs. Code can access data when its authority acts for a reader of each of the security policies indicated by its label.
Data can flow from a source with security label L_1 to a sink with security label L_2 as long as L_2 is, at least, as restrictive as L_1. This relationship between labels is written as $L_1 \sqsubseteq L_2$. It is clear that label L_2 is, at least, as restrictive as L_1 if every policy in L_1 is guaranteed to be enforced by L_2. To determine that, a sound and complete relabeling rule is introduced. This rule allows to transform, or relabel, one security policy into another *more restrictive*. Then, $L_1 \sqsubseteq L_2$ iff every policy of L_1 can be relabeled into a policy of L_2. Moreover, during computation, values might be derived from different data sources. Consequently, labels must enforce the security policies for each of these sources. For instance, if we add two number, the result's label must be, at least, as restrictive as the labels of both operands. However, the label of the sum should be the *least* restrictive label having this property in order to avoid unnecessary restrictiveness. The least restrictive set of policies between two given set of policies L_1 and L_2, written $L_1 \sqcup L_2$, is just the union of them ($L_1 \cup L_2$). Having defined notions for valid flows and the least restrictive set of policies, it is then possible to statically check that programs satisfies the security policies established by labeled data.
Declassification is introduced in this framework by another relabeling rule. Essentially, declassification is performed by making a copy of the released data and marked it with the same labels as before the downgrading but excluding those ones appearing in the authority of the code. As an example, we can assume two principal for the authentication example in Figure 7: `system` and `user`. We assume that the code is run with authority `system`. We assign label $\{\texttt{user} : \texttt{user}, \texttt{system}\}$ to $username_l$ and $password_l$ and label $\{\texttt{system} : \texttt{system}\}$ to $secret_h$. According to DLM, the label of $secret_h == username_l$ is $\{\texttt{user} : \texttt{user}, \texttt{system}; \texttt{system} : \texttt{system}\}$. However, by declassifying that expression in a piece of code with authority `system`, the label is rewritten to $\{\texttt{user} : \texttt{user}, \texttt{system}\}$, which allows users to know if they have access to the system.

5 Information-flow security as a library

Language-based information-flow security aims to guarantee security policies related with confidentiality and integrity of data. It is commonly achieved by some form of static or dynamic analysis which rejects programs that would violate such policies. Over the years, a great many such systems have been presented, supporting a wide variety of programming constructs [SM03]. However, the impact on programming practice has been rather limited.

One possible reason is that most systems are presented in the context of a simple, elegant, and minimal language, with a well-defined semantics to make proofs of soundness possible. Such systems cannot immediately be adopted by programmers—they must first be embedded in a real programming language with a real compiler, which is a major task in its own right. Two of such languages have been developed—Jif [Mye99, MZZ^{+}06] (based on Java) and FlowCaml [PS02, Sim03] (based on Caml) which is unfortunately not maintained anymore.

When a system implementor chooses a programming language, information flow security is only one factor among many. While Jif or FlowCaml might offer the desired security guarantees, they may be unsuitable for other reasons, and thus not adopted. This motivated Li and Zdancewic to propose an alternative approach, whereby information flow security is provided via a *library* in an existing programming language [LZ06]. Constructing such a library is a much simpler task than designing and implementing a new programming language, and moreover leaves system implementors free to choose any language for which such a library exists.

Li and Zdancewic showed how to construct such a library for the functional programming language Haskell. The library provides an abstract type of secure programs, which are composed from underlying Haskell functions using operators that impose information-flow constraints. The abstract data type is defined as an arrow [Hug00] and arrows combinators are accordingly introduced by the library. It is then possible to combine arrow computations to create more complex functions at the same time that constrains related to information-flow policies are considered when constructing such functions. Secure programs must be *certified* before running. In order to do that, the library checks that all the collected constraints related to arrow computations are satisfied, before the underlying functions are invoked—thus guaranteeing information-flow security policies are fulfilled. While secure programs are a little more awkward to write than ordinary Haskell functions, Li and Zdancewic argue that typically only a small part of a system need manipulate secret data—for example, an authentication module—and only this part need be programmed using their library.

However, Li and Zdancewic's library does impose quite severe restrictions on what a secure program fragment may do. In particular, these fragments may have no effects of any sort, since the library only tracks information flow through the inputs and outputs of each fragment. While absence of side-effects can be guaranteed in Haskell (via the type system), this is still a strong restriction.

6 Thesis overview

This thesis mainly proposes techniques to deal with the *internal timing covert channel*. It proposes remedies for leaks produced by exploiting scheduler properties through the timing behavior of threads in order to modify how the public variables are updated. We distinguish between scenarios where it is possible (or not) to modify the scheduler in order to be able to guarantee confidentiality policies. Additionally, the dissertation

proposes techniques to provide information-flow security as a library. Figure 13 shows a road map of the thesis and describes the scenario considered for each chapter.
The thesis takes a language-based approach to information-flow enforcement. In this section, we briefly outline the contents of the six chapters.

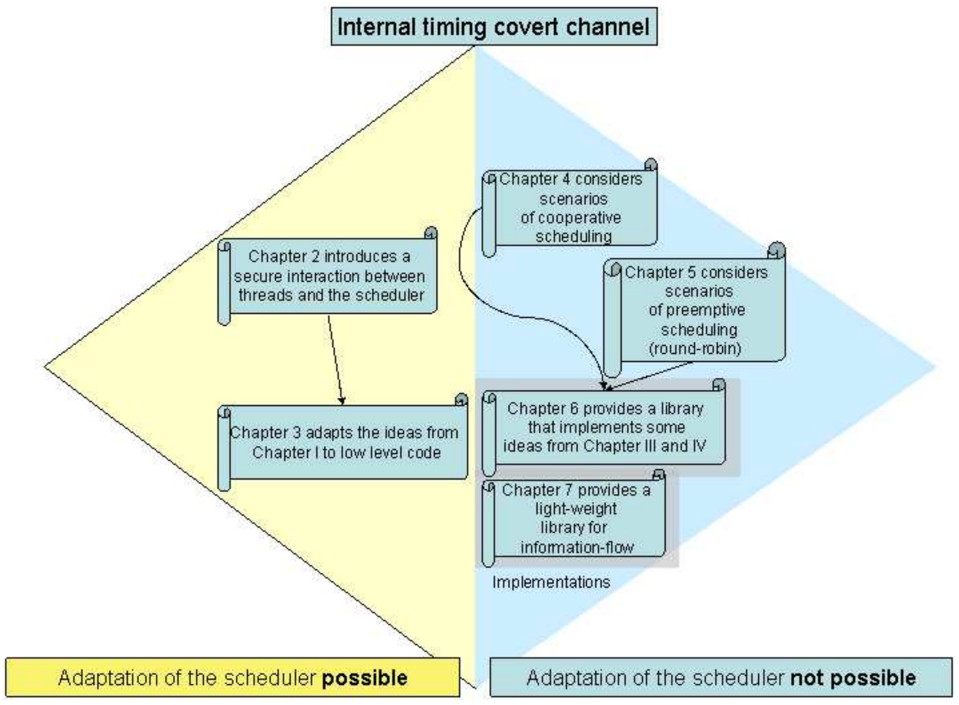
Internal timing covert channel
Chapter 2 introduces a secure interaction between threads and the scheduler
Chapter 3 adapts the ideas from Chapter I to low level code
Chapter 4 considers scenarios of cooperative scheduling
Chapter 5 considers scenarios of preemptive scheduling (round-robin)
Chapter 6 provides a library that implements some ideas from Chapter III and IV
Chapter 7 provides a light-weight library for information-flow
Implementations
Adaptation of the scheduler **possible**
Adaptation of the scheduler **not possible**

Fig. 13. Road map of the thesis

Chapter 2: Securing interaction between threads and the scheduler Existing approaches involving concurrent programs and information flow security often present non-standard semantics, lack of compositionality, inability to handle dynamic threads, scheduler dependence, and efficiency overhead for code that results from security-enforcing transformations. Particularly, Volpano and Smith propose a special primitive called `protect` in order to remove internal timing leaks. By definition, $\texttt{protect}(c)$ takes one atomic step in the semantics with the effect of executing c until termination. Internal timing leaks are removed if every computation that branches on secrets is wrapped by $\texttt{protect}()$ commands. However, implementing `protect` imposes a major challenge. This chapter suggests a remedy for some of the described shortcomings and a framework that allows the implementation of a generalized version of `protect`. More precisely, it introduces a novel treatment of the interaction between threads and the scheduler. A permissive non-interference-like security specification and a security type system that provably enforces this specification are obtained as a result of such interaction. The type system guarantees security for a wide class of schedulers and provides a flexible treatment of dynamic thread creation as well as synchronization primitives. The proposed techniques relies on the modification of the scheduler in the run-time environment.

This chapter is based on a paper accepted to the special issue of the Journal of Logic and Algebraic Programming dedicated to the Nordic Workshop on Programming Theory '07 and a paper accepted to the 18th IEEE Computer Security Foundations Workshop, Venice, Italy, July 5-7, 2006.

Chapter 3: Security of multithreaded programs by compilation Multithreaded bytecode is ubiquitous in, for instance, mobile phones scenarios. For example, multithreading is sued for preventing screen lock-up in mobile applications when sending an SMS. This chapter proposes a framework for enforcing secure information-flow for multithreaded low-level programs. The approach presents security-type systems that provably guarantee non-interference. Inspired by ideas from proof-carrying code, producers of code can derive security types for low-level programs from security types from source programs. Consumers then receive these security types and the low-level code in order to verify that the bytecode satisfies the non-interference property. In this way, our approach is particularly suitable for scenarios of untrusted mobile code. Moreover, even if the code is trusted, compilers are often too complex to be part of the trusted computing base. In this case, the type annotations of compiled programs can be checked directly at bytecode level, which removes completely compilers from the trusted base. An attractive feature of this approach is that there are no more restrictions on multithreaded source programs than on sequential ones, and yet we guarantee that their compilations are provably secure for a wide class of schedulers. This might be counterintuitive since multithreaded programs may exploit covert channels as, for instance, internal timing. Indeed, the special primitives described in Chapter I have been adapted in order to control these channels at low-level code.

This chapter is an extended version of the paper accepted to the 10th European Symposium on Research in Computer Security (ESORICS), Dresden, Germany, September 24-26, 2007

Chapter 4: Security for multithreaded programs under cooperative scheduling In some scenarios, the modification of the run-time environment might not be an acceptable requirement. In this light, this chapter presents a transformation that eliminates the need for `protect` under cooperative scheduling. In fact, no additional interactions, besides yielding control to a thread, are needed in order to avoid internal timing leaks. Variations in the transformation can enforce both termination-insensitive and termination-sensitive security specifications in a language with dynamic thread creation.

This chapter is an extended version of the paper accepted to the Andrei Ershov International Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia, June 27-30, 2006.

Chapter 5: Closing internal timing channels by transformation For those scenarios where the scheduler is preemptive and behaves as round robin, this chapter presents a transformation that closes the internal timing channel for multithreaded programs. The transformation is based on spawning dedicated threads, whenever computations may affect secrets, and carefully synchronizing them. Moreover, the transformation only rejects programs that have symptoms of illegal flows inherent from sequential settings.
This chapter has been published in the Proceedings of the 11th Annual Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006.

Chapter 6: A library for secure multi-threaded information flow in Haskell Recently, Li and Zdancewic have proposed an approach to provide information-flow security via a library rather than producing a new language from the scratch. They show how to implement such a library in Haskell. This chapter presents an extension of Li and Zdancewic's library that provides information-flow security for multithreaded programs. The extension provides reference manipulation, a run-time mechanism to close internal timing leaks, and a flexible treatment of dynamic thread creation. In order to provide such features, the library combines some ideas presented in this thesis together with some other ones taken from literature: type system with effects, singleton types, projection functions, cooperative round-robin schedulers, and type classes in Haskell. Moreover, an online-shopping case study has been implemented in order to evaluate the proposed techniques. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice and shows how the library can help to avoid internal timing leaks. Up to the publication date, this is the first implemented tool to provide information-flow security in concurrent programs and the first implementation of a case study that involves concurrency and information-flow policies.
This chapter has been published in the Proceedings of the 20th IEEE Computer Security Foundations Symposium, Venice, Italy, July 6-8, 2007.

Chapter 7: A light-weight library for information-flow security in Haskell Rather than producing a new language from scratch, information-flow security can also be provided as a library. In previous work, this has been done using the arrow framework. In this work, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve the same goals. We present a monadic library to provide information-flow security for Haskell programs. The library introduces mechanisms to protect confidentiality of data for pure computations, that we then easily, and modularly, extend to include dealing with side-effects. We also present combinators to dynamically enforce different declassification policies when released of information is required in a controlled manner. It is possible to enforce policies related to what, by whom, and when information is released or a combination of them. The well-known concept of monads together with the lightweight characteristic of our approach makes the library suitable to build applications where confidentiality of data is an issue.

This chapter is an extended version of a paper accepted to the ACM SIGPLAN 2008 Haskell Symposium, Victoria, British Columbia, Canada, September 2008.

References

[Aga00a] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

[Aga00b] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, December 2000.

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

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

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[BS06] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In Peter Sestoft, editor, *Proc. European Symp. on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

[CHM04] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference: Information theory and information flow. Presented at Workshop on Issues in the Theory of Security (WITS'04), April 2004.

[CHM05a] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112:149–166, January 2005. Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages (QAPL 2004).

[CHM05b] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language*, 18(2):181–199, 2005.

[CHM07] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

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

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

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

[Lam73] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[LZ06] P. Li and S. Zdancewic. Encoding information flow in haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of power analysis attacks on smartcards. In *WOST'99: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 17–17, Berkeley, CA, USA, 1999. USENIX Association.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MS01] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 126–142, June 2001.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

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

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[RHNS06] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. Annual Asian Computing Science Conference*, LNCS, December 2006.

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

[RS06] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[Sim03] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

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

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

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE Computer Society, 2005.

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

[SW63] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1963.

[Vle90] T. V. Vleck. Timing channels, May 1990. Poster session, IEEE TCSP conference, Oakland CA.

[VS98] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 34–43, June 1998.

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

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

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

CHAPTER

2

Securing Interaction between Threads and the Scheduler in Presence of Synchronization

Extended version of
Securing Interaction between Threads and the Scheduler,
In Proceedings of the 19th IEEE Computer Security Foundations Workshop, Venice, Italy, July 5-7, 2006. IEEE Computer Society Press.
Chapter published in the Special Issue of Journal of Logic and Algebraic Programming dedicated to the Nordic Workshop on Programming Theory (NWPT'07), Elsevier Editorial.

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

Alejandro Russo and Andrei Sabelfeld

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

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

1 Introduction

The problem of information flow in multithreaded programs remains an important open challenge [SM03]. While information flow in sequential programs is relatively well understood, information-flow security specifications and enforcement mechanisms for sequential programs do not generalize naturally to multithreaded programs [SV98]. In this light, it is hardly surprising that Jif [MZZ$^+$06] and Flow Caml [Sim03], the mainstream compilers that enforce secure information flow, lack support for multithreading. Nevertheless, the need for information flow control in multithreaded programs is pressing because concurrency and multithreading are ubiquitous in modern programming languages. Furthermore, multithreading is essential in security-critical systems because threads provide an effective mechanism for realizing the *separation-of-duties* principle [VM01].

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

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

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

This paper develops an approach that is compatible with each of these properties by a novel treatment of the interaction between threads and the scheduler. We enrich the language with primitives for raising and lowering the security levels of threads. Threads with different security levels are treated differently by the scheduler, ensuring that the interleaving of publicly-observable events may not depend on sensitive data. As a result, we present a permissive noninterference-like security specification and a compositional security type system that provably enforces this specification. The type system guarantees security for a wide class of schedulers and provides a flexible and efficiency-friendly treatment of dynamic threads.
The main novelty of this paper, compared to a previous workshop version [RS06a], is the inclusion of synchronization primitives into the underlying language.
In the rest of the paper we present background and related work (Section 2), the underlying language (Section 3), the security specification (Section 4), and the type-based analysis (Section 5). We discuss an extension to cooperative schedulers (Section 6), an example (Section 7), implementation issues (Section 8), and present an extension of the framework with synchronization primitives (Section 9), before we conclude the paper (Section 10).

2 Motivation and background

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

2.1 Leaks via scheduler

Assume a partition of variables into high (secret) and low (public). Suppose h and l are a high and a low variable, respectively. Intuitively, information flow in a program is secure (or satisfies *noninterference* [Coh78, GM82, VSI96]) if public outcomes of the program do not depend on high inputs. Typical leaks in sequential programs arise from

explicit flows (as in assignment $l := h$) and *implicit* [DD77] flows via control flow (as in conditional $\texttt{if}\ h > 0\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$).
The ability of sequential threads to share memory opens up new information channels. Consider the following thread commands:

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

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:

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

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

$$\begin{aligned} &c_1: \texttt{lock}; h := 0;\ l := h; \texttt{unlock} \\ &c_2: \texttt{lock};\ h := \mathit{secret}; \texttt{unlock} \end{aligned}$$

The lock primitives ensure that the undesired interleaving of c_1 and c_2 is prevented. Note that this solution prevents a *race condition* [SBN$^+$97] in the sense that it is now impossible for the two threads (where one of them attempts writing) to simultaneously access variable h.
Unfortunately, synchronization primitives, which are typically used for race-condition prevention (e.g., [CF07]), offer no general solution. The source of the leak in program $d_1 \parallel d_2$ is *internal timing* [VS99]. The essence of the problem is that the timing behavior of a thread may affect—via the scheduler—the interleaving of assignments. As we will see later in this section, securing interleavings from within the program (such as with synchronization primitives) is a highly delicate matter.
What is the key reason for these flows? Observe that in both cases, it is the interleaving of the threads that introduces leaks. Hence, it is the *scheduler* and its interaction with the threads that needs to be secured in order to prevent undesired information disclosure. In this paper, we suggest a treatment of schedulers that allows the programmer to ensure from within the program that undesired interleavings are prevented.
In the rest of this section, we review existing approaches to information flow in multithreaded programs that are directly related to the paper. We refer to an overview of language-based information security [SM03] for other, less related, work.

2.2 Possibilistic security

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

2.3 Scheduler-specific security

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

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

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

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

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

For simplicity, assume that $\texttt{sleep(n)}$ is an abbreviation for n consecutive `skip` commands. Consider a scheduler that picks thread d_1 first and then proceeds to run a thread for 70 steps before giving the control to the other thread. If $h > 0$ then d_1 will run for 70 steps and, while being in the middle of $\texttt{sleep(100)}$, the control will be given to thread d_2. Thread d_2 will try to acquire the lock but will block, which will result in d_2 being placed as the first thread in the wait list for the lock. The scheduler will then schedule d_1 again, and d_1 will release the lock with `unlock` and try to grab the lock with `lock`. However, it will fail because d_2 is the first in the wait list. As a result, d_1 will be put behind d_2 in the wait list. Further, d_2 will be scheduled to set l to 0, release the lock, and finish. Finally, d_1 is able to grab the lock and execute $l := 1$, release the lock, and finish. The final value of l is 1. If, on the other hand, $h \leq 0$ then, clearly, d_1 will finish within 70 steps, and the control will be then given to d_2, which will grab the lock, execute $l := 0$, release the lock, and finish. The final value of l in this case is 0, which demonstrates that the program is insecure. Generally, under many schedulers, chances for $l := 0$ in d_2 to execute before $l := 1$ in d_1 are higher if the initial value of h is positive. Thus, the above implementation fails to remove the internal timing leak.
This example illustrates the need for a tighter interaction with the scheduler. The scheduler needs to be able to suspended certain threads instantly. This flexibility motivates the introduction of the `hide` and `unhide` constructs in this paper.
Returning to probabilistic scheduler-specific noninterference, Smith has continued this line of work [Smi01] to emphasize practical enforcement. In contrast to previous work, the security type system accepts `while` loops with high guards when no assignments to low variables follow such loops. Independently, Boudol and Castellani [BC01, BC02] provide a type system of similar power and show possibilistic noninterference for typable programs. This system does not rely on `protect`-like primitives but winds up rejecting assignments to low variables that follow conditionals with high guards.
The approaches above do not handle dynamic threads. Smith [Smi03] has suggested that the language can be extended with dynamic thread creation. The extension is discussed informally, with no definition for the semantics of `fork`, the thread creation construct. A compositional typing rule for `fork` is given, which allows spawning threads under conditionals with high guards. However, the uniform scheduler assumption is critical for such a treatment (as it is also for the treatment of `while` loops). Consider the following example:

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

This program is considered secure according to [Smi03]. Suppose the scheduler happens to first execute e_3 and then schedule the first thread (e_1) if the threadpool has more than three threads and the second thread (e_2) otherwise. This results in an information leak from h to l because the size of the threadpool depends on h. Note that the above program is insecure for many other schedulers. A minor deviation from the strictly uniform probabilistic choice of threads may result in leaking information.
A possible alternative aimed at scheduler-independence is to force threads (created in branches of `if`s with high guards) along with their children to be protected, i.e., to

disable all other threads until all these threads have terminated (this can be implemented by, for example, thread priorities). Clearly, this would take a high efficiency tall on the encouraged programming practice of placing dedicated potentially time-consuming computation in separate threads. For example, creating a new thread for establishing a network connection is a much recommended pattern [Knu02, Mah04].

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

2.4 Scheduler-independent security

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

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

2.5 Security via low determinism

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

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

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

Fig. 1. Command syntax

2.6 Security in the presence of synchronization

Andrews and Reitman [AR80] propose a logic for reasoning about information flow in a language with semaphores. However, the logic comes with no soundness arguments or decision algorithms.
External timing-sensitive security has been extended to languages with semaphores primitives by Sabelfeld [Sab01] and message passing by Sabelfeld and Mantel [SM02]. Although our focus is internal timing, the semantic presentation of semaphores from the former work serves as a useful starting point for this paper.
Recently, Russo et al. [RHNS07] have proposed a transformation that closes internal timing leaks by spawning sensitive computation in dedicated threads. Semaphores play a crucial role for the synchronization of these threads. However, contrary to this work, the source language for the transformation lacks semaphores.

3 Language

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

3.1 Semantics for commands

A command c and a memory m together form a *command configuration* $\langle\!| c, m |\!\rangle$. The semantics of configurations are presented in Figure 2. A small semantic step has the form $\langle\!| c, m |\!\rangle \xrightarrow{\alpha} \langle\!| c', m' |\!\rangle$ 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 $\rightsquigarrow\!\bullet$ that communicates to the scheduler to treat the thread as high (whether or not the thread was already high). The unhiding command `unhide` has the dual effect: it communicates to the scheduler by event $\bullet\!\rightsquigarrow$ that the thread should be treated as low. To intuitively illustrate how to utilize

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

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

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

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

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

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

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

Fig. 2. Semantics for commands

```
try {
  if l1 then l2 := 1; hide; c1 else l2 := 0; hide; c2
} catch {unhide; c3}
```

Fig. 3. An `unhide` refers to several `hide`

`hide` and `unhide`, we modify the motivating example given in Section 2.1, where we wrap the branching command around `hide` and `unhide` commands as follows:

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

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

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

We define independent commands `hide` and `unhide` instead of forcing them to wrap code blocks syntactically (cf. `protect`). We expect this choice to be useful when adding exceptions to the language. For example, consider the program in Figure 3. Command `try` determines code blocks that might throw an exception, while command `catch` states exception handlers. Variables l_1,l_2, and l_3 are public. Commands c_1 and c_2 contain branches whose guards involve secrets. Command c_3 is part of the exception handler. In this program, the `unhide` command in the exception handler refers to several `hide` primitives under the `try` statement.

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

3.2 Semantics for schedulers

Figure 4 depicts the semantic rules that describe the behavior of the scheduler. A scheduler is a program σ (written in a language, not necessarily related to one from Figure 1) that, together with a memory ν, forms a *scheduler configuration* $\langle\!|\sigma, \nu|\!\rangle$. We assume that the scheduler memory is disjoint from the program memory. The scheduler memory contains variable q that regulates for how many steps a thread can be scheduled. *Live* (i.e., ready to execute) threads are tracked by variable t that consists of low and high parts. The low part is named by $t_\circ$, while the high part is composed of two subpools named $t_\bullet$ and t_e. Threads in $t_\bullet$ are always high, but threads in t_e were low in the past, are high at present, and might eventually be low in the future. Threads are moved back and forth from $t_\circ$ to t_e by executing the hiding and unhiding commands. Variable r represents the running thread. Variable s regulates whether low threads may be scheduled. When s is $\circ$, both low and high threads may be scheduled. However, when s is $\bullet$, only high threads may be scheduled, preventing low threads from observing internal timing information about high threads. In addition, the scheduler might have some internal variables.
Whenever a scheduler-operation rule handles an event, it either corresponds to processing information from the top level (such as threads creation and termination) or to communicating information to the top level (such as thread selection). The rules allow to derive steps of the form $\langle\!|\sigma, \nu|\!\rangle \xrightarrow{\alpha} \langle\!|\sigma', \nu'|\!\rangle$. By convention, we refer to the variables in ν as q, t, r and s and variables in ν' as q', t', r' and s'. When these variables are not explicitly mentioned, we adopt the convention that they remain unchanged after the transition. We assume that besides event-driven transitions, the scheduler might perform internal operations that are not visible at the top level (and may not change the variables above). We abstract away from these transitions, assuming that their event labels are empty. Although the transition system in Figure 4 is nondeterministic, we only consider deterministic instances of schedulers for simplicity. We expect a natural generalization of our results to probabilistic schedulers.
The rules can be viewed as a set of basic assumptions that we expect the scheduler to satisfy. We abstract away from the actual scheduler implementation—it can be arbitrary, as long as it satisfies these basic assumptions and runs infinitely long. We discuss an example of a scheduler that conforms to these assumptions in Section 4.
The rule for event $\alpha^{r}_{\vec{d}}$ ensures that the scheduler updates the appropriate part of the threadpool (low or high, depending on α) with newly created threads. Operation $N(\vec{d})$ returns thread identifiers for $\vec{d}$ and generates fresh ones when new threads are spawn by `fork` or `hfork`. The rule for event $r \rightsquigarrow$ keeps track of a nonterminal step of thread r; as an effect, counter q is decremented. A terminal step of thread r results in a $r \rightsquigarrow \times$ event, which requires the scheduler to remove thread r from the threadpool. Events

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

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

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

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

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

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

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

Fig. 4. Semantics for schedulers

$\uparrow_\circ r'$ 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 $\langle\vec{c}, m, \sigma, \nu\rangle \overset{\alpha}{\rightarrow} \langle\vec{c'}, m', \sigma', \nu'\rangle$, where α ranges over the same set of events as in the semantics for schedulers.
The semantics for threadpool configurations is displayed in Figure 5. 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 $\alpha^r_{\vec{d}}$

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

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

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

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

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

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

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

Fig. 5. Semantics for threadpools

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

```
d : fork_low(c_low);
    fork_medium(c_medium);
    fork_high(c_high);
    hide_medium;
      if k then h := 3;
           else k := 1; k' := 3;
      hide_high;
        if h then h := 0;
             else h := 4; h' := 3;
      unhide_high;
    k'' := 5;
    unhide_medium;
```

Fig. 6. Example of multi-level commands $\texttt{hide}_\ell$, $\texttt{unhide}_\ell$, and $\texttt{fork}_\ell$

Although the semantics accommodates two security levels for threads, extensions to more levels do not pose significant challenges. Assume a security lattice $\mathcal{L}$, where security levels are ordered by a partial order $\sqsubseteq$, with the intention to only allow leaks from data at level ℓ_1 to data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$. The low-and-high policy discussed above forms a two-level lattice with elements low and $high$ so that $low \sqsubseteq high$ but $high \not\sqsubseteq low$.
In the presence of a general security lattice, the threadpool is partitioned into as many parts as the number of security levels. Commands $\texttt{hide}_\ell$, $\texttt{unhide}_\ell$, and $\texttt{fork}_\ell$ are parameterized over security level ℓ. Initially, all threads are in the $\bot$-threadpool. Whenever a thread executes a $\texttt{hide}_\ell$ command, it enters ℓ-threadpool. The semantics needs to ensure that no threads from ℓ'-threadpools, for all ℓ' such that $\ell \not\sqsubseteq \ell'$ may execute until the hidden thread reaches $\texttt{unhide}_\ell$. Naturally, command $\texttt{fork}_\ell$ creates threads in ℓ-threadpool.
To illustrate the use of commands $\texttt{hide}_\ell$, $\texttt{unhide}_\ell$, and $\texttt{fork}_\ell$, we present the thread d in Figure 6. We assume three security levels in our lattice: low, $medium$, and $high$, where $low \sqsubseteq medium \sqsubseteq high$. Commands c_{low}, c_{medium}, and c_{high} describe low, medium, and high threads, respectively. Variables l, k, and h (and their prime versions) are associated with security levels low, $medium$, and $high$, respectively. The program starts by spawning three threads at different security levels. Before the first $\texttt{hide}$, the low-threadpool is composed by the threads d and c_{low}, while threads c_{medium} and c_{high} are placed in the $medium$ and $high$-threadpools, respectively. At this point, any of the threads can be scheduled. Once executed $\texttt{hide}_{medium}$, thread c_{low} is not scheduled for execution until reaching the command $\texttt{unhide}_{medium}$. After executing the first branching instruction, $\texttt{hide}_{high}$ is executing. Then, thread c_{medium} is not able to run and only

d and c_{high} can be executed at that point of the program. After executing $\texttt{unhide}_{high}$, thread c_{medium} can be scheduled to run. Finally, c_{low} can be scheduled to run after executing $\texttt{unhide}_{medium}$.
We will discuss how general multi-level security can be defined and enforced in Sections 4 and 5, respectively.

4 Security specification

We specify security for programs via noninterference. The attacker's view of program memory is defined by a *low-equivalence* relation $=_L$ such that $m_1 =_L m_2$ if the projections of the memories onto the low variables are the same $m_1|_L = m_2|_L$. As formalized in Definition 4 below, a program is secure under some scheduler if for any two initial low-equivalent memories, whenever the two runs of the program terminate, then the resulting memories are also low-equivalent.
We generalize this statement to a class of schedulers, requiring schedulers to comply to the basic assumptions from Section 3 and also requiring that they themselves are not leaky, i.e., that schedulers satisfy a form of noninterference.
Scheduler-related events have different distinguishability levels. Events $\circ^{r}_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\circ r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r is a low thread and r' can be either a low or a high thread) operate on low threads and are therefore low events. On the other hand, events $\bullet^{r}_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\bullet r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r and r' are high threads) are high.
With the security partition defined on scheduler events, we specify the indistinguishability of scheduler configurations via *low-bisimulation*. Because we only consider deterministic schedulers, an equivalent trace-based definition is possible. However, we have chosen a bisimulation-based definition of indistinguishability because it is both intuitive and concise. The intuition behind indistinguishability of scheduler configurations is this: A candidate relation R is a low-bisimulation if the following conditions hold. For two configurations that are related by R, if one of them (say the first) can make a high step to some other configuration then this other configuration will be related to the second configuration. If none of the configurations can make a high step, but one of the configurations can make a low step, then the other one should also be able to make a low step with the same label and the resulting configurations must be related by R. Formally:

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

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

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

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

Fig. 7. Round-robin scheduler

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

Noninterference for schedulers requires low-bisimilarity under any memory:

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

Figure 7 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*

$$\langle\!| c, m_1, \sigma, \nu_{init} |\!\rangle \Downarrow \mathit{cfg}_1 \;\;\&\;\; \langle\!| c, m_2, \sigma, \nu_{init} |\!\rangle \Downarrow \mathit{cfg}_2 \Longrightarrow \\ m(\mathit{cfg}_1) =_L m(\mathit{cfg}_2)$$

A form of scheduler independence is built in the definition by the universal quantification over all noninterferent schedulers. Although the universally quantified condition may appear difficult to guarantee, we will show that the security type system from Section 5 ensures that any typable program is secure. Note that this security definition is *termination-insensitive* [SM03] in that it ignores nonterminating program runs. Our approach can be applied to termination-sensitive security in a straightforward manner, although this is beyond the scope of this paper.
As common, noninterference can be expressed for a general security lattice $\mathcal{L}$ by quantifying over all security levels $\ell \in \mathcal{L}$ and demanding two-level noninterference between data at levels ℓ_1 such that $\ell_1 \sqsubseteq \ell$ (acting as low) and data at levels ℓ_2 such that $\ell_2 \not\sqsubseteq \ell$ (acting as high).

5 Security type system

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

5.1 Typing rules

Figure 8 displays the typing rules for expressions and commands. Suppose Γ is a *typing environment* which includes security type information for variables (whether they are low or high) and two variables, *pc* and *hc*, ranging over security types (low or $high$). By convention, we write Γ_v for Γ restricted to all variables *but* v.
Expression typing judgments have the form $\Gamma \vdash e : \tau$ where τ is low only if all variables in e (denoted $FV(e)$) are low. If there exists a high variable that occurs in e then τ must be $high$. Expression types make no use of type variables *pc* and *hc*.
Command typing judgments have the form $\Gamma \vdash c : \tau$. As a starting point, let us see how the rules track sequential-style information flow. The assignment rule ensures that information cannot leak *explicitly* by assigning an expression that contains high variables into a low variable. Further, *implicit* flows are prevented by the program-counter mechanism [DD77, VSI96]. This mechanism ensures that no assignments to low variables are allowed in the branches of a control statement (`if` or `while`) when the guard of the control statement has type $high$. (We call such `if`'s and `while`'s *high*.) This is achieved by the program-counter type variable *pc* from the typing context Γ. The intended guarantee is that whenever $\Gamma_{pc}, pc \mapsto high \vdash c : \tau$ then c may not assign to low variables.

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

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

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

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

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

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

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

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

Fig. 8. Security type system

The typing rules ensure that branches of high `if`'s and `while`'s may only be typed in a high *pc* context.

Security type variables *hc* (that describes *hiding context*) and τ (that describes the command type) help tracking information flow specific to the multithreaded setting. The main job of these variables is to record whether the current thread is in the high part of the threadpool ($\mathit{hc} = \mathit{high}$) or is in the low part ($\mathit{hc} = \mathit{low}$). 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 $\tau_e \sqsubseteq \Gamma(\mathit{hc})$ 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 8, we obtain a standard type system for securing information flow in sequential programs (cf. [VSI96]). This illustrates that our type provides a general technique for modular extension of systems that track information flow in a sequential setting.

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

5.2 Soundness

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Extension to cooperative schedulers

It is possible to extend our model to cooperative schedulers. This is done by a minor modification of the semantics and type system rules. One can show that the results from Section 5 are preserved under these modifications.
The language is extended with primitive `yield` whose semantics is as follows:

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

The semantics for commands also needs to propagate label $\not\rightsquigarrow$ in the sequential composition rules.
Event $\not\rightsquigarrow$ signals to the scheduler that the current thread yields control. The scheduler semantics needs to react to such an event by resetting counter q' to 0:

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

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 5 now allows α to range over $\{r \rightsquigarrow \bullet, \bullet \rightsquigarrow r, r \not\rightsquigarrow\}$, which propagates yielding events $\not\rightsquigarrow$ from threads to the scheduler. Similar to scheduler events $r \rightsquigarrow \bullet\times$ and $\bullet \rightsquigarrow r\times$, a new transition is added to the threadpool semantics to include the case when `yield` is executed as the last command by a thread.
At the type-system level, yielding control while inside a high control command, as well as inside `hide`/`unhide` pairs, is potentially dangerous. These situations are avoided by a type rule for `yield` that restricts *pc* and *hc* to low:

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

A theorem that implies soundness for the modified type system can be proved similarly to Theorem 1.
Recently, we have suggested a mechanism for enforcing security under cooperative scheduling [RS06b]. Besides checking for explicit and implicit flows, the mechanism ensures that there are no `yield` commands in high context. Similarly, the rule above

implies that `yield` may not appear in high context. On the other hand, the mechanism from [RS06b] allows no dynamic thread creation in high context. This is improved by the approach sketched in this section, because it retains the flexibility that is offered by `hfork`.

7 Ticket purchase example

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

Consider the code fragment in Figure 9. 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_l contains data for the passenger being processed. Variable n_l is assigned the (public) number of frequent-flier miles for flight f_l. Variable m_h is assigned the current number of miles of passenger p_l, which is secret. Variable s_h is assigned the (secret) status (e.g., BASIC or GOLD) of passenger p_l. The value of s_h is then stored in o_h. Variable ok_l records if the procedure to print a ticket has been successful.

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

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

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

results in a typable (and therefore secure) program.

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

Fig. 9. Ticket purchase code

8 Feasibility study of an implementation

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

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

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

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

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

Fig. 10. Extended command syntax

9 Synchronization primitives

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

9.1 Extended language

The extended syntax of the language is displayed in Figure 10. A semaphore is a special variable, written sem, that ranges over nonnegative integers and can only be manipulated by two commands: $\texttt{wait}(sem)$ and $\texttt{signal}(sem)$. We assume, without losing generality, that every semaphore variable is initialized with 0. The semantics for these commands (in the line of [Sab01]) is shown in Figure 11. Command $\texttt{wait}(sem)$ blocks a thread if sem has a value of 0, indicated by event $\overset{b(sem)}{\rightharpoonup}$, or otherwise decrements its value by 1. Command $\texttt{signal}(sem)$ triggers event $\overset{u(sem)}{\rightharpoonup}$.

9.2 Extended semantics for schedulers

Threads that are blocked on semaphore variables cannot be scheduled. Clearly, schedulers need to know when threads are blocked (or not) in order to decide if they can be chosen to run. For this purpose, we introduce a new scheduler variable t_w that stores the set of blocked threads. The semantic rules involving this variable are shown in Figure 12. Rules for selecting threads to run, represented by events $\uparrow_\circ r'$ and $\uparrow_\bullet r'$, are adapted to rule out blocked threads. Observe how threads placed in t_w are removed from the possible values of r'. Events b^r and u^r_a indicate to the scheduler that threads

$$\frac{\langle sem, m \rangle \downarrow 0}{\langle \texttt{wait}(sem), m \rangle \overset{b(sem)}{\rightharpoonup} \langle \texttt{stop}, m \rangle}$$

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

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

Fig. 11. Semantics for `wait`() and `signal`()

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

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

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

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

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

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

Fig. 12. Extended semantics for schedulers

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

9.3 Extended semantics for threadpools

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

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

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

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

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

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

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

Fig. 13. Threadpool semantics for semaphores primitives

Semantic rules for semaphore operations at the level of threadpools are shown in Figure 13. Event b^r_{sem} is triggered when the top level configuration receives a $b(sem)$ signal and the blocked thread is placed at the end of the queue associated with sem. When a $u(sem)$ signal is generated by the running thread, it awakes the first thread in the queue associated with sem and triggers event u^r_{sem}. Moreover, it communicates to the scheduler which thread has been awakened with event u^r_a. In case that the queue associated with sem is empty, no thread is awakened and the scheduler is informed about that by event u^r_r. Events $b^r_{sem}\times$ and $u^r_{sem}\times$ are triggered when threads terminate with synchronization commands under circumstances similar to b^r_{sem} and u^r_{sem}, respectively.

9.4 Attacks using semaphores

Confidentiality of data might be compromised if commands related to semaphores are freely allowed in programs. To illustrate this, we show an attack in Figure 14. The program contains semaphore variables s_1, s_2, p, and f, and variables h and l to store secret and public data, respectively. The code blocks and unblocks threads that assign to public variables in an order that depends on h. That is, the execution of $l := 1$ is followed by $l := 0$ when $h \geq 0$, and $l := 0$ is followed by $l := 1$ otherwise. Observe that the branching command presents no timing differences. Nevertheless, some information

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

Fig. 14. Attack using semaphores

about h is revealed. Restrictions on the use of semaphores are needed in order to avoid such leaks.

9.5 Extended security specification

In Section 4, we state that a program is secure under some scheduler if for any two initial low-equivalence memories, whenever the two runs of the program terminate, then the resulting memories are also low-equivalent. Since semaphores variables are stored in programs memories as any other variables, the low-equivalent relation, as defined previously, is enough to capture the attacker's view of memories even in the presence of semaphores. However, the notion of "configuration cfg *terminates* in configuration cfg'" needs to be adapted. An entire program terminates if there are no blocked threads and no threads to schedule. More precisely, $cfg \Downarrow cfg'$ if $cfg \rightarrow^* cfg'$ where the threadpool of cfg' is empty and the waiting queue $w(sem)$ is empty for every semaphore sem.
To maintain the assumption that schedulers are not leaky, it is necessary to extend the low-bisimulation defined in Section 4 with the events related to synchronization. The distinguishability level of events b^r_{sem}, u^r_{sem}, $b^r_{sem}\times$, and $u^r_{sem}\times$ is the same as the security level of thread r. Definitions 1, 2, and 3 can be easily extended to consider such events and we therefore omit the details here.
We introduce a low-equivalence relation on queues of waiting threads. We define such relation as $=_L$ where $w_1 =_L w_2$ if for every low semaphore sem, it holds that $w_1(sem) = w_2(sem)$. We are now in condition to present the extended security specification:

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

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

9.6 Extended type system

The type system proposed in Section 5 is extended to enforce secure uses of semaphores. As for variables, semaphores have security types (low or $high$) associated with them, which are included in the typing environment Γ. Typing rules for semaphore commands are depicted in Figure 15. The first rule establishes that signals to any semaphore can be performed in low threads. However, signals to public semaphores cannot be sent

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

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

Fig. 15. Typing rules from synchronization primitives

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

Fig. 16. Waiting on low semaphores in high threads

from high threads. To illustrate why this restriction is imposed, we can think about signals on low semaphores as updates on low variables, which must be avoided inside of high threads. The second rule imposes that threads can only wait on semaphores that matches their security level. Waiting on semaphores at security level ℓ in threads of security level ℓ', where $\ell \neq \ell'$, might affect the timing behavior of threads at security level ℓ and ℓ'. For instance, waiting on high semaphores in low threads might affect low threads timing behavior depending on some secret data and lead to internal timing leaks. Moreover, waiting on low semaphores in high threads might affect, also through internal timing, how assignments to public variables are performed. To illustrate this, we show an example in Figure 16. The code involves high thread d_1, low threads d_2 and d_3, low semaphore s_l, and variables h and l to store secret and public information, respectively. Let us assume a scheduler that picks thread d_1 first and then proceeds to run threads for 15 steps before yielding the control. In this case, d_1 terminates before yielding. After that, depending on the secret, two scenarios are possible. If $h \geq 0$, then d_2 blocks until d_3 completes its execution and produces 0 as the final value of l. If $h < 0$, on the other hand, d_2 is likely to execute $l := 0$ before d_3 runs $l := 1$. The final value of l is then 1, which demonstrates that the program is insecure.
The restrictions enforced by the type system are summarized in Figure 17. The first and second columns describe the use of $\texttt{wait}()$ and $\texttt{signal}()$, respectively. The first and second rows describe the use of semaphores in low and high threads, respectively. In addition, s_l (resp., s_h) means that low (resp., high) semaphores can be safely used.

9.7 Soundness of the extension

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

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

Fig. 17. Secure use of semaphores

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

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

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

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

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

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

The following definition introduces a relation between the list of blocked threads and the scheduler memory.

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

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

Theorem 2. *Given a command p and the multithreaded configurations $\langle\!| \vec{c_1}, m_1, \sigma_1, \nu_1, w_1 |\!\rangle$ and $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\rangle$ so that $m_1 =_L m_2$, 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 $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $BEL(w_1, \nu_1), EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$,*

$BEL(w_2, \nu_2)$, written as R_5 $(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2,\ p)$, $(\Gamma[\mathbf{hc} \mapsto low] \vdash c_i : low)_{i \in L(\vec{c_1}, \nu_1)}$, written as $R_6(\vec{c_1}, \nu_1)$, $(\Gamma[\mathbf{hc} \mapsto high,\ \mathbf{pc} \mapsto high] \vdash c_i : high)_{i \in H(\vec{c_1}, \nu_1) \cup H(\vec{c_2}, \nu_2)}$, written as $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $(\Gamma[\mathbf{hc} \mapsto high] \vdash c_i : low)_{i \in EL(\vec{c_1}, \nu_1) \cup EL(\vec{c_2}, \nu_2)}$, 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 $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, $N(w_1) = BL(w_1, \nu_1) \cup BH(w_1, \nu_1) \cup BEL(w_1, \nu_1)$, written $R_{10}(w_1, \nu_1)$, $R_{10}(w_2, \nu_2)$, sets $BH(w_1, \nu_1)$, $BL(w_1, \nu_1)$, $BEL(w_1, \nu_1)$, and $N(\vec{c_1})$ are disjoint, written as $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $BL(w_1, \nu_1) = BL(w_2, \nu_2)$, written as $R_{12}(w_1, \nu_1, w_2, \nu_2)$, $(\Gamma[\mathbf{hc} \mapsto low] \vdash c_i : low)_{i \in BL(w_1, \nu_1)}$, written as $R_{13}(w_1, \nu_1)$, $(\Gamma[\mathbf{hc} \mapsto high,\ \mathbf{pc} \mapsto high] \vdash c_i : high)_{i \in BH(w_1, \nu_1) \cup BH(w_2, \nu_2)}$, written as $R_{14}(w_1, \nu_1, w_2, \nu_2)$, $(\Gamma[\mathbf{hc} \mapsto high] \vdash c_i : low)_{i \in BEL(w_1, \nu_1) \cup BEL(w_2, \nu_1)}$, written as $R_{15}(w_1, \nu_1, w_2, \nu_2)$, $w_1 =_L w_2$, written as $R_{16}(w_1, w_2)$, $w_1 \rhd \nu_1$, written as $R_{17}(w_1, \nu_1)$, $R_{17}(w_2, \nu_2)$, then:

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

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

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

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

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

10 Conclusion

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

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

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

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

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

Practical enforcement We have demonstrated that security can be enforced for both cooperative and preemptive schedulers using a compositional type system. The type system accommodates permissive programming. We have illustrated by an example in Section 7 that the permissiveness of dynamic thread creation is not majorly restricted by the type system. The type system does not involve padding to eliminate timing leaks at the cost of efficiency.
Most recently, together with Barthe and Rezk [BRRS07], we have adapted our type system to an unstructured assembly language. Our future work plans include handling richer low-level languages (such as languages with exceptions and bytecode) and facilitating tool support for them.

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix

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

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

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

Proof. By simple induction on the typing derivation. □

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

Proof. By induction on the type derivation of c.

`skip`) It holds trivially since `skip` does not modify the low memory and it does not produce any labeled event.

$x := e$) By the typing rule for assignment, we know that $\Gamma(x) = high$. The result follows from the fact that the assignment does not change the low memory and produces an unlabeled event.

$c_1; c_2$) By the typing derivation for sequential composition, we have that

$$\Gamma_{hc}, hc \mapsto high \vdash c_1 : \tau' \tag{1}$$
$$\Gamma_{hc}, hc \mapsto \tau' \vdash c_2 : \tau \tag{2}$$

for some type τ'. By the semantic rule for sequential composition, we have two more cases to consider:

$$\langle\!| c_1, m |\!\rangle \xrightarrow{\alpha} \langle\!| \texttt{stop}, m' |\!\rangle \tag{3}$$
$$\langle\!| c_1, m |\!\rangle \xrightarrow{\alpha} \langle\!| c_1', m' |\!\rangle \tag{4}$$

Both cases are proved similarly. Thus, we only show how to prove the later one. The result then follows from applying IH on (1) and (4), and thus obtaining $m =_L m'$ and that $\alpha \notin \{\circ, \rightsquigarrow \bullet\}$.

`if` e `then` c_1 `else` c_2) It holds trivially since the semantic rule for branchings reduces the `if` to c_1 or c_2 without modifying the memory and without producing any labeled events.

`while` e `do` c) It is proved similarly to the `if` $-$ `then` $-$ `else`.

`hfork`$(c, \vec{d})$) It holds trivially since the semantic rule for `hfork` reduces to $\langle\!| c, m |\!\rangle$ and produces the labeled event $\alpha = \bullet_{\vec{d}}$.

□

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

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

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

Proof. By case analysis on c. The only typable command under the hypothesis of the lemma is the sequential composition. Then, we consider the case when $c = c_1; c_2$ for the given commands c_1 and c_2. We assume, by associativity of sequential composition, that c_1 consists on a single command. The cases when $c_1 = \texttt{skip}$ and $c_1 = x := e$ are proved by just inspecting the typing rules and applying the subsumption rule when needed. The interesting cases are proved as follows.

$c_1 = \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f$) The proof proceeds similarly regardless of the boolean value obtained from evaluating e. Therefore, we only show the case when the guard is evaluated to True. By inspecting the semantics rules for commands, we know that $\langle\!| c_1; c_2, m |\!\rangle \rightharpoonup \langle\!| c_t, m |\!\rangle$. By the typing derivation of $c_1; c_2$, we know that

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto \tau_e \sqcup \tau_c \sqcup high, \mathbf{hc} \mapsto high \vdash c_t : high \quad (5)$$

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto \tau_c, \mathbf{hc} \mapsto high \vdash c_2 : low \quad (6)$$

If $\tau_c = high$, the result immediately follows from applying the typing rule for sequential composition to (5) and (6). Otherwise, we can apply the subsumption rule to (5) to obtain that

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto \tau_c, \mathbf{hc} \mapsto high \vdash c_t : high \quad (7)$$

The result follows from applying the typing rule for sequential composition to (7) and (6).

$c_1 = \texttt{while}\ e\ \texttt{do}\ c_w$) It is proved as the conditional case. The result follows by inspecting the typing derivation of $c_1; c_2$, applying the sequential composition and subsumption typing rules when needed.

$c_1 = \texttt{hfork}(c, \vec{d})$) By inspecting the semantics rules for commands, we know that $\langle\!| c_1; c_2, m |\!\rangle \xrightharpoonup{\bullet \vec{d}} \langle\!| c; c_2, m |\!\rangle$. By inspecting the typing derivation of $c_1; c_2$, we obtain that

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto high, \mathbf{hc} \mapsto high \vdash c : high \quad (8)$$

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto \tau_c, \mathbf{hc} \mapsto high \vdash c_2 : low \quad (9)$$

If $\tau_c = high$, the result immediately follows from applying the typing rule for sequential composition to (8) and (9). Otherwise, we can apply the subsumption rule to (8) to obtain that

$$\Gamma_{hc,pc}, \mathbf{pc} \mapsto \tau_c, \mathbf{hc} \mapsto high \vdash c : high \quad (10)$$

The result follows from applying the typing rule for sequential composition to (10) and (9).

□

Lemma 6. For a given command c such that $\Gamma_{hc}, \mathbf{hc} \mapsto low \vdash c : low$, memories m_1 and m_2 such that $m_1 =_L m_2$, and $\langle\!| c, m_1 |\!\rangle \xrightharpoonup{\alpha} \langle\!| c', m'_1 |\!\rangle$; it holds that $\langle\!| c, m_2 |\!\rangle \xrightharpoonup{\alpha} \langle\!| c', m'_2 |\!\rangle$ and $m'_1 =_L m'_2$.

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

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

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

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

Proof. By induction on the size of command c.

$|c| = 1$) The only typable command of size 1 is `unhide`. Thus, the result follows from taking $c = \texttt{unhide}$.

$|c| > 1$) The only typable command which size bigger than 1 is the sequential composition. In other words, $c = c_1; c_2$, for a single command c_1 and a command c_2.

$c_1 = \texttt{skip}$) We know then that $\Gamma_{hc}, hc \mapsto high \vdash \texttt{skip} : high$ and $\Gamma_{hc}, hc \mapsto high \vdash c_2 : low$. Therefore, we can conclude that

$$\Gamma_{hc}, hc \mapsto low \vdash \texttt{hide}; c_2 : low \tag{11}$$

By applying IH on (11), we obtain that there exists commands c_2' and p_2 such that $c_2' \in \{c_2'; \texttt{unhide},\ \texttt{unhide},\ c_2'; \texttt{unhide}; p,\ \texttt{unhide}; p\}$ where c_2' has no `unhide` commands. The result follows by taking $c' = \texttt{skip}; c_2'$ and $p = p_2$.

$c_1 = \texttt{unhide}$) The result trivially follows by taking $p = c_2$ and because $c = \texttt{unhide}; p$.

$c_1 = x := e$) This case is proved in a similar way as when $c_1 = \texttt{skip}$.

$c = \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'$) By the typing derivation of c, we know that

$$\Gamma_{hc}, hc \mapsto high \vdash \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'; c_2 : low \tag{12}$$

By the type derivation of (12), we also have that

$$(\Gamma_{hc}, hc \mapsto high, pc \mapsto high \vdash c_i' : high)_{i=1,2} \tag{13}$$
$$\Gamma_{hc}, hc \mapsto high \vdash \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2' : high \tag{14}$$
$$\Gamma_{hc}, hc \mapsto high \vdash c_2 : low \tag{15}$$

Therefore, we can conclude that

$$\Gamma_{hc}, hc \mapsto low \vdash \texttt{hide}; c_2 : low \tag{16}$$

By applying Lemma 2 to (13), commands `hide` and `unhide` do not appear in $(c_i')_{i=1,2}$. By applying IH on (16), we obtain that there exists commands c'' and p_2 such that $c_2 \in \{c''; \texttt{unhide},\ \texttt{unhide},\ c''; \texttt{unhide}; p_2,\ \texttt{unhide}; p_2\}$, where c'' has no `unhide` commands. The result follows by taking command $c' = \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'; c''$ or $c' = \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'$ (depending on the form of c_2) and $p = p_2$.

$c = (\texttt{while}\ e\ \texttt{do}\ c_1); c_2$) In this case, the proof is similar to that when command $c = \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'$.

$c = \texttt{hfork}(c, \vec{d}); c_2$) By the type derivation of c, we know that

$$\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash c : high \tag{17}$$
$$\Gamma_{hc,pc}, hc \mapsto high, pc \mapsto high \vdash \vec{d} : high \tag{18}$$
$$\Gamma_{hc}, hc \mapsto high \vdash c_2 : low \tag{19}$$

Therefore, we can conclude that

$$\Gamma_{hc}, hc \mapsto low \vdash \texttt{hide}; c_2 : low \tag{20}$$

By applying Lemma 2 to (17) and (18), we obtain that c and $\vec{d}$ contains no `hide` or `unhide`. By applying IH on (20), we obtain that there exists commands c'' and p_2 such that $c_2 \in \{c''; \texttt{unhide},\ \texttt{unhide},\ c''; \texttt{unhide}; p_2,\ \texttt{unhide}; p_2\}$, where c'' has no `unhide`. The result follows by taking $c' = \texttt{hfork}(c, \vec{d}); c''$ or $c' = \texttt{hfork}(c, \vec{d})$ (depending on the form of c_2) and $p = p_2$.

□

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

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

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

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

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

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

In order to prove $R_5(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2, p')$, $R_7(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, and $R_8(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, we need to split the proof in two more cases: $c_r \in H(\vec{c_1}, \nu_1)$ and $c_r \in EL(\vec{c_1}, \nu_1)$.

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

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

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

$c_r \in H(\vec{c_1}, \nu_1)$) $R_1(m_1', m_2)$ holds by applying Lemma 3. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1',$ $\vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu_1')$ also holds. $R_7(\vec{c_1'},$ $\nu_1', \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 4. By taking $p' = p$, we have that $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p')$ and $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ hold because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold; and because the eventually low thread, if there exists one, has made no progress. $R_9(\sigma_1', \nu_1', \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$\nu'_1, \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$c_r \in H(\vec{c_1}, \nu_1)$) $R_1(m'_1, m_2)$ holds by applying Lemma 3. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu'_1)$ also holds. $R_7(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the thread c_r has finished. By taking $p' = p$, we have that $R_5(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2, p)$ and $R_8(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ hold because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold; and because the eventually low thread, if there exists one, has made no progress. $R_9(\sigma'_1, \nu'_1, \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$\alpha_1 = \uparrow_\bullet r'$) By taking $p' = p$, we have that $R_1(m'_1, m_2)$, $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu'_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}', \nu'_1, \vec{c_2}, \nu_2)$, $R_5(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2, p')$, $R_6(\vec{c_1}', \nu'_1)$, $R_7(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$, and $R_8(\vec{c'_1}, \nu'_1, \vec{c_2}, \nu_2)$ holds since $R_1(m_1, m_2)$, $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $R_6(\vec{c_1}, \nu_1)$, $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold and because the transition has only modified the variable t_r in the scheduler. $R_9(\sigma'_1, \nu'_1, \sigma_2, \nu_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$\alpha_1 = \circ^r_{\vec{d}}$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \texttt{fork}(c, \vec{d})$ or $c_r = \texttt{fork}(c, \vec{d}); c^*$ for some commands c and c^*. We are only going to show the proof for the case when $c_r = \texttt{fork}(c, \vec{d}); c^*$ since the proof for $c_r = \texttt{fork}(c, \vec{d})$ proceeds in a similar way. By inspecting the semantics for threadpools and commands, we have the transition $\langle c_r, m_1 \rangle \xrightarrow{\circ_{\vec{d}}} \langle c; c^*, m_1 \rangle$, and that $\langle \sigma_1, \nu_1 \rangle \xrightarrow{\circ^r_{\vec{d}}} \langle \sigma'_1, \nu'_1 \rangle$. Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \xrightarrow{\circ^r_{\vec{d}}} \langle \sigma'_2, \nu'_2 \rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle c_r, m_2 \rangle \xrightarrow{\circ_{\vec{d}}} \langle c; c^*, m_2 \rangle$. We can therefore conclude that $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle \xrightarrow{\circ^r_{\vec{d}}} \langle \vec{c_2}', m'_2, \sigma'_2, \nu'_2 \rangle$. $R_1(m'_1, m'_2)$ holds by applying Lemma 6. $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}', \nu'_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}', \nu'_2)$ hold since propositions $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds, and by inspecting the semantics for the scheduler together with the fact that $N(\vec{d})$ are fresh names for threads. $R_4(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since

$R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 added the same new low threads to both configurations. By taking $p' = p$, we have that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p)$ holds since proposition $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and because the eventually low thread, if exists one, has made no progress. $R_6(\vec{c_1}', \nu_1')$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and by inspecting the Lemma 7. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the low step α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the step α_1. Finally, proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = r\rightsquigarrow$) By inspecting the semantics rules for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, $\langle c_r, m_1 \rangle \rightharpoonup \langle c', m_1' \rangle$, and that $\langle \sigma_1, \nu_1 \rangle \overset{r\rightsquigarrow}{\rightharpoondown} \langle \sigma_1', \nu_1' \rangle$. Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \overset{r\rightsquigarrow}{\rightharpoondown} \langle \sigma_2', \nu_2' \rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle c_r, m_2 \rangle \rightharpoonup \langle c', m_2 \rangle$. Therefore, we can conclude that the transition $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle \overset{r\rightsquigarrow}{\rightharpoonup} \langle \vec{c_2}', m_2', \sigma_2', \nu_2' \rangle$ holds.
$R_1(m_1', m_2')$ holds by applying Lemma 6 to c_r. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by applying Lemma 6 to c_r. By taking $p' = p$, we have that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p)$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and because the eventually low threads, if they exist, have made no progress. $R_6(\vec{c_1}', \nu_1')$ holds since $R_6(\vec{c_1}, \nu_1)$ holds and by applying Lemma 7 to c_r. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exist, have been not modified by the transition α_1. Finally, $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

$\alpha_1 = r \rightsquigarrow \bullet$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r = \texttt{hide}; c^*$ for some command c^*, $\langle\!\vert c_r, m_1 \vert\!\rangle \xrightarrow{\rightsquigarrow\bullet} \langle\!\vert c_r', m_1 \vert\!\rangle$, and that $\langle\!\vert \sigma_1, \nu_1 \vert\!\rangle \xrightarrow{r\rightsquigarrow\bullet} \langle\!\vert \sigma_1', \nu_1' \vert\!\rangle$. We also know that $\langle\!\vert c_r, m_2 \vert\!\rangle \xrightarrow{\rightsquigarrow\bullet} \langle\!\vert c_r', m_2 \vert\!\rangle$ since $L(\vec{c_1}) = L(\vec{c_2})$. Moreover, we know that $\langle\!\vert \sigma_1, \nu_1 \vert\!\rangle \sim_L \langle\!\vert \sigma_2, \nu_2 \vert\!\rangle$ and α_1 is low, we also have that $\langle\!\vert \sigma_2, \nu_2 \vert\!\rangle \xrightarrow{r\rightsquigarrow\bullet} \langle\!\vert \sigma_2', \nu_2' \vert\!\rangle$. We can thus conclude that the transition $\langle\!\vert \vec{c_2}, m_2, \sigma_2, \nu_2 \vert\!\rangle \xrightarrow{r\rightsquigarrow\bullet} \langle\!\vert \vec{c_2}', m_2', \sigma_2', \nu_2' \vert\!\rangle$ holds.
We know that $EL(\vec{c_1}) = \emptyset$ because a low thread was scheduled to produce the event $r \rightsquigarrow \bullet$. Then, $EL(\vec{c_2}) = \emptyset$ since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds. By applying Lemma 8 to c_r, we know that $c^* = c'; \texttt{unhide}$, $c^* = \texttt{unhide}$, $c^* = c'; \texttt{unhide}; p^*$, or $c^* = \texttt{unhide}; p^*$, where c' has no $\texttt{unhide}$.
$R_1(m_1', m_2')$ holds since $m_1' = m_1$ and $m_2' = m_2$. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, $R_3(\vec{c_2}', \nu_2')$, and $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ hold since the following equalities $EL(\vec{c_1}, \nu_1) = EL(\vec{c_2}, \nu_2) = \emptyset$ hold, $(L(\vec{c_i}', \nu_i') = L(\vec{c_i}, \nu_i) \backslash \{c_r\})_{i=1,2}$, and $(EL(\vec{c_i}', \nu_i') = \{c_r\})_{i=1,2}$ hold by inspecting the semantics for threadpools and the scheduler.
In the cases where $c^* = c'; \texttt{unhide}$ or $c^* = \texttt{unhide}$, $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p')$ holds by taking $p' = \texttt{skip}$ (see Definition 6). On the other cases, by taking $p' = p^*$, we know that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', p^*)$ holds because the application of Lemma 8 gave us the appropriate p^* that satisfies Definition 6. $R_6(\vec{c_1}', \nu_1')$ holds since $L(\vec{c_1}', \nu_1') =$ $L(\vec{c_1}, \nu_1) \backslash \{c_r\}$ and $R_6(\vec{c_1}, \nu_1)$ hold. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since proposition $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and by inspecting the type derivation of c_r. Finally, proposition $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

$\alpha_1 = \bullet \rightsquigarrow r_e$) We know that $r_e \in t_{e_1}$. By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_{r_e} = \texttt{unhide}; c^*$ or $c_{r_e} = \texttt{unhide}$ for some command c^*, $c_{r_e} \in EL(\vec{c_1}, \nu_1)$, $\langle\!\vert c_{r_e}, m_1 \vert\!\rangle \xrightarrow{\bullet\rightsquigarrow} \langle\!\vert c^*, m_1 \vert\!\rangle$, and that $\langle\!\vert \sigma_1, \nu_1 \vert\!\rangle \xrightarrow{\bullet\rightsquigarrow r_e}$ $\langle\!\vert \sigma_1', \nu_1' \vert\!\rangle$. We are only going to consider the case when $c_{r_e} = \texttt{unhide}; c^*$ since the proof for $c_{r_e} = \texttt{unhide}$ is analogous. Therefore, we omit the proof when $\alpha_1 = \bullet \rightsquigarrow r_e \times$.
Because $\langle\!\vert \sigma_1, \nu_1 \vert\!\rangle \sim_L \langle\!\vert \sigma_2, \nu_2 \vert\!\rangle$ and α_1 is low, we also have that $\langle\!\vert \sigma_2, \nu_2 \vert\!\rangle \xrightarrow{\bullet\rightsquigarrow r_e}$

$\langle\!| \sigma_2', \nu_2' |\!\rangle$. Because $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds, we know that $c^* = p$ and that the thread with name r_e belongs to the threadpool $\vec{c_2}$ as well. Let us call it $c^2_{r_e}$. Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and it is not possible for a thread to make progress by a high computation, we have that $c^2_{r_e} = \texttt{unhide}; p$. As a consequence of that, it holds that $\langle\!| c^2_{r_e}, m_2 |\!\rangle \overset{\bullet\rightsquigarrow}{\rightharpoonup} \langle\!| p, m_2 |\!\rangle$. Thus, transition $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2 |\!\rangle \overset{\bullet\rightsquigarrow r_e}{\rightharpoonup} \langle\!| \vec{c_2}', m_2', \sigma_2', \nu_2' |\!\rangle$ holds.

$R_1(m_1', m_2')$ holds trivially since `unhide` has no changed the memories. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds; and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because after the transition α_1, the threads c_{r_e} and $c^2_{r_e}$ become the thread p. By inspecting the semantics for the scheduler, we have that $EL(\vec{c_1'}, \nu_1') = EL(\vec{c}_2', \nu_2') = \emptyset$. Then, by taking $p' = \texttt{skip}$, it trivially holds that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', \texttt{skip})$. $R_6(\vec{c_1}', \nu_1')$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds ; and by inspecting the type derivation of c_{r_e}. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $EL(\vec{c_1}', \nu_1') = EL(\vec{c}_2', \nu_2') = \emptyset$. Finally, $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$.

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

□

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

Proof. For arbitrary σ, m_1, and m_2 so that $m_1 =_L m_2$ and σ is noninterferent, assume $\langle\!| c, m_1, \sigma, \nu_{init} |\!\rangle \Downarrow \mathit{cfg}_1$ & $\langle\!| c, m_2, \sigma, \nu_{init} |\!\rangle \Downarrow \mathit{cfg}_2$. 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. □

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

$\vdash c_i : high)_{i \in BH(w_1,\nu_1) \cup BH(w_2,\nu_2)}$, written as $R_{14}(w_1,\nu_1,w_2,\nu_2)$, $(\Gamma[\mathit{hc} \mapsto high] \vdash c_i : low)_{i \in BEL(w_1,\nu_1) \cup BEL(w_2,\nu_1)}$, written as $R_{15}(w_1,\nu_1,w_2,\nu_2)$, $w_1 =_L w_2$, written as $R_{16}(w_1,w_2)$, $w_1 \rhd \nu_1$, written as $R_{17}(w_1,\nu_1)$, $R_{17}(w_2,\nu_2)$, then:

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

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

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

Proof. By case analysis on command/scheduler steps. We are only going to show the proofs for commands related to synchronization and `unhide` when the configuration $\langle \vec{c_1}, m_1, \sigma_1, \nu_1 \rangle$ makes some progress. The proof for other commands proceeds similarly as in Theorem 1. We assume that the thread c_r belongs to $\vec{c_1}$. Analogous proofs are obtained when $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle$ makes progress instead.

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

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

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

$R_1(m'_1, m_2)$ holds by inspecting the semantics of threadpools and applying Lemma 3 to c_r. By inspecting the threadpool semantics and since c_r has been blocked, we have that $N(\vec{c}_1\,') = N(\vec{c}_1) \backslash \{c_r\}$. Moreover, we have that $N(\vec{c}\,'_1) = H(\vec{c}_1, \nu_1) \cup L(\vec{c}_1, \nu_1) \cup EL(\vec{c}_1, \nu_1) \setminus \{c_r\}$. We also know that $H(\vec{c_1}\,', \nu'_1) = H(\vec{c}_1, \nu_1) \setminus \{c_r\}$ since $r \in t_{\bullet_1}$ and c_r has been blocked. By this last fact and $R_3(\vec{c_1}, \nu_1)$, it holds $R_2(\vec{c_1}\,', \nu'_1)$ as expected. $R_2(\vec{c_2}, \nu_2)$ holds since it already holds by hypothesis. $R_3(\vec{c_1}\,', \nu'_1)$ holds since $R_3(\vec{c_1}, \nu_1)$ holds and $H(\vec{c_1}\,', \nu'_1) = H(\vec{c}_1, \nu_1) \backslash \{c_r\}$. $R_3(\vec{c_2}, \nu_2)$ holds since it already holds by hypothesis. $R_4(\vec{c}\,'_1,$

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

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

Propositions R_{1-4} can be proved in a similar way as when $c_r \in H(\vec{c_1}, \nu_1)$. By hypothesis, we know that $BEL(w_1, \nu_1), EL(\vec{c_1}, \nu_1) \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, $BEL(w_2, \nu_2)$. By inspecting the semantics, we also know that $t_{e_1} = \{c_r\}$. By inspecting Definition 9, we have that $\emptyset, \{c_r\} \sim_{el,p,t_{e_1}} EL(\vec{c_2}, \nu_2)$, $BEL(w_2, \nu_2)$. We need to do case analysis to determine if $EL(\vec{c_2}, \nu_2) = \emptyset$ or $BEL(w_2, \nu_2) = \emptyset$. Both cases proceed in a similar way and therefore we omit when $BEL(w_2, \nu_2) = \emptyset$. Consequently, we have that $\emptyset, \{c_r\} \sim_{el,p,t_{e_1}} \emptyset, \{d_r\}$ where there exist commands c' and d' without `unhide` instructions such that $c_r \in \{c'; \texttt{unhide}, \texttt{unhide}\}$ and $d_r \in \{d'; \texttt{unhide}, \texttt{unhide}\}$ or $c_r \in \{c'; \texttt{unhide}; p, \texttt{unhide}; p\}$ and $d_r \in \{d'; \texttt{unhide}; p, \texttt{unhide}; p\}$. Since the triggered event is b^r_{sem}, we can deduce that $c_r \in \{c'; \texttt{unhide}\}$ or $c_r \in \{c'; \texttt{unhide}; p\}$. By inspecting the threadpool semantics, we have that $\langle\!| c_r, m_1 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c_r', m_1' |\!\rangle$, where $c_r' \neq \texttt{stop}$. Consequently, we know that $c_r' \in \{c''; \texttt{unhide}, \texttt{unhide}\}$ or $c_r' \in \{c''; \texttt{unhide}; p, \texttt{unhide}; p\}$ where $\langle\!| c', m_1 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c'', m_1' |\!\rangle$. Let us take $p' = p$. Since $t'_{w_1} = t_{w_1} \cup \{r\}$ and $t_{e_1} = \{r\}$, we have that $BEL(w_1', \nu_1') = \{c_r'\}$, $EL(\vec{c_1}', \nu_1') = \emptyset$, and that $\{c_r'\}, \emptyset \sim_{el,p',t_{e_1}} \emptyset, \{d_r\}$ holds. Therefore, $R_5(\vec{c_1}', w_1', \nu_1', \vec{c_2}, \nu_2, w_2, p')$ holds. Propositions $R_6(\vec{c_1}', \nu_1')$ and $R_7(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$ hold since $R_6(\vec{c_1}, \nu_1)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ hold and because low and high threads have made no progress. Proposition $R_8(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$ holds since $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and $EL(\vec{c_1}', \nu_1') = \emptyset$. Propositions R_{9-13} are proved similarly as when $c_r \in H(\vec{c_1}, \nu_1)$. Proposition $R_{14}(w_1', \nu_1', w_2, \nu_2)$ holds since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ holds and because high threads have made no progress. By $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, applying Lemma 5 to c_r, and $R_{15}(w_1, \nu_1, w_2, \nu_2)$, we obtain that $R_{15}(w_1', \nu_1', w_2, \nu_2)$ holds. Proposition $R_{16}(w_1', w_2)$ holds since $R_{16}(w_1, w_2)$ holds and because no low semaphores or the eventually low thread, if it exists, are affected by the transition. By inspecting the semantics, $c_r \in EL(\vec{c_1}, \nu_1)$, and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, we have that $\Gamma(sem) = high$ and $r \in t_{e_1}$. By these last facts and $R_{17}(w_1, \nu_1)$, we obtain that $R_{17}(w_1', \nu_1')$ holds. Proposition $R_{17}(w_2, \nu_2)$ holds since it already holds by hypothesis.

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

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

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

By inspecting the semantics of threadpools, this event can be produced by two rules in Figure 13. Which rule is applied depends on the existence of threads on the waiting list $w_1(sem)$ when executing `signal`, which is captured by the scheduler events u^r_r and u^r_a. The proof proceeds similarly in both cases. Therefore, we omit the case when the scheduler triggers the event u^r_r.

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

$R_1(m_1', m_2)$ holds by inspecting the semantics of threadpools and applying Lemma 3 to c_r. By semantics, the thread c_a has been awakened and place into the threadpool. Since $R_{17}(w_1, \nu_1)$ holds and $\Gamma(sem) = high$ by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, we have that $a \in t_{\bullet_1} \cup t_{e_1}$ and consequently $R_2(\vec{c}'_1, \nu_1')$ holds. Proposition $R_2(\vec{c_2}, \nu_2)$ holds since it holds by hypothesis. Proposition $R_3(\vec{c}'_1, \nu_1')$ holds by $R_3(\vec{c_1}, \nu_1)$ and $R_{11}(w_1, \nu_1)$. Proposition $R_3(\vec{c_2}, \nu_2)$ holds since it holds already by hypothesis. Propositions R_{4-6} can be proved in a similar way as when $\alpha_1 = b^r_{sem}$. To prove proposition $R_7(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, we need to consider if $a \in t_{\bullet_1}$. If that is the case, it is proved by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and $R_{14}(w_1, \nu_1, w_2, \nu_2)$. Otherwise, it holds since it already holds by hypothesis. To prove proposition $R_8(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$, we need to consider if $a \in t_{e_1}$. If that is the case, it is proved by $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$. Otherwise, it holds since it already holds by hypothesis. By inspecting the semantics, we know that $N(w_1') = N(w_1) \setminus \{c_a\}$ and that, depending if $a \in t_{\bullet_1}$ or $a \in t_{e_1}$, we have that $BH(w_1', \nu_1') = BH(w_1, \nu_1) \setminus \{c_a\}$ or $BEL(w_1', \nu_1') = BEL(w_1, \nu_1) \setminus \{c_a\}$, respectively. Consequently, we obtain that $R_{10}(w_1', \nu_1')$ holds. Proposition $R_{10}(w_2, \nu_2)$ since it already holds by hypothesis. Proposition $R_{11}(w_1', \nu_1')$ holds since $R_{11}(w_1, \nu_1)$ holds and c_a has been move from one subpool of threads to another. Proposition $R_{12}(w_2, \nu_2)$ holds since it already holds by hypothesis. Proposition $R_{13}(w_1' , \nu_1')$ holds since $R_{13}(w_1, \nu_1)$ holds and no threads have been blocked by the transition α_1. Propositions $R_{14}(w_1', \nu_1', w_2, \nu_2)$ and $R_{15}(w_1', \nu_1', w_2, \nu_2)$ hold since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$ hold and because c_a has been removed from the subpool of threads $BH(w_1', \nu_1')$ or $BEL(w_1', \nu_1')$ by the transition α_1. Since $R_{17}(w_1, \nu_1)$ holds and $\Gamma(sem) = high$ by $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, it holds that $w_1' =_L w_1$ and thus $R_{16}(w_1', w_2)$ holds. Proposition $R_{17}(w_1', \nu_1')$ holds since $R_{17}(w_1, \nu_1)$

holds and c_a has been removed from the waiting list of sem. Proposition $R_{17}(w_2, \nu_2)$ holds since it already holds by hypothesis.

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

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

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

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

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

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

$\alpha_1 = b^r_{sem}$) By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \mathtt{wait}(sem); c'$ for some command c'. By inspecting the semantics for threadpools and commands, we have the transition $\langle\!| c_r, m_1 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c'_r, m_1 |\!\rangle$, and that $\langle\!| \sigma_1, \nu_1 |\!\rangle \overset{b^r}{\rightharpoondown} \langle\!| \sigma'_1, \nu'_1 |\!\rangle$. Because $\langle\!| \sigma_1, \nu_1 |\!\rangle \sim_L \langle\!| \sigma_2, \nu_2 |\!\rangle$ and α_1 is low, we also have that $\langle\!| \sigma_2, \nu_2 |\!\rangle \overset{b^{sem}}{\rightharpoondown} \langle\!| \sigma'_2, \nu'_2 |\!\rangle$. In addition to that, we also know that $c_r \in L(\vec{c_2}, \nu_2)$ since $L(\vec{c_2}, \nu_2) = L(\vec{c}_1, \nu_1)$, and that $\langle\!| c_r, m_2 |\!\rangle \overset{b(sem)}{\rightharpoonup} \langle\!| c'_r, m_2 |\!\rangle$. We can therefore conclude that $\langle\!| \vec{c_2}, m_2, \sigma_2, \nu_2, w_2 |\!\rangle \overset{b^r_{sem}}{\rightarrow} \langle\!| \vec{c_2}', m'_2, \sigma'_2, \nu'_2, w'_2 |\!\rangle$. $R_1(m'_1, m'_2)$ holds by applying Lemma 6 to c_r. Propositions $R_2(\vec{c_1}', \nu'_1)$, $R_2(\vec{c_2}', \nu'_2)$, $R_3(\vec{c_1}', \nu'_1)$, and $R_3(\vec{c_2}', \nu'_2)$ hold since propositions $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, $R_3(\vec{c_2}, \nu_2)$, $R_{10}(w_1, \nu_1)$ and $R_{10}(w_2, \nu_2)$ hold and since $L(\vec{c}'_1, \nu'_1) = L(\vec{c}_1, \nu_1) \setminus \{c'_r\}$ by inspecting the semantics for threadpools. Proposition $R_4(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 block the same low thread on both configurations. By taking $p' = p$, we have that $R_5(\vec{c_1}', w'_1, \nu'_1, \vec{c_2}', w'_2, \nu'_2, p)$ holds since proposition $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, \nu_2, w_2, p)$ holds and because the eventually low thread, if exists one, has made no progress. $R_6(\vec{c_1}', \nu'_1)$ holds by $R_6(\vec{c_1}, \nu_1)$ and the fact that $c'_r \notin L(\vec{c_1}', \nu'_1)$. $R_7(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds because $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu'_1, \vec{c_2}', \nu'_2)$ holds because $R_8(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the eventually low threads in both configurations, if they exists, have been not modified by the transition α_1. Proposition $R_9(\sigma'_1, \nu'_1, \sigma'_2, \nu'_2)$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. By inspecting the semantics of threadpools, $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$, and $R_{12}(w_1, \nu_1, w_2, \nu_2)$, we have that $N(w'_1) = N(w'_2) = N(w_1) \cup \{c'_r\}$, $N(\vec{c_1}') = N(\vec{c}_2') = N(\vec{c}_1) \setminus \{c_r\}$, and $BL(w'_1, \nu'_1) = BL(w'_2, \nu'_2) = BL(w_1, \nu_2) \cup \{c'_r\}$. By these last facts, $R_{10}(w_1, \nu_2)$, $R_{10}(w_2, \nu_2)$, $R_{11}(w_1, \nu_1)$, $R_{11}(w_2, \nu_2)$, $R_{12}(w_1, \nu_1, w_2, \nu_2)$, we have that it holds $R_{10}(w'_1, \nu'_2)$, $R_{10}(w'_2, \nu'_2)$, $R_{11}(w'_1, \nu'_1)$, $R_{11}(w'_2, \nu'_2)$, $R_{12}(w'_1, \nu'_1, w'_2, \nu'_2)$. $R_{13}(w'_1, \nu'_1)$ holds since $R_{13}(w_1, \nu_1)$ holds and by Lemma 7. Proposition $R_{14}(w'_1, \nu'_1, w'_2, \nu'_2)$ and $R_{15}(w'_1, \nu'_1, w'_2, \nu'_2)$ holds since $R_{14}(w_1, \nu_1, w_2, \nu_2)$ and $R_{15}(w_1, \nu_1, w_2, \nu_2)$ holds and because transition α_1 does not affect high and eventually low threads, if they exists. Since

$c_r = wait(sem); c'$, $c_r \in L(\vec{c_1}, \nu_1)$, $R_6(\vec{c_1}, \nu_1)$, and typing rules, we have that $\Gamma(sem) = low$. By this fact and $R_{16}(w_1, w_2)$, By inspecting the semantics, $c_r \in L(\vec{c_1}, \nu_1)$, and $R_6(\vec{c_1}, \nu_1)$, we have that $\Gamma(sem) = low$, $r \in t_{\circ_1} = t_{\circ_2}$. By these last facts, $R_{17}(w_1, \nu_1)$, and $R_{17}(w_2, \nu_2)$, we obtain that $R_{17}(w_2', \nu_2')$ and $R_{17}(w_2', \nu_2')$ hold.

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

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

By inspecting the semantics of threadpools, this event can be produced by two rules in Figure 13. Which rule is applied depends on the waiting list $w(sem)$ when executing `signal`, which is captured by the scheduler events u^r_r and u^r_a. The proof proceeds similarly in both cases. Therefore, we omit the case when the scheduler triggers the event u^r_r.

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

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

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

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

$R_1(m_1', m_2')$ holds trivially since `unhide` has no changed the memories. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}', \nu_2')$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}', \nu_2')$ hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ holds; and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because after the transition α_1, the threads c_{r_e} and $c^2_{r_e}$ become the thread p. By inspecting the semantics for the scheduler, we have that $EL(\vec{c_1'}, \nu_1') = EL(\vec{c_2'}, \nu_2') = \emptyset$. By $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ and since we cannot apply $ii)$, we also know that $BEL(w_1, \nu_1) = BEL(w_2, \nu_2) = \emptyset$ and consequently $BEL(w_1', \nu_1') = BEL(w_2', \nu_2') = \emptyset$ since no thread is blocked by the transition α_1. Then, by taking $p' =$ `skip`, it trivially holds that $R_5(\vec{c_1}', w_1', \nu_1', \vec{c_2}', w_2', \nu_2', \texttt{skip})$. $R_6(\vec{c_1}', \nu_1')$ holds since $R_5(\vec{c_1}, w_1, \nu_1, \vec{c_2}, w_2, \nu_2, p)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds ; and by inspecting the type derivation of c_{r_e}. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have been not modified by the transition α_1. $R_8(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ holds since $EL(\vec{c_1}', \nu_1') = EL(\vec{c_2}', \nu_2') = \emptyset$. Finally, $R_9(\sigma_1', \nu_1', \sigma_2', \nu_2')$ holds since $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$ holds and by applying the definition of $\sim_L$. Propositions R_{10-17} hold since they hold already by hypothesis and since transition α_1 does not affect blocked threads.

□

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

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

CHAPTER

3

Security of Multithreaded Programs by Compilation

Extended version of
Security of Multithreaded Programs by Compilation,
In Proceedings of the 10th European Symposium on Research in Computer Security, Dresden, Germany, September 24-26, 2007, LNCS, Springer-Verlag.

Security of Multithreaded Programs by Compilation

Gilles Barthe[1], Tamara Rezk[2], Alejandro Russo[3], and Andrei Sabelfeld[3]

[1] IMDEA Software, Madrid, Spain
[2] Inria Sophia Antipolis and MSR-INRIA, France
[3] Chalmers University of Technology, Sweden

Abstract. Information security is a pressing challenge for mobile code technologies. In order to claim end-to-end security of mobile code, it is necessary to establish that the code neither intentionally nor accidentally propagates sensitive information to an adversary. Although mobile code is commonly multithreaded low-level code, the literature is lacking enforcement mechanisms that ensure information security for such programs. This paper offers a modular solution to the security of multithreaded programs. The modularity is three-fold: we give modular extensions of sequential semantics, sequential security typing, and sequential security-type preserving compilation that allow us enforcing security for multithreaded programs. Thanks to the modularity, there are no more restrictions on multithreaded source programs than on sequential ones, and yet we guarantee that their compilations are provably secure for a wide class of schedulers.

1 Introduction

Information security is a pressing challenge for mobile code technologies. Current security architectures provide no end-to-end security guarantees for mobile code: such code may either intentionally or accidentally propagate sensitive information to an adversary. However, recent progress in the area of language-based information-flow security [SM03] indicates that insecure flows in mobile code can be prevented by program analysis.
While much of existing work focuses on source languages, recent work has developed security analyses for increasingly expressive bytecode and assembly languages [BR05, GS05, MCB05, BPR07a, BRB07]. Given sensitivity annotations on inputs and outputs, these analyses provably guarantee noninterference [GM82], a property of programs that there are no insecure flows from sensitive inputs to public outputs.
It is, however, unsettling that information flow for multithreaded low-level programs has not been addressed so far. It is especially concerning because multithreaded bytecode is ubiquitous in mobile code scenarios. For example, multithreading is used for preventing screen lock-up in mobile applications [Mah04]. In general, creating a new thread for long and/or potentially blocking computation, such as establishing a network connection, is a much recommended pattern [Knu02].
This paper is the first to propose a framework for enforcing secure information flow for multithreaded low-level programs. We present an approach for deriving security-type systems that provably guarantee noninterference. On the code consumer side, these type systems can be used for checking the security of programs before running them.

Our solution goes beyond guarantees offered by security-type checking to code consumers. To this end, we have developed a framework for security-type preserving compilation, which allows code producers to derive security types for low-level programs from security types for source programs. This makes our solution practical for the scenario of untrusted mobile code. Moreover, even if the code is trusted (and perhaps even immobile), compilers are often too complex to be a part of the trusted computing base. Security-type preserving compilation removes the need to trust the compiler, because the type annotations of compiled programs can be checked directly at bytecode level.

The single most attractive feature of our framework is that security is guaranteed by source type systems that are no more restrictive than ones for sequential programs. This might be counterintuitive: there are covert channels in the presence of threads, such as internal timing channels [VS99], that do not arise in a sequential setting. Indeed, special primitives for interacting with the scheduler have been designed (e.g., [RS06a]) in order to control these channels. The pinnacle of our framework is that such primitives are automatically introduced in the compilation phase. This means that source-language programmers do not have to know about their existence and that there are no restrictions on dynamic thread creation at the source level. At the target level, the prevention of internal timing leaks does not introduce unexpected behaviors: the effect of interacting with the scheduler may only result in disallowing certain interleavings. Note that disallowing interleavings may, in general, affect the liveness properties of a program. Such a trade-off between between liveness and security is shared with other approaches (e.g., [SV98, VS99, Smi01, Smi03, RS06a]).

For an example of an internal timing leak, consider a simple two-threaded source-level program, where hi is a sensitive (high) and lo is a public (low) variable:

$$\texttt{if}\ hi\ \{\texttt{sleep}(100)\}; lo := 1 \parallel \texttt{sleep}(50); lo := 0$$

If hi is originally non-zero, the last command to assign to lo is likely to be $lo := 1$. If hi is zero, the last command to assign to lo is likely to be $lo := 0$. Hence, this program is likely to leak information about hi into lo. In fact, all of hi can be leaked into lo via the internal timing channel, if the timing difference is magnified by a loop (see, e.g., [RHNS07]).

In order for the timing difference of the thread that branches on hi not to make a difference in the interleaving of the assignments to lo, we need to ensure that the scheduler treats the first thread as "hidden" from the second thread: the second thread should not be scheduled until the first thread reaches the junction point of the `if`. We will show that the compiler enforces such a discipline for the target code so that the compilation of such source programs as above is free of internal timing leaks.

Our work benefits from modularity, which is three-fold. First, the framework has the ability to modularly extend sequential semantics. This grants us with language-independence from the sequential part. Further, the framework allows modular extensions of sequential security type systems. Finally, security type preserving compilation is also a modular extension of the sequential counterpart.

To illustrate the applicability of the framework, we instantiate it with some scheduler examples. These examples clarify what is expected of a scheduler to prevent internal timing leaks. Also, we give an instantiation of our framework for a simple assembly language that features an operand stack, conditions, and jumps. This instantiation is for illustration only, and we expect our results to apply to Java bytecode, for which mechanisms for tracking information flow by security-type systems were recently developed [BPR07a]. Being compatible with bytecode verification, our approach pushes the feasibility of replacing trust assumptions by type checking for mobile-code security one step further.

Figure 1 describes the process of producing and verifying a secure application, and illustrates that there is no need to trust the compiler nor the source type system. On the left of the figure, the code producer writes an application that is typable with respect to an information-flow type system, which needs to make no special provision for concurrency. The application is then transformed to bytecode thanks to an extended compiler that also produces a security environment, that assigns a security level to each program point, and additional information about the control structure of the program—in earlier works, we used control dependence regions for describing the control-flow information; here we use a more abstract representation, in the form of a next function. The consumer receives the compiled application, together with the security environment, the next function, and a type, and verifies that the next function satisfies some properties that are required for soundness, and that the program is typable w.r.t. the type system. If both verifications succeed, the execution of the program will not reveal information when executed with a secure scheduler. Thus, the TCB consists of the next checker, of the type system, and of the scheduler. Note that the next checker and the type system are not restricted to compiled programs; however, type-preserving compilation ensures that typable source programs are compiled into typable programs, and that the next function that is generated by the compiler is accepted by the next checker.

This paper revises and extends an earlier conference version [BRRS07] with proofs, explanations, and examples. The proofs that we have to exclude due to the lack of space can be found in the full electronic version [BRRS08].

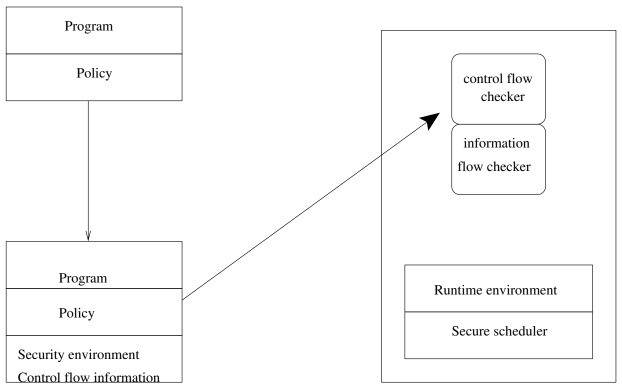

Fig. 1. Information-flow scenario

2 Syntax and semantics of multithreaded programs

This section sets the scene by defining the syntax and semantics for multithreaded programs. We introduce the notion of secure schedulers that help dealing with covert channels in the presence of multithreading.

Syntax and program structure Assume we have a set Thread of thread identifiers, a partially ordered set Level of security levels, a set $\mathsf{LocState}$ of local states and a set $\mathsf{GMemory}$ of global memories. The definition of programs is parameterized by a set of sequential instructions SeqIns. The set of all instructions extends SeqIns by a dynamic thread creation primitive $\mathsf{start}\ pc$ that spawns a new thread with a start instruction at program point pc.

Definition 1 (Program). *A program P consists of a set of program points $\mathcal{P}$, with a distinguished entry point 1 and a distinguished exit point exit, and an instruction map $\mathsf{insmap}_P : \mathcal{P} \setminus \{\mathrm{exit}\} \to \mathsf{Ins}$, where $\mathsf{Ins} = \mathsf{SeqIns} \cup \{\mathsf{start}\ pc\}$ with $pc \in \mathcal{P} \setminus \{\mathrm{exit}\}$. We sometimes write $P[i]$ instead of $\mathsf{insmap}_P\ i$.*

Each program has an associated successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$. The successor relation describes possible successor instructions in an execution. We assume that exit is the only program point without successors, and that any program point i s.t. $P[i] = \mathsf{start}\ pc$ is not branching, and has a single successor, denoted by $i + 1$ (if it exists); in particular, we do not require that $i \mapsto pc$. As common, we let $\mapsto^\star$ denote the reflexive and transitive closure of the relation $\mapsto$ (similar notation is used for other relations).

Definition 2 (Initial program points). *The set $\mathcal{P}_{init}$ of initial program points is defined as: $\{i \in \mathcal{P} \mid \exists j \in \mathcal{P},\ P[j] = \mathsf{start}\ i\} \cup \{1\}$.*

We assume the attacker level $k \in \mathsf{Level}$ partitions all elements of Level into *low* and *high* elements. Low elements are no more sensitive than k: an element ℓ is low if $\ell \leq k$. All other elements (including incomparable ones) are high. We assume that the set of high elements is not empty. This partition reduces the set Level to a two-element set $\{low, high\}$, where $low < high$, which we will adopt without loss of generality.
Programs come equipped with a *security environment* [BRB07] that assigns a security level to each program point and is used to prevent *implicit flows* [DD77]. The security environment is also used by the scheduler to select the thread to execute.

Definition 3 (Security environment, low, high, and always high program points).

1. *A security environment is a function $se : \mathcal{P} \to \mathsf{Level}$.*
2. *A program point $i \in \mathcal{P}$ is* low*, written $L(i)$, if $se(i) = low$;* high*, written $H(i)$, if $se(i) = high$; and* always high*, written $AH(i)$, if $se(j) = high$ for all points j such that $i \mapsto^\star j$.*

Semantics The operational semantics for multithreaded programs is built from an operational semantics for sequential programs and a scheduling function that picks the thread to be executed among the currently active threads. The scheduling function takes as parameters the current state, the execution history, and the security environment.

Definition 4 (State).

1. *The set* $\mathsf{SeqState}$ *of sequential states is a product* $\mathsf{LocState} \times \mathsf{GMemory}$ *of the local state* $\mathsf{LocState}$ *and global memory* $\mathsf{GMemory}$ *sets.*
2. *The set* $\mathsf{ConcState}$ *of concurrent states is a product* $(\mathsf{Thread} \rightharpoonup \mathsf{LocState}) \times \mathsf{GMemory}$ *of the partial-function space* $(\mathsf{Thread} \rightharpoonup \mathsf{LocState})$*, mapping thread identifiers to local states, and the set* $\mathsf{GMemory}$ *of global memories.*

It is convenient to use accessors to extract components from states: we use $s.\mathsf{lst}$ and $s.\mathsf{gmem}$ to denote the first and second components of a state s. Then, we use $s.\mathsf{act}$ to denote the set of active threads, i.e., $s.\mathsf{act} = \mathsf{Dom}(s.\mathsf{lst})$. We sometimes write $s(tid)$ instead of $s.\mathsf{lst}(tid)$ for $tid \in s.\mathsf{act}$. Furthermore, we assume given an accessor pc that extracts the program counter for a given thread from a local state.
We follow a concurrency model [RS06a] that lets the scheduler distinguish between different types of threads. A thread is *low* (resp., *high*) if the security environment marks its program counter as low (resp., high). A high thread is *always high* if the program point corresponding to the program counter is always high. A high thread is *hidden* if it is high but not always high. (Intuitively, the thread is hidden in the sense that the scheduler will, independently from the hidden thread, pick the following low threads.) Formally, we have the following definitions:

$$\begin{aligned} s.\mathsf{lowT} &= \{tid \in s.\mathsf{act} \mid L(s.\mathsf{pc}(tid))\} \\ s.\mathsf{highT} &= \{tid \in s.\mathsf{act} \mid H(s.\mathsf{pc}(tid))\} \\ s.\mathsf{ahighT} &= \{tid \in s.\mathsf{act} \mid AH(s.\mathsf{pc}(tid))\} \\ s.\mathsf{hidT} &= \{tid \in s.\mathsf{act} \mid H(s.\mathsf{pc}(tid)) \wedge \neg AH(s.\mathsf{pc}(tid))\} \end{aligned}$$

A scheduler treats different classes of threads differently. To see what guarantees are provided by the scheduler, it is helpful to foresee what discipline a type system would enforce for each kind of threads. From the point of view of the type system, a low thread becomes high while being inside of a branch of a conditional (or a body of a loop) with a high guard. Until reaching the respective junction point, the thread may not have any low side effects. In addition, until reaching the respective junction point, the high thread must be hidden by the scheduler: no low threads may be scheduled while the hidden thread is alive. This prevents the timing of the hidden thread from affecting the interleaving of low side effects in low threads. In addition, there are threads that are spawned inside of a branch of a conditional (or a body of a loop) with a high guard. These threads are always high: they may not have any low side effects. On the other hand, such threads do not have to be hidden in the same way: they can be interleaved with both low and high threads. Recall the example from Section 1. The intention is that the scheduler treats the first thread (which is high while it is inside the branch) as "hidden" from the second (low) thread: the second thread should not be scheduled until the first thread reaches the junction point of the `if`.
We proceed to defining computation history and secure schedulers, which operate on histories as parameters.

Definition 5 (History).

1. *A history is a list of pairs* (tid, ℓ) *where* $tid \in \mathsf{Thread}$ *and* $\ell \in \mathsf{Level}$. *We denote the empty history by* $\epsilon^H ist$.

2. *Two histories h and h' are indistinguishable, written $h \overset{Hist}{\sim} h'$, if $h|_{low} = h'|_{low}$, where $h|_{low}$ is obtained from h by projecting out pairs with the high level in the second component.*

We denote the set of histories by History. We now turn to the definition of a secure scheduler. The definition below is of a more algebraic nature than that of [RS06a], but captures the same intuition, namely that a secure scheduler: i) always picks an active thread; ii) chooses a high thread whenever there is one hidden thread; and iii) only uses the names and levels of low and the low part of histories to pick a low thread.

Definition 6 (Secure scheduler). *A secure scheduler is a function* $\mathsf{pickt} : \mathsf{ConcState} \times \mathsf{History} \rightharpoonup \mathsf{Thread}$, *subject to the following constraints, where* $s, s' \in \mathsf{ConcState}$ *and* $h, h' \in \mathsf{History}$*:*

1. *for every s such that* $s.\mathsf{lowT} \cup s.\mathsf{highT} \neq \emptyset$, $\mathsf{pickt}(s, h)$ *is defined, and* $\mathsf{pickt}(s, h) \in s.\mathsf{act}$*;*
2. *if* $s.\mathsf{hidT} \neq \emptyset$, *then* $\mathsf{pickt}(s, h) \in s.\mathsf{highT}$*; and*
3. *if* $h \overset{Hist}{\sim} h'$ *and* $s.\mathsf{lowT} = s'.\mathsf{lowT}$, *then* $\langle \mathsf{pickt}(s, h), \ell\rangle :: h \overset{Hist}{\sim} \langle \mathsf{pickt}(s', h'), \ell'\rangle :: h'$, *where* $\ell = se(s.pc(\mathsf{pickt}(s, h)))$ *and* $\ell' = se(s'.pc(\mathsf{pickt}(s', h')))$.

Example 1. Consider a round-robin policy: $\mathsf{pickt}(s, h) = rr(AT, last(h))$, where $AT = s.\mathsf{act}$, and the partial function $last(h)$ returns the identity of the most recently picked thread recorded in h (if it exists). Given a set of thread ids, an auxiliary function rr returns the next thread id to pick according to a round-robin policy. This scheduler is insecure because low threads can be scheduled even if a hidden thread is present, which violates req. 2 above.

Example 2. An example of a secure round-robin scheduler is defined below. The scheduler takes turns in picking high and low threads.

$$\mathsf{pickt}(s, h) = \begin{cases} rr(AT_L, last_L(h)), & \begin{array}{c}\text{if } h = \epsilon^H ist \text{ or}\\ h = (\mathsf{tid}, L).h' \text{ and } AT_H = \emptyset \text{ and } AT_L \neq \emptyset \text{ or}\\ h = (\mathsf{tid}, H).h' \text{ and } \mathsf{hidT} = \emptyset \text{ and } AT_L \neq \emptyset\end{array} \\ rr(AT_H, last_H(h)), & \begin{array}{c}\text{if } \mathsf{hidT} \neq \emptyset \text{ or}\\ h = (\mathsf{tid}, H).h' \text{ and } AT_L = \emptyset \text{ and } AT_H \neq \emptyset \text{ or}\\ h = (\mathsf{tid}, L).h' \text{ and } AT_H \neq \emptyset\end{array} \end{cases}$$

We assume that AT_L and AT_H are functions of s that extract the set of identifiers of low and high threads, respectively, and the partial function $last_\ell$ returns the identity of the most recently picked thread at level ℓ recorded in h, if it exists. The scheduler may only pick active threads (cf. req. 1). In addition to the alternation between high and low threads, the scheduler may only pick a low thread if there are no hidden threads (cf. req. 2). The separation into high and low threads ensures that for low-equivalent histories, the observable choices of the scheduler are the same (cf. req. 3). For simplicity, we have described a one-step secure scheduler. However, the definition above can be easily extended to schedulers where threads are scheduled for some fixed number of steps.

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s.pc(ctid) = i \quad P[i] \in \mathsf{SeqIns}\\ \langle s(ctid), s.\mathsf{gmem}\rangle \leadsto_{\text{seq}} \sigma, \mu \quad \sigma.\mathsf{pc} \neq \text{exit}\end{array}}{s, h \leadsto_{\text{conc}} s.[\mathsf{lst}(ctid) := \sigma, \mathsf{gmem} := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s.\mathsf{pc}(ctid) = i \quad P[i] \in \mathsf{SeqIns}\\ \langle s(ctid), s.\mathsf{gmem}\rangle \leadsto_{\text{seq}} \sigma, \mu \quad \sigma.\mathsf{pc} = \text{exit}\end{array}}{s, h \leadsto_{\text{conc}} s.[\mathsf{lst} := \mathsf{lst} \setminus ctid, \mathsf{gmem} := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s.\mathsf{pc}(ctid) = i \quad P[i] = \mathsf{start}\, pc\\ \mathsf{fresht}_{se(i)}(s) = ntid \quad s(ctid).[\mathsf{pc} := i+1] = \sigma'\end{array}}{s, h \leadsto_{\text{conc}} s.[\mathsf{lst}(ctid) := \sigma', \mathsf{lst}(ntid) := \lambda_{\text{init}}(pc)], \langle ctid, se(i)\rangle :: h}$$

Fig. 2. Semantics of multithreaded programs

$$\frac{P[i] \in \mathsf{SeqIns} \qquad i \vdash_{\text{seq}} \mathrm{S} \Rightarrow \mathrm{T}}{se, i \vdash \mathrm{S} \Rightarrow \mathrm{T}} \quad \frac{P[i] = \mathsf{start}\, pc \quad se(i) \leq se(pc)}{se, i \vdash \mathrm{S} \Rightarrow \mathrm{S}}$$

Fig. 3. Typing rules

To define the execution of multithreaded programs, we assume given a (deterministic) sequential execution relation $\leadsto_{\text{seq}} \subseteq \mathsf{SeqState} \times \mathsf{SeqState}$ that takes as input a current state and returns a new state, provided the current instruction is sequential.
We assume given a function $\lambda_{\text{init}} : \mathcal{P} \to \mathsf{LocState}$ that takes a program point and produces an initial state with program pointer pointing to pc. We also assume given a family of functions fresht_ℓ that takes as input a set of thread identifiers and generates a new thread identifier at level ℓ. We assume that the ranges of fresht_ℓ and $\mathsf{fresht}_{\ell'}$ are disjoint whenever $\ell \neq \ell'$. We sometimes use fresht_ℓ as a function from states to Thread.

Definition 7 (Multithreaded execution). *One step execution* $\leadsto_{\text{conc}} \subseteq (\mathsf{ConcState} \times \mathsf{History}) \times (\mathsf{ConcState} \times \mathsf{History})$ *is defined by the rules of Figure 2. We write* $s, h \leadsto_{\text{conc}} s', h'$ *when executing* s *with history* h *leads to state* s' *and history* h'.

The first two rules of Figure 2 correspond to non-terminating and terminating sequential steps. In the case of termination, the current thread is removed from the domain of lst. The last rule describes dynamic thread creation caused by the instruction $\mathsf{start}\ pc$. A new thread receives a fresh name $ntid$ from $\mathsf{fresht}_{se(i)}$ where $se(i)$ records the security environment at the point of creation. This thread is added to the pool of threads under the name $ntid$. All rules update the history with the current thread id and the security environment of the current instruction. The evaluation semantics of programs can be derived from the small-step semantics in the usual way. We let $main$ be the identity of the main thread.

Definition 8 (Evaluation semantics). *The evaluation relation* $\Downarrow_{\text{conc}} \subseteq (\mathsf{ConcState} \times \mathsf{History}) \times \mathsf{GMemory}$ *is defined by the clause* $s, h \Downarrow_{\text{conc}} \mu$ *iff* $\exists s', h'.\ s, h \leadsto^{\star}_{\text{conc}} s', h' \wedge s'.\mathsf{act} = \emptyset \wedge s'.\mathsf{gmem} = \mu$. *We write* $P, \mu \Downarrow_{\text{conc}} \mu'$ *as a shorthand for* $\langle f, \mu\rangle, \epsilon^{H} ist \Downarrow_{\text{conc}} \mu'$, *where* f *is the function* $\{\langle main, \lambda_{\text{init}}(1)\rangle\}$.

3 Security policy

Noninterference is defined relative to a notion of indistinguishability between global memories. For the purpose of this paper, it is not necessary to specify the definition of memory indistinguishability.

Definition 9 (Noninterfering program). *Let $\sim_g$ be an indistinguishability relation on global memories. A program P is* noninterfering *if for all memories $\mu_1, \mu_2, \mu_1', \mu_2'$:*

$$\mu_1 \sim_g \mu_2 \text{ and } P, \mu_1 \Downarrow \mu_1' \text{ and } P, \mu_2 \Downarrow \mu_2' \text{ implies } \mu_1' \sim_g \mu_2'$$

4 Type system

This section introduces a type system for multithreaded programs as an extension of a type system for noninterference for sequential programs. In Section 5, we show that the type system is sound for multithreaded programs, in that it enforces the noninterference property defined in the previous section. In Section 6, we instantiate the framework to a simple assembly language.

4.1 Assumptions on type system for sequential programs

We assume given a set LType of local types for typing local states, with a distinguished local type $\mathrm{T}_{\mathrm{init}}$ to type initial states, and a partial order $\leq$ on local types. Typing judgments in the sequential type system are of the form $se, i \vdash_{\mathrm{seq}} \mathrm{S} \Rightarrow \mathrm{T}$, where se is a security environment, i is a program point in program P, and S and T are local types. Typing rules are used to establish a notion of typable program [4], which ensures that runs of typable programs verify at each step the constraints imposed by the typing rules.

Definition 10 (Typable sequential program). *A sequential program P is typable w.r.t. type $\mathcal{S} : \mathcal{P} \to$ LType and security environment se, written $se, \mathcal{S} \vdash P$ if*

1. *$\mathcal{S}_1 = \mathrm{T}_{\mathrm{init}}$ (the initial program point is mapped to the initial local type); and*
2. *for all $i \in \mathcal{P}$ and $j \in \mathcal{P}$ $i \mapsto j$ implies that there exists $\mathrm{S} \in$ LType such that $se, i \vdash_{\mathrm{seq}} \mathcal{S}_i \Rightarrow \mathrm{S}$ and $\mathcal{S}_j \leq \mathrm{S}$,*

where we write $\mathcal{S}_i$ instead of $\mathcal{S}(i)$.

The sequential type system is assumed to satisfy further properties e.g. unwinding lemmas, that have already been established for some specific languages and that are formulated precisely in Section 5.

[4] The notion of typable sequential program is the same as typable multithreaded program, formally given later in this section.

4.2 Type system for multithreaded programs

The typing rules for the concurrent type system have the same form as those of the sequential type system and are given in Figure 3.

Definition 11 (Typable multithreaded program). *A concurrent program P is typable w.r.t. type $\mathcal{S} : \mathcal{P} \to \mathsf{LType}$ and security environment se, written $se, \mathcal{S} \vdash P$, if*

1. *$\mathcal{S}_i = \mathrm{T}_{\mathrm{init}}$ for all initial program points i of P (initial program point of main threads or spawn threads); and*
2. *for all $i \in \mathcal{P}$ and $j \in \mathcal{P}$: $i \mapsto j$ implies that there exists $\mathrm{s} \in \mathsf{LType}$ such that $se, i \vdash \mathcal{S}_i \Rightarrow \mathrm{s}$ and $\mathcal{S}_j \leq \mathrm{s}$.*

5 Soundness

The purpose of this section is to prove, under sufficient hypotheses on the sequential type system and assuming that the scheduler is secure, that typable programs are noninterfering. Formally, we want to prove that under suitable hypotheses (detailed below), the following theorem holds:

Theorem 1. *If the scheduler is secure and $se, \mathcal{S} \vdash P$, then P is noninterfering, provided Hypotheses 1-6 hold.*

Throughout this section, we assume that P is a typable program, i.e., $se, \mathcal{S} \vdash P$, and that the scheduler is secure. Moreover, we state some general hypotheses that are used in the soundness proofs. We revisit these hypotheses in Section 6 and show how they can be fulfilled.

State equivalence In order to prove noninterference, we rely on a notion of state equivalence. The definition is modular, in that it is derived from an equivalence between global memories $\sim_g$ and a partial equivalence relation $\sim_l$ between local states. (Intuitively, partial equivalence relations on local and global memories represent the observational power of the adversary.) In comparison to [BPR07a], equivalence between local states (operand stacks and program counters for the JVM) is not indexed by local types, since these can be retrieved from the program counter and the global type of the program.

Definition 12 (State equivalence). *Two concurrent states s and t are:*

1. *equivalent w.r.t. local states, written $s \overset{\mathsf{lmem}}{\sim} t$, iff $s.\mathsf{lowT} = t.\mathsf{lowT}$ and for every $tid \in s.\mathsf{lowT}$, we have $s(tid) \sim_l t(tid)$.*
2. *equivalent w.r.t. global memories, written $s \overset{\mathsf{gmem}}{\sim} t$, iff $s.\mathsf{gmem} \sim_g t.\mathsf{gmem}$.*
3. *equivalent, written $s \sim t$, iff $s \overset{\mathsf{gmem}}{\sim} t$ and $s \overset{\mathsf{lmem}}{\sim} t$.*

In order to carry out the proofs, we also need a notion of program counter equivalence between two states.

Definition 13. *Two states s and s' are pc-equivalent, written, $s \overset{pc}{\sim} s'$ iff $s.\mathsf{lowT} = t.\mathsf{lowT}$ and for every $tid \in s.\mathsf{lowT}$, we have $s.\mathsf{pc}(tid) = t.\mathsf{pc}(tid)$.*

Unwinding lemmas In this section, we formulate unwinding hypotheses for sequential instructions and extend them to a concurrent setting. Two kinds of unwinding statements are considered: a *locally respects unwinding result*, which involves two executions and is used to deal with execution in low environments, and a *step consistent unwinding result*, which involves one execution and is used to deal with execution in high environments. From now on, we refer to local states and global memories as λ and μ, respectively.

Hypothesis 1 (Sequential locally respects unwinding). *Assume* $\lambda_1 \sim_l \lambda_2$ *and* $\mu_1 \sim_g \mu_2$ *and* $\lambda_1.\mathsf{pc} = \lambda_2.\mathsf{pc}$. *If* $\langle \lambda_1, \mu_1 \rangle \leadsto_{\mathrm{seq}} \langle \lambda_1', \mu_1' \rangle$ *and* $\langle \lambda_2, \mu_2 \rangle \leadsto_{\mathrm{seq}} \langle \lambda_2', \mu_2' \rangle$, *then* $\lambda_1' \sim_l \lambda_2'$ *and* $\mu_1' \sim_g \mu_2'$.

In addition, we also need a hypothesis on the indistinguishability of initial local states.

Hypothesis 2 (Equivalence of local initial states). *For every initial program point* i, *we have* $\lambda_{\mathrm{init}}(i) \sim_l \lambda_{\mathrm{init}}(i)$.

We now extend the unwinding statement to concurrent states; note that the hypothesis $s'.\mathsf{lowT} = t'.\mathsf{lowT}$ is required for the lemma to hold. This excludes the case of a thread becoming hidden in an execution and not another (i.e., a high while loop).

Lemma 1 (Concurrent locally respects unwinding). *Assume* $s \sim t$ *and* $h_s \overset{Hist}{\sim} h_t$ *and* $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = ctid$ *and* $s.pc(ctid) = t.pc(ctid)$. *If* $s, h_s \leadsto_{\mathrm{conc}} s', h_{s'}$ *and* $t, h_t \leadsto_{\mathrm{conc}} t', h_{t'}$, *and* $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, *then* $s' \sim t'$ *and* $h_{s'} \overset{Hist}{\sim} h_{t'}$.

We now turn to the second, so-called step consistent, unwinding lemma. The lemma relies on the hypothesis that the current local memory is high, i.e., invisible by the attacker. Formally, highness is captured by a predicate $High^{\mathsf{lmem}}(\lambda)$ where λ is a local state.

Hypothesis 3 (Sequential step consistent unwinding). *Assume* $\lambda_1 \sim_l \lambda_2$ *and* $\mu_1 \sim_g \mu_2$. *Let* $\lambda_1.\mathsf{pc} = i$. *If* $\langle \lambda_1, \mu_1 \rangle \leadsto_{\mathrm{seq}} \langle \lambda_1', \mu_1' \rangle$ *and* $High^{\mathsf{lmem}}(\lambda_1)$ *and* $H(i)$, *then* $\lambda_1' \sim_l \lambda_2$ *and* $\mu_1' \sim_g \mu_2$.

Lemma 2 (Concurrent step consistent unwinding). *Assume* $s \sim t$ *and* $h_s \overset{Hist}{\sim} h_t$ *and* $\mathsf{pickt}(s, h) = ctid$ *and* $s.\mathsf{pc}(ctid) = i$ *and* $High^{\mathsf{lmem}}(s(ctid))$ *and* $H(i)$. *If* $s, h_s \leadsto_{\mathrm{conc}} s', h_{s'}$ *and* $s'.\mathsf{lowT} = t.\mathsf{lowT}$, *then* $s' \sim t$ *and* $h_{s'} \overset{Hist}{\sim} h_t$.

The proofs of the unwinding lemmas are by a case analysis on the semantics of concurrent programs.
In addition to the above assumptions, we also need another hypothesis stating that, under the assumptions of the concurrent locally respects unwinding lemma, either the executed instruction is a low instruction, in which case the program counter of the active thread remains equal after one step of execution, or that the executed instruction is a high instruction, in which case the active thread is hidden in one execution (high loop) or both (high conditional).

Hypothesis 4 (Preservation of pc equality). *Assume* $s \sim t$*;* $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = \mathit{ctid}$*;* $s(\mathit{ctid}).\mathsf{pc} = t(\mathit{ctid}).\mathsf{pc}$*;* $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$*; and* $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$*. Then,* $s'(\mathit{ctid}).\mathsf{pc} = t'(\mathit{ctid}).\mathsf{pc}$*; or* $H(s'(\mathit{ctid}).\mathsf{pc})$*; or* $H(t'(\mathit{ctid}).\mathsf{pc})$*.*

Note that the hypothesis is formulated w.r.t. concurrent states and concurrent execution. However, it is immediate to derive the above hypothesis from its restriction to sequential states and sequential execution.
We also need an hypothesis about visibility by the attacker:

Hypothesis 5 (High hypotheses).

1. *For every program point* i*, we have* $\mathit{High}^{\mathsf{lmem}}(\lambda_{\text{init}}(i))$*.*
2. *If* $\langle \lambda, \mu \rangle \rightsquigarrow_{\text{seq}} \langle \lambda', \mu' \rangle$ *and* $\mathit{High}^{\mathsf{lmem}}(\lambda)$ *and* $H(\lambda.\mathsf{pc})$ *then* $\mathit{High}^{\mathsf{lmem}}(\lambda')$*.*
3. *If* $\mathit{High}^{\mathsf{lmem}}(\lambda_1)$ *and* $\mathit{High}^{\mathsf{lmem}}(\lambda_2)$ *then* $\lambda_1 \sim_l \lambda_2$*.*

The next *function* Finally, the soundness proof relies on the existence of a function next that satisfies several properties. Intuitively, next computes for any high program point its minimal observable successor, i.e., the first program point with a low security level reachable from it. If executing the instruction at program point i can result in a hidden thread (high if or high while), then $\mathsf{next}(i)$ is the first program point such that $i \mapsto^\star \mathsf{next}(i)$ and the thread becomes visible again. The existence of the next function is closely related to control dependence regions, which are discussed in Section 6.1.

Hypothesis 6 (Existence of next **function).** *There exists a function* $\mathsf{next} : \mathcal{P} \rightharpoonup \mathcal{P}$ *such that the* next *properties (NeP) hold:*

$\mathbf{NePd})$ $\mathsf{Dom}(\mathsf{next}) = \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\, i \mapsto^\star j \wedge \neg H(j)\}$
$\mathbf{NeP1})$ $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = \mathsf{next}(j)$
$\mathbf{NeP2})$ $i \in \mathsf{Dom}(\mathsf{next}) \wedge j \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = j$
$\mathbf{NeP3})$ $j, k \in \mathsf{Dom}(\mathsf{next}) \wedge i \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = \mathsf{next}(k)$
$\mathbf{NeP4})$ $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge k \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = k$

Intuitively, properties **NeP1**, **NeP2**, and **NeP3** ensure that the next of instructions within an outermost high conditional statement coincides with the junction point of the conditional; in addition, properties **NeP1**, **NeP2**, and **NeP4** ensure that the next of instructions within an outermost high loop coincides with the exit point of the loop.

6 Instantiation

In this section, we apply our main results to a simple assembly language with conditional jumps and dynamic thread creation. We present the assembly language with a semantics and a type system for noninterference but without considering concurrent primitives and plug these definitions into the framework for multithreading. Then, we present a compilation function from a simple while-language with dynamic thread creation into assembly code. The source and target languages are defined in Figure 4. The compilation function allows us to easily define control dependence regions and junction points in the target code. Function next is then defined using that information. Moreover, we prove that the obtained definition of next satisfies the properties required in Section 5. Finally, we conclude with a discussion about how a similar instantiation can be done for the JVM.

$e ::= x \mid n \mid e\ op\ e \qquad c ::= x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{fork}(c)$

$instr ::=$	$\mathsf{binop}\ op$	binary operation on stack
$\mid$	$\mathsf{push}\ n$	push value on top of stack
$\mid$	$\mathsf{load}\ x$	load value of x on stack
$\mid$	$\mathsf{store}\ x$	store top of stack in variable x
$\mid$	$\mathsf{ifeq}\ j$	conditional jump
$\mid$	$\mathsf{goto}\ j$	unconditional jump
$\mid$	$\mathsf{start}\ j$	creation of a thread

where $op \in \{+, -, \times, /\}$, $n \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}$.

Fig. 4. Source and target language

$$\frac{P[i] = \mathsf{push}\ n}{se, i \vdash_{\text{seq}} st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{se, i \vdash_{\text{seq}} k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma(x)}{se, i \vdash_{\text{seq}} k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{se, i \vdash_{\text{seq}} st \Rightarrow (\Gamma(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{se, i \vdash_{\text{seq}} st \Rightarrow st} \qquad \frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{reg}(i),\ k \leq se(j')}{se, i \vdash_{\text{seq}} k :: st \Rightarrow \mathsf{lift}_k(st)}$$

Fig. 5. Transfer rules

6.1 Sequential part of the language

The instantiation requires us to define the semantics and a type system to enforce non-interference for the sequential primitives of the language. On the semantics side, we assume that a local state is a pair $\langle os, pc\rangle$ where os is an operand stack, i.e., a stack of values, and pc is a program counter, whereas a global state μ is a map from variables to values. The operational semantics is standard and therefore we omit it. We also define $\lambda_{\text{init}}(pc)$ to be the local state $\langle \epsilon, pc\rangle$, where ϵ is the empty operand stack.

The enforcement mechanism consists of local types which are stacks of security levels, i.e., $\mathsf{LType} = \mathsf{Stack}(\mathsf{Level})$; we let T_{init} be the empty stack of security levels. Typing rules are summarized in Figure 5, where $\mathsf{lift}_k(st)$ denotes the point-wise extension of $\lambda k'.\ k \sqcup k'$ to stacks of security levels, and $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ denotes the region of branching points. We express the chosen security policy by assigning a security level $\Gamma(x)$ to each variable x.

The definition of state equivalence is inspired from [BR05]: two global memories are indistinguishable iff they coincide on all low variables. Equivalence between local memories is defined relative to a mapping of program points to stack types, using the notion of operand stack indistinguishability used in [BR05]. Formally, we instantiate the definitions of local and global state equivalence, and of high stacks as follows:

$$\begin{array}{rcl}\langle os, pc\rangle \sim_l \langle os', pc'\rangle & \Longleftrightarrow & os \overset{\mathsf{os}}{\sim}_{\mathrm{S}(pc),\mathrm{S}(pc')} os' \\ \mu_1 \sim_g \mu_1' & \Longleftrightarrow & \mu_1 \overset{\mathsf{vmap}}{\sim} \mu_1' \\ High^{\mathsf{lmem}}(\langle os, pc\rangle) & \Longleftrightarrow & \mathsf{highos}(os, \mathrm{S}(pc))\end{array}$$

where $\overset{\mathsf{os}}{\sim}$, $\overset{\mathsf{vmap}}{\sim}$, and highos are defined as in [BR05]:

- $\mu_1 \overset{\mathsf{vmap}}{\sim} \mu_1'$ iff $\mu_1(x) = \mu_1'(x)$ for all $x \in \mathcal{V}$ such that $\Gamma(x) \leq k$
- $\mathsf{highos}(os, \mathrm{S})$ iff os and S have the same length n, and $\mathrm{S}(i) \not\leq k$ for all $1 \leq i \leq n$
- $s \overset{\mathsf{os}}{\sim}_{\mathrm{S},\mathrm{S}'} s'$ is defined by the clauses

$$\frac{\mathsf{highos}\ (s, \mathrm{S}) \qquad \mathsf{highos}\ (s', \mathrm{S}')}{s \overset{\mathsf{os}}{\sim}_{\mathrm{S},\mathrm{S}'} s'} \qquad\qquad \frac{s \overset{\mathsf{os}}{\sim}_{\mathrm{S},\mathrm{S}'} s' \qquad v \sim_k v'}{v :: s \overset{\mathsf{os}}{\sim}_{k::\mathrm{S},k::\mathrm{S}'} v' :: s'}$$

- state equivalence $s \overset{\bullet}{\sim}_{S,S'} s'$ is defined as

$$os \overset{\mathsf{os}}{\sim}_{S,S'} os' \wedge \mu \overset{\mathsf{vmap}}{\sim} \mu'$$

  assuming $s = \langle\langle os, i\rangle, \mu\rangle$ and $s' = \langle\langle os', i'\rangle, \mu'\rangle$.

We conclude this section by showing that all hypotheses, except Hypotheses 6, follow immediately from definitions, or from the results of [BR05]. Note that the latter rely on some assumptions about control dependence regions in programs. Essentially, these regions represent an over-approximation of the range of branching points. This concept is formally introduced by the functions $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ and $\mathsf{jun} : \mathcal{P} \rightharpoonup \mathcal{P}$, which respectively compute the control dependence region and the junction point for a given instruction. Both functions need to satisfy some properties in order to guarantee noninterference in typable programs. These properties, which are known as SOAP properties [BR05], are given in the Appendix, and can be guaranteed by compilation.

Lemma 3. *Hypotheses 1, 2, 3, 4 and 5 hold for all typable programs.*

6.2 Concurrent extension

As shown in Definition 7, the concurrent semantics is obtained from the semantics for sequential commands together with a transition for the instruction `start`. Moreover, the sequential type system in Figure 5 is extended by the typing rules presented in Figure 3 to consider concurrent programs.
The proof of noninterference for concurrent programs relies on the existence of the function next. Similarly to the technique of [BRN06], we name program points where control flow can branch or writes can occur. We add natural number labels to the source language as follows:

$$c ::= [x := e]^n \mid c; c \mid [\mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c]^n \mid [\mathtt{while}\ e\ \mathtt{do}\ c]^n \mid [\mathtt{fork}(c)]^n$$

$$\mathcal{E}(x) = \mathsf{load}\ x \quad \mathcal{E}(n) = \mathsf{push}\ n \quad \mathcal{E}(e\ op\ e') = \mathcal{E}(e) :: \mathcal{E}(e') :: \mathsf{binop}\ op$$

$$\begin{aligned}
\mathcal{S}(x := e,\ T) &= (\mathcal{E}(e) :: \underline{\mathsf{store}\ x},\ T)\\
\mathcal{S}(c_1;\ c_2,\ T) &= \mathit{let}\ (lc_1,\ T_1) = \mathcal{S}(c_1,\ T); (lc_2,\ T_2) = \mathcal{S}(c_2,\ T_1);\\
&\quad\ \mathit{in}\ \ (lc_1 :: lc_2,\ T_2)\\
\mathcal{S}(\mathsf{while}\ e\ \mathsf{do}\ c,\ T) &= \mathit{let}\ le = \mathcal{E}(e); (lc,\ T') = \mathcal{S}(c,\ T);\\
&\quad\ \mathit{in}\ \ (\mathsf{goto}\ (pc + \#lc + 1) :: lc :: le :: \underline{\mathsf{ifeq}\ (pc - \#lc - \#le)},\\
&\qquad T')\\
\mathcal{S}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\ T) &= \mathit{let}\ le = \mathcal{E}(e); (lc_1,\ T_1) = \mathcal{S}(c_1,\ T); (lc_2,\ T_2) = \mathcal{S}(c_2,\ T_1);\\
&\quad\ \mathit{in}\ \ (le :: \underline{\mathsf{ifeq}\ (pc + \#lc_2 + 2)} :: lc_2 :: \mathsf{goto}\ (pc + \#lc_1 + 1) ::\\
&\qquad lc_1,\ T_2)\\
\mathcal{S}(\mathsf{fork(c)},\ T) &= \mathit{let}\ (lc,\ T') = \mathcal{S}(c,\ T); \mathit{in}\ (\underline{\mathsf{start}\ (\#T' + 2)},\ T' :: lc :: \mathsf{return})\\
\mathcal{C}(c) &= \mathit{let}\ (lc,\ T) = \mathcal{S}(c,\ []); \mathit{in}\ \mathsf{goto}\ (\#T + 2) :: T :: lc :: \mathsf{return}
\end{aligned}$$

Fig. 6. Compilation function

This labeling allows us to define control dependence regions for the source code and use this information to derive control dependence regions for the assembly code. We introduce two functions, sregion and tregion, to deal with control dependence regions in the source and target code, respectively.

Definition 14 (function sregion**).** *For each branching command $[c]^n$, $sregion(n)$ is defined as the set of labels that are inside of the command c except for those ones that are inside of* `fork` *commands.*

As in [BRN06], control dependence regions for low-level code are defined based on the function sregion and a compilation function. For a complete source program c, we define the compilation $\mathcal{C}(c)$ in Figure 6. We use symbol $\#$ to compute the length of lists. Symbol :: is used to insert one element to a list or to concatenate two existing lists. The current program point in a program is represented by pc. The function $\mathcal{C}(c)$ calls the auxiliary function $\mathcal{S}$ which returns a pair of programs. The first component of that pair stores the compiled code of the main program, while the second one stores the compilation code of spawned threads. We now define control dependence regions for assembly code and respective junction points.

Definition 15 (function tregion**).** *For a branching instruction $[c]^n$ in the source code, $tregion(n)$ is defined as the set of instructions obtained by compiling the commands $[c']^{n'}$, where $n' \in sregion(n)$. Moreover, if c is a while loop, then $n \in$* tregion(n) *as well as the instructions obtained from compiling the guard of the loop. Otherwise, the* goto *instruction after the compilation of the else-branch also belongs to* tregion(n).

Junction points are computed by the function jun. The domain of this function consist of every branching point in the program. We define jun as follows:

Definition 16 (junction points). *For every branching point $[c]^n$ in the source program, we define* $\mathsf{jun}(n) = max\{i | i \in \mathsf{tregion}(n)\} + 1$.

$$\frac{\vdash e : L \qquad \vdash_\alpha c \,:\, E \qquad E(n) = F(n) = \phi(\alpha)}{\vdash_\alpha [\textsf{while}\ e\ \textsf{do}\ c]^n_\alpha \,:\, E, F}$$

$$\frac{\vdash e : L \qquad \vdash_\alpha c : E, F \qquad \vdash_\alpha c' : E, F \qquad E(n) = F(n) = \phi(\alpha)}{\vdash_\alpha [\textsf{if}\ e\ \textsf{then}\ c\ \textsf{else}\ c']^n_\alpha \,:\, E, F}$$

$$\frac{\vdash e : H \qquad \vdash_\bullet c \,:\, E, F \qquad E(n) = F(n) = H}{\vdash_\bullet [\textsf{while}\ e\ \textsf{do}\ c]^n_\bullet \,:\, E, F}$$

$$\frac{\vdash e : H \qquad \vdash_\bullet c : E, F \qquad \vdash_\bullet c' : E, F \qquad E(n) = F(n) = H}{\vdash_\bullet [\textsf{if}\ e\ \textsf{then}\ c\ \textsf{else}\ c']^n_\bullet \,:\, E, F}$$

$$\frac{\vdash_\alpha c : E, F \qquad \vdash_\alpha c' : E, F}{\vdash_\alpha c; c' \,:\, E, F} \qquad\qquad \frac{\vdash_\alpha c : E, F \qquad E(n) = F(n) = \phi(\alpha)}{\vdash_\alpha [\textsf{fork}(c)]^n_\alpha \,:\, E, F}$$

ASSIGN

$$\frac{\vdash e : k \qquad k \sqcup E(n) \leq \Gamma(x) \qquad E(n) = F(n) = \phi(\alpha)}{\vdash_\alpha [x := e]^n_\alpha \,: E, F}$$

TOP-H-WHILE

$$\frac{\vdash e : H \qquad \vdash_\bullet c \,:\, E, F \qquad E(n) = L \qquad F(n) = H}{\vdash_\circ [\textsf{while}\ e\ \textsf{do}\ c]^n_\bullet \,:\, E}$$

TOP-H-COND

$$\frac{\vdash e : H \qquad \vdash_\bullet c : E, F \qquad \vdash_\bullet c' : E, F \qquad E(n) = L \qquad F(n) = H}{\vdash_\circ [\textsf{if}\ e\ \textsf{then}\ c\ \textsf{else}\ c']^n_\bullet \,:\, E, F}$$

Fig. 7. Intermediate typing rules for high-level language commands

Having defined control dependence regions and junction points for low-level code, we proceed to defining next. Intuitively, next is only defined for instructions that belong to regions corresponding to the outermost branching points whose guards involved secrets. For every instruction i inside of an outermost branching point $[c]^n$, we define $\textsf{next}(i) = \textsf{jun}(n)$. Observe that this definition captures the intuition about next given in the beginning of Section 5. However, it is necessary to know, for a given program, what are the outermost branching points whose guards involved secrets. With this in mind, we extend one of the type systems given in [BRN06] to identify such points. We add some rules for outermost branching points that involved secrets together with some extra notations to know when a command is inside of one of those points or not.

A source program c is typable, written $\vdash_\circ c : E, F$, if its command part is typable with respect to E and F according to the rules given in Figure 7. The typing judgment has the form $\vdash_\alpha [c]^n_{\alpha'} \,:\, E, F$, where E and F are functions from labels to security levels. Function E and F play the role of security environment for the source code which easily allows to define the security environment for the target code (see Definition 24 in Appendix). Specifically, functions E and F help to determine the security environment for the guards and commands involved in branching points, respectively. We then omit writing E and F in type judgments when expressing properties only related with source code. Variable α denotes if c is part of a branching instruction that branches on secrets ($\bullet$) or public data ($\circ$). Variable α' represents the level of the guards in branching instructions. Function ϕ is defined as follows: $\phi(\bullet) = H$ and $\phi(\circ) = L$. The most interesting rules are $TOP-H-COND$ and $TOP-H-WHILE$. These rules can be only applied when the branching commands are the outermost ones and when they branch on secrets. Observe that such commands are the only ones that are typable considering $\alpha = \circ$ and $\alpha' = \bullet$. Moreover, the type system prevents *explicit* (via assignment) and *implicit* (via control) flows [DD77]. To this end, the type system enforces the same constraints as standard security type systems for sequential languages (e.g., [VSI96]). Explicit flows are prevented by rule $ASSIGN$, while implicit flows are ruled out by demanding a security environment of level H inside of commands that branch on secrets. The type system guarantees information-flow security at the same time as it identifies the outermost commands that branch on secrets. Function next is defined as follows:

Definition 17 (function next). *For every branching point c in the source program such that $\vdash_\circ [c]^n_\bullet$, we have that $\forall k \in \mathsf{tregion}(n).\mathsf{next}(k) = \mathsf{jun}(n)$.*

This definition satisfies the properties from Section 5, as shown by the following lemma.

Theorem 2. *Definition 17 satisfies properties **NePd** and **NeP1–4**.*

Notice that one does not need to trust the compiler in order to verify that properties **NePd** and **NeP1–4** are satisfied. Indeed, these properties are intended to be checked independently from the compiler by code consumers. We are now in condition to show the soundness of the instantiation.

Corollary 1 (Soundness of the instantiation). *The derived type system guarantees noninterference for multithreaded assembly programs.*

6.3 Compilation example

In this section, we illustrate our approach through an example. Specifically, we show the steps that need to be taken by code producers before they hand over the code to consumers for security type checking.

Section 1 has an example of internal-timing leaks by a command that branches on secrets and, depending on which branch is taken, takes different timing behavior. Based on this example, consider the following program:

$$\begin{array}{l}\texttt{fork}(hi' := 0;\ hi' := 0;\ lo := 0);\\ \texttt{if}\ hi\ \texttt{then}\ hi' := 0;\ hi' := 0;\ hi' := 0\ \texttt{else}\ hi' := 0;\\ lo := 1\end{array}$$

Variables hi and hi' store secret data while lo stores public information. Assuming a one-step round-robin scheduler, the last command to assign lo is $lo := 1$ when hi is true; and $lo := 0$ when hi is false. As the example in Section 1, this program suffers from internal-timing leaks. We show how our approach prevents them.

Firstly, the producer compiles the program by applying the compilation function $\mathcal{C}$ in Figure 6 to obtain:

```
1 goto 9          9  start 2       17 push 0
2 push 0          10 load hi       18 store hi'
3 store hi'       11 ifeq 15       19 push 0
4 push 0          12 push 0        20 store hi'
5 store hi'       13 store hi'     21 push 1
6 push 0          14 goto 21       22 store lo
7 store lo        15 push 0        23 return
8 return          16 store hi'
```

Instructions 2–8 result from compiling the body of the `fork` command, while instructions 10–22 are obtained by compiling the main thread. Instructions 1, 9, and 23 properly plug together the code corresponding to the generated threads.

Secondly, the producer labels the source code in such a way that *primary instructions* [BRN06], which are underlined in the compiled code, match the instruction that generated them at the source level. More specifically, the source code is labeled as follows:

$[\texttt{fork}([hi' := 0]^3;\ [hi' := 0]^5;\ [lo := 0]^7)]^9;$
$[\texttt{if}\ hi\ \texttt{then}\ [hi' := 0]^{16};\ [hi' := 0]^{18};\ [hi' := 0]^{20}\ \texttt{else}\ [hi' := 0]^{13}]^{11};$
$[lo := 1]^{22}$

The program has only one branching point. The producer consequently obtains that $\mathsf{sregion}(11) = \{16, 18, 20, 13\}$, $\mathsf{tregion}(11) = \{12, 13, 14, 15, 16, 17, 18, 19, 20\}$, and $\mathsf{jun}(11) = 21$ by applying Definitions 14, 15, and 16, respectively.

Thirdly, the producer applies the intermediate type system in Figure 7 to the labeled source code in order to obtain the following definitions for functions E and F:

labels	E	F	**labels**	E	F	**labels**	E	F	**labels**	E	F
3	L	L	9	L	L	16	H	H	12	L	L
5	L	L	11	L	H	18	H	H			
7	L	L	13	H	H	20	H	H			

By Definition 24 (see Appendix), the security environment se for the compiled code is determined as follows.

instr.	se	**instr.**	se	**instr.**	se	**instr.**	se
1	L	7	$E(7)$	13	$E(13)$	19	$E(20)$
2	$E(3)$	8	L	14	$F(11)$	20	$E(20)$
3	$E(3)$	9	$E(9)$	15	$E(16)$	21	$E(22)$
4	$E(4)$	10	$E(11)$	16	$E(16)$	22	$E(22)$
5	$E(4)$	11	$E(11)$	17	$E(18)$	23	L
6	$E(7)$	12	$E(13)$	18	$E(18)$		

Observe that the instructions inside of the `if-then-else` command have H as their security environment. To illustrate it, we show the compiled code where instructions with H as their security environment are marked with gray. Unmarked instructions have security environment defined as L.

```
1 goto 9       7  store lo   13 store hi'   19 push 0
2 push 0       8  return     14 goto 21     20 store hi'
3 store hi'    9  start 2    15 push 0      21 push 1
4 push 0       10 load hi    16 store hi'   22 store lo
5 store hi'    11 ifeq 15    17 push 0      23 return
6 push 0       12 push 0     18 store hi'
```

Fourthly and lastly, the producer obtains function next by applying Definition 17. Specifically, $\forall k \in \mathsf{tregion}(11).\mathsf{next}(k) = \mathsf{jun}(11) = 21$ since $\vdash_\circ [\texttt{if}\ hi\ \texttt{then}\ hi' := 0;\ hi' := 0;\ hi' := 0\ \texttt{else}\ hi' := 0;]^{11}_{\bullet}$ by the type derivation of the program on the intermediate type system.
At this point, the producer obtains all the necessary information to accompany the code for the consumer: the compiled code, se, and next. Then, the consumer proceeds to check that the compiled code type-checks with respect to se and that the next satisfies the properties described in Hypothesis 6. If the producer generates se and next following the described approach, then the consumer will succeed with the consumer-side checks (see Corollary 1).

6.4 Type preserving compilation

The compilation of sequential programs is type-preserving, as shown in previous work [BRN06]. Our framework allows extending type-preservation to multithreading. Moreover, it enables us to obtain a key *non-restrictiveness* result: although the source-level type system is no more restrictive than a typical type system for a sequential language (e.g., [VSI96]), the compilation of (possibly multithreaded) typable programs is guaranteed to be typable at low-level. Due to the lack of space, we only give an instantiation of this result to the source and target languages of this section:

Theorem 3. *For a given source-level program c, assume $nf(c)$ is obtained from c by replacing all occurrences of $\texttt{fork}(d)$ by d. If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [VSI96] then $se, \mathcal{S} \vdash \mathcal{C}(c)$ for some se and $\mathcal{S}$.*

This theorem and Theorem 1 entail the following corollary:

Corollary 2. *If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [VSI96] then $\mathcal{C}(c)$ is secure.*

Java Virtual Machine The modular proof technique developed in the previous section is applicable to a Java-like language. If the sequential type system is compatible with bytecode verification, then the concurrent type system is also compatible with it. This implies that Java bytecode verification can be extended to perform security type checking. Note that the definition of a secure scheduler is compatible with the JVM, where the scheduler is mostly left unspecified. Moreover, it is possible to, in effect, override an arbitrary scheduler from any particular implementation of JVM with a secure scheduler that keeps track of high and low threads as a part of an application's own state (cf. [TRH07]).
However, some issues arise in the definition of a concurrent JVM: in particular, we cannot adapt the semantics and results of [BPR07a] directly, because the semantics of

method calls is big-step. Instead, we must rely on a more standard semantics where states include stack frames, and prove unwinding lemmas for such a semantics; fortunately, the technical details in [BR05] took this route, and the same techniques can be used here.

Another point is that the semantics of the multithreaded JVM obtained by the method described in Section 2 only partially reflects the JVM specification. In particular, it ignores object locks, which are used to perform synchronization throughout program execution. Dealing with synchronization is a worthwhile topic for future work.

7 Related work

Information-flow type systems for low-level languages, including JVML, and their relation to information-flow type systems for structured source languages, have been studied by several authors [BR05, GS05, MCB05, BRN06, BPR07a, BRB07]. Nevertheless, the present work provides, to the best of our knowledge, the first proof of noninterference for a concurrent low-level language, and the first proof of type-preserving compilation for languages with concurrency.

This work exploits recent results on interaction between the threads and the scheduler [RS06a] in order to control internal timing leaks. The interaction is modeled by *hide* and *unhide* primitives that communicate to the scheduler whether a thread's timing behavior should be "hidden". In this paper, there is no need for explicit hide/unhide primitives because scheduler is driven by the security environment. If a thread is inside of a conditional with a high guard, then it executes in a high security environment and thus its timing behavior is automatically hidden from threads that run in a low security environment.

Other approaches [SV98, VS99, Smi01, Smi03] to handling internal timing rely on $\texttt{protect}(c)$ which, by definition, hides the internal timing of command c. It is not clear how to implement $\texttt{protect}()$ without modifying the scheduler (unless the scheduler is cooperative [RS06b, TRH07]). It is possible to prevent internal timing leaks by spawning dedicated threads for computations that involve secrets and carefully synchronizing the resulting threads [RHNS07]. However, this implies high synchronization costs. Yet other approaches prevent internal timing leaks in code by disallowing any races on public data [ZM03, HWS06, Ter08]. However, they wind up rejecting such innocent programs as $lo := 0 \parallel lo := 1$ where lo is a public variable. Still other approaches prevent internal timing by disallowing low assignments after high branching [BC02, Alm06]. Less related work [Aga00, SS00, Sab01, SM02, KM06] considers external timing, where an attacker can use a stopwatch to measure computation time. This work considers a more powerful attacker, and, as a price paid for security, disallows loops branching on secrets.

Further afield, different flavors of possibilistic noninterference have been explored in process-calculus settings [HVY00, FG01, Rya01, HY02, Pot02], but without considering the impact of scheduling. Most recently, van der Meyden and Zhang [vZ08] have investigated how the choice of a scheduler can affect security definitions in an abstract automata-based setting. However, they leave enforcement mechanisms and treatment of dynamic thread creation unaddressed. For additional related work, we refer to an overview of language-based information-flow security [SM03].

8 Conclusions

We have presented a framework for controlling information flow in multithreaded low-level code. Thanks to its modularity and language-independence, we have been able to reuse several results for sequential languages. An appealing feature enjoyed by the framework is that security-type preserving compilation is no more restrictive for programs with dynamic thread creation than it is for sequential programs. Primitives for interacting with the scheduler are introduced by the compiler behind the scenes, and in such a way that internal timing leaks are prevented.
We have demonstrated an instantiation of the framework to a simple imperative language and have argued that our approach is amenable to extensions to object-oriented languages. The compatibility with bytecode verification makes our framework a promising candidate for establishing mobile-code security via type checking.

Acknowledgment

This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905.

References

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

[Alm06] A. Almeida Matos. *Typing secure information flow: declassification and mobility*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2006.

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

[BPR07a] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer, 2007.

[BPR07b] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. Technical report, INRIA, 2007. Extended version of [BPR07a].

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.

[BRB07] G. Barthe, T. Rezk, and A. Basu. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*, 2007.

[BRN06] Gilles Barthe, Tamara Rezk, and David Naumann. Deriving an information flow checker and certifying compiler for java. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 230–242. IEEE Computer Society, 2006.

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

[BRRS08] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multi-threaded programs by compilation. Technical report, May 2008. Located at `http://www.cs.chalmers.se/~russo/tissecfull.pdf`.

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

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

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

[GS05] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proceedings of VMCAI'05*, volume 3385 of *LNCS*, pages 346–362. Springer-Verlag, 2005.

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

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

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

[KM06] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*, pages 47–62. Springer-Verlag, July 2006.

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

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

[MCB05] R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In M. Coppo, E. Lodi, and G.M. Pinna, editors, *Proceedings of ICTCS 2005*, volume 3701 of *LNCS*, pages 360–374. Springer-Verlag, 2005.

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

[Rez06] T. Rezk. *Verification of confidentiality policies for mobile code*. PhD thesis, Université de Nice Sophia-Antipolis, 2006.

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

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

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

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

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.
[SM02] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.
[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
[Smi01] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.
[Smi03] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.
[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.
[Ter08] T. Terauchi. A type system for observational determinism. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.
[TRH07] Ta Chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007. To appear.
[VS99] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.
[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
[vZ08] R. van der Meyden and C. Zhang. Information flow in systems with schedulers. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.
[ZM03] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

9 Appendix

Proof of soundness of concurrent type system

Proofs of unwinding lemmas

Lemma 1 (Concurrent locally respects unwinding). Assume $s \sim t$ and $h_s \overset{Hist}{\sim} h_t$ and $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = ctid$ and $s.pc(ctid) = .pc(t)ctid$. If $s, h_s \rightsquigarrow_{\mathrm{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\mathrm{conc}} t', h_{t'}$, and $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, then $s' \sim t'$ and $h_{s'} \overset{Hist}{\sim} h_{t'}$.

Proof. We only prove that $s' \sim t'$, since $h_{s'} \overset{Hist}{\sim} h_{t'}$ is a direct consequence of the hypotheses and of the definition of secure scheduler. We distinguish two cases:

1. The instruction to be executed is a sequential instruction. By definition of the semantics: $\langle s(ctid), s.\mathsf{gmem}\rangle \rightsquigarrow_{\mathrm{seq}} \langle s'(ctid), s'.\mathsf{gmem}\rangle$ and $\langle t(ctid), t.\mathsf{gmem}\rangle \rightsquigarrow_{\mathrm{seq}} \langle t'(ctid), t'.\mathsf{gmem}\rangle$. By hypothesis, we have $s \overset{\mathsf{gmem}}{\sim} t$ and $s(ctid) \sim_l t(ctid)$. Thus

by the sequential LR unwinding hypothesis, we have $s' \overset{\mathsf{gmem}}{\sim} t'$ and $s'(ctid) \sim_l t'(ctid)$. Since $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, we conclude that $s' \overset{\mathsf{lmem}}{\sim} t'$ and hence $s' \sim t'$ by definition of state equivalence.

2. The instruction to be executed is of the form $\mathsf{start}\ pc$. By the hypotheses and the definition of state equivalence, it is sufficient to show that $\mathsf{fresht}_\sigma(s) = \mathsf{fresht}_\sigma(t) = ntid$, where $se(pc) = \sigma$ and $\lambda_{\mathrm{init}}(pc) \sim_l \lambda_{\mathrm{init}}(pc)$. This follows from equivalence of local initial states.

□

Lemma 2 (Concurrent step consistent unwinding). Assume $s \sim t$ and $h_s \overset{Hist}{\sim} h_t$ and $\mathsf{pickt}(s, h) = ctid$ and $s.\mathsf{pc}(ctid) = i$ and $High^{\mathsf{lmem}}(s(ctid))$ and $H(i)$. If $s, h_s \rightsquigarrow_{\mathrm{conc}} s', h_{s'}$ and $s'.\mathsf{lowT} = t.\mathsf{lowT}$, then $s' \sim t$ and $h_{s'} \overset{Hist}{\sim} h_t$.

Proof. We only prove that $s' \sim t'$, since $h_{s'} \overset{Hist}{\sim} h_{t'}$ is a direct consequence of the hypotheses and of the definition of secure scheduler. We distinguish two cases:

1. The instruction to be executed is a sequential instruction. By definition of the semantics: $\langle s(ctid), s.\mathsf{gmem}\rangle \rightsquigarrow_{\mathrm{seq}} \langle s'(ctid), s'.\mathsf{gmem}\rangle$. By hypothesis, we have $s \overset{\mathsf{gmem}}{\sim} t$ and $s(ctid) \sim_l t(ctid)$. Thus by the sequential SC unwinding hypothesis, we have $s' \overset{\mathsf{gmem}}{\sim} t$ and $s'(ctid) \sim_l t(ctid)$. We conclude from $s \overset{\mathsf{lmem}}{\sim} t$ and from $s'.\mathsf{lowT} = t.\mathsf{lowT}$ that $s' \overset{\mathsf{lmem}}{\sim} t$, and that $s' \sim_g t$ by definition of state equivalence.
2. The instruction to be executed is of the form $\mathsf{start}\ pc$. By the hypotheses and the definition of state equivalence, it is sufficient to notice that the created thread is not observable, i.e., $H(pc)$ which follows from $se(i) \leq se(pc)$ by typability, and $H(i)$.

□

Lemma 3. Hypotheses 1, 2, 3, 4 and 5 hold for all typable programs.

Proof. **Hypothesis 1** is an instance of the low lemma of [BR05]. The lemma can be formulated as:

$$\left.\begin{array}{c} s \overset{\bullet}{\sim}_{S,S'} s' \\ s \rightsquigarrow_{\mathrm{seq}} t \\ s' \rightsquigarrow_{\mathrm{seq}} t' \\ s.\mathsf{pc} = s'.\mathsf{pc} = i \\ i \vdash S \Rightarrow T \\ i \vdash S' \Rightarrow T' \end{array}\right\} \Rightarrow t \overset{\bullet}{\sim}_{T,T'} t' \wedge (t.\mathsf{pc} = t'.\mathsf{pc} \vee \Phi_{t,t',T,T'})$$

where

$$\Phi_{t,t',T,T'} = \mathsf{highos}(t, T) \wedge \mathsf{highos}(t', T') \wedge (H(t.\mathsf{pc}) \vee H(t'.\mathsf{pc}))$$

Indeed, assume that $se, \mathcal{S} \vdash P$ and define $S = S' = \mathcal{S}_i$. Applying the lemma, we conclude that $t \overset{\bullet}{\sim}_{T,T'} t'$ for $T \leq \mathcal{S}_j$ and $T' \leq \mathcal{S}_{j'}$ where $j = t.\mathsf{pc}$ and $j' = t.\mathsf{pc}'$.

There are two cases to consider: either $j = j'$, in which case we can apply the double monotony lemma, see [BPR07b], to conclude, or $j \neq j'$, in which case $\mathcal{S}_j$ and $\mathcal{S}_{j'}$ are high, in which case we conclude by definition of operand stack indistinguishability.

Hypothesis 2 is a trivial consequence of the definition of $\lambda_{\text{init}}(i) = \langle \epsilon, i \rangle$ and of the fact that stack types should be empty at initial program points.

Hypothesis 3 is an instance of the high lemma of [BR05]. The lemma can be formulated as:

$$\left.\begin{array}{c} s \overset{\bullet}{\sim}_{S,S'} s' \\ s \leadsto_{\text{seq}} t \\ \mathsf{highos}(s, S) \\ H(s.\mathsf{pc}) \\ s.\mathsf{pc} \vdash S \Rightarrow T \end{array}\right\} \Rightarrow t \overset{\bullet}{\sim}_{T,S'} s' \wedge \mathsf{highos}(t, T)$$

Indeed, assume that $se, \mathcal{S} \vdash P$ and define $S = \mathcal{S}_i$. Applying the lemma, we conclude that $t \overset{\bullet}{\sim}_{T,S'} s'$ for $T \leq \mathcal{S}_j$ where $j = t.\mathsf{pc}$. We can apply the single monotony lemma, see [BPR07b], to conclude.

Hypothesis 4 follows from unfolding the definition and from the low lemma. Indeed, we have to show that one of the following holds: either $s'(ctid).\mathsf{pc} = t'(ctid).\mathsf{pc}$, or $H(s'(ctid).\mathsf{pc})$ or $H(t'(ctid).\mathsf{pc})$. By the low lemma, we conclude.

Hypothesis 5 (1) and (3) are immediate consequences of the definition of initial states and operand stack indistinguishability. Item (2) follows from a simple analysis of the type system.

□

Execution traces To conclude the proof of noninterference, we introduce an auxiliary function $\leadsto^{\mathsf{vis}}_{\text{conc}}$ that collapses execution steps on hidden threads: intuitively, $s \leadsto^{\mathsf{vis}}_{\text{conc}} s'$ iff neither s or s' have a hidden thread, and $s \leadsto_{\text{conc}} s'$ or $s \leadsto^{\star}_{\text{conc}} s'$ and all intermediate states have a hidden state. The formal definition of $s \leadsto^{\mathsf{vis}}_{\text{conc}} s'$ is given in Figure 8; the definition relies on the following four predicates on concurrent states, where $\#X$ denotes the cardinal of X:

$$\begin{aligned} \mathsf{GH}(s) &\Leftrightarrow \forall tid \in s.\mathsf{highT}.\ High^{\mathsf{lmem}}(s(tid)) \\ \mathsf{GH}_{\leq 1}(s) &\Leftrightarrow \mathsf{GH}(s) \wedge \#(s.\mathsf{hidT}) \leq 1 \\ \mathsf{GH}_1(s) &\Leftrightarrow \mathsf{GH}(s) \wedge \#(s.\mathsf{hidT}) = 1 \\ \mathsf{GH}_0(s) &\Leftrightarrow \mathsf{GH}(s) \wedge \#(s.\mathsf{hidT}) = 0 \end{aligned}$$

and two execution relations $\leadsto^{\mathsf{vis}}_{\text{conc}}$ and $\leadsto^{\mathsf{hid}}_{\text{conc}}$ defined by the clauses of Figure 8; informally, $s \leadsto^{\mathsf{vis}}_{\text{conc}} s'$ iff $s \leadsto^{\star}_{\text{conc}} s'$ and $\mathsf{GH}_0(s)$ and $\mathsf{GH}_0(s')$, and all intermediate steps s_i verify $\mathsf{GH}_1(s_i)$.

Lemma 4. *If* $P, \mu_1 \Downarrow \mu_1'$, *then* $s_{\text{init}}(\mu_1), \epsilon^H ist(\leadsto^{\mathsf{vis}}_{\text{conc}})^\star s$ *with* $s.\mathsf{lowT} = \emptyset$ *and* $s.\mathsf{gmem} = \mu_1'$.

Proof. First, we prove that $\mathsf{GH}_{\leq 1}$ is an invariant of program execution, using the GH hypotheses and the hypothesis that the scheduler is secure. Then, we prove that final states must verify GH_0. It is then easy to conclude. □

$$\frac{s, h \rightsquigarrow_{\mathrm{conc}} s', h' \quad \mathsf{GH}_0(s) \quad \mathsf{GH}_0(s')}{s, h \rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}} s', h'}$$

$$\frac{s, h \rightsquigarrow_{\mathrm{conc}} s', h' \quad \mathsf{GH}_1(s) \quad \mathsf{GH}_1(s')}{s, h \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h'} \qquad \frac{s, h \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h' \quad s', h' \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s'', h''}{s, h \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s'', h''}$$

$$\frac{s, h \rightsquigarrow_{\mathrm{conc}} s', h' \quad s', h' \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s'', h'' \quad s'', h'' \rightsquigarrow_{\mathrm{conc}} s''', h''' \quad \mathsf{GH}_0(s) \quad \mathsf{GH}_0(s''')}{s, h \rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}} s''', h'''}$$

Fig. 8. Auxiliary execution relations

Next, we prove the invariance of next under high steps in presence of a hidden thread. Below, we extend next as a function to states s such that $\mathsf{GH}_1(s)$, and define $\mathsf{next}(s)$ as $s.pc(tid)$, where $s.\mathsf{hidT} = \{tid\}$.

Lemma 5. *If* $s, h_s \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h_{s'}$ *and* $s \sim t$ *and* $s \overset{pc}{\sim} t$ *then* $s' \sim t$ *and* $h_s \overset{Hist}{\sim} h_{s'}$, *and* $s' \overset{pc}{\sim} t$, *and* $\mathsf{next}(s) = \mathsf{next}(s')$.

Proof. The proof proceeds by induction over the derivation of $s, h_s \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h_{s'}$ and uses the fact that the scheduler is secure, and that by definition of $\rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}}$, $\mathsf{GH}_1(s)$ and $\mathsf{GH}_1(s')$.

- If $s, h \rightsquigarrow_{\mathrm{conc}} s', h'$. Since $s, h_s \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h_{s'}$, we have $s.\mathsf{lowT} = s'.\mathsf{lowT}$; besides, $s.\mathsf{lowT} = t.\mathsf{lowT}$ as $s \overset{pc}{\sim} t$. Hence $s'.\mathsf{lowT} = t.\mathsf{lowT}$. Let $\mathsf{pickt}(s, h_s) = ctid$ and $s.pc(ctid) = i$. As $\mathsf{GH}_1(s)$, we have $H(i)$ (since the scheduler is secure), and thus $High^{\mathsf{lmem}}(s(ctid))$. Item i) follows from the concurrent SC unwinding lemma (Lemma 2). Item ii) follows from the fact that $s \overset{pc}{\sim} s'$, and item iii) follows from the observation that if $i \in \mathsf{Dom}(\mathsf{next})$, i.e., $s.\mathsf{hidT} = \{ctid\}$, then $s'.pc(ctid) = i' \in \mathsf{Dom}(\mathsf{next})$, and hence by **NeP1**, $\mathsf{next}(i) = \mathsf{next}(i')$, hence $\mathsf{next}(s) = \mathsf{next}(s')$; otherwise, if $i \notin \mathsf{Dom}(\mathsf{next})$, then $\mathsf{next}(s) = \mathsf{next}(s')$ holds trivially.
- If $s, h \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s_0, h_0$ and $s_0, h_0 \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s', h_{s'}$, then we can apply the induction hypothesis to both reduction sequences, using the conclusions of the first application of the induction hypothesis to apply the second one, to conclude that $s_0 \sim t$ and $h \overset{Hist}{\sim} h_0$ and $s_0 \overset{pc}{\sim} t$ and $\mathsf{next}(s) = \mathsf{next}(s_0)$ and $s' \sim t$ and $h_0 \overset{Hist}{\sim} h_{s'}$ and $s' \overset{pc}{\sim} t$ and $\mathsf{next}(s_0) = \mathsf{next}(s')$. We are done by transitivity of history equivalence and pc equivalence and equality.

□

Next, we prove a locally respects lemma for $\rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}}$ by using Lemma 5.

Lemma 6. *If* $s, h_s \rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}} s', h_{s'}$ *and* $t, h_t \rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}} t', h_{t'}$ *and* $s \sim t$ *and* $s \overset{pc}{\sim} t$ *and* $h_s \overset{Hist}{\sim} h_t$ *then* $s' \sim t'$ *and* $s' \overset{pc}{\sim} t'$ *and* $h'_s \overset{Hist}{\sim} h'_t$.

Proof. Let $k_s = se(i_s)$, where $i_s = s(ctid_s).\mathsf{pc}$ and $ctid_s = \mathsf{pickt}(s, h_s)$ and $k_t = se(i_t)$ where $i_t = t(ctid_t).\mathsf{pc}$ and $ctid_t = \mathsf{pickt}(t, h_t)$. By definition of $\rightsquigarrow^{\mathsf{vis}}_{\mathrm{conc}}$, there are four cases to treat; we only consider two cases:

– if $s, h_s \rightsquigarrow_{\mathrm{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\mathrm{conc}} t', h_{t'}$. Note that $s.\mathsf{lowT} = t.\mathsf{lowT}$ follows from $s \overset{\mathsf{pc}}{\sim} t$. Hence, by definition of secure scheduler, there are two cases to treat: either $ctid_s = ctid_t$ and $k_s = k_t$, or else $k_s \not\leq k$ and $k_t \not\leq k$.
In the first case, observe that necessarily $s'.\mathsf{lowT} = s.\mathsf{lowT}$ and $t'.\mathsf{lowT} = t.\mathsf{lowT}$, and thus $s'.\mathsf{lowT} = t'.\mathsf{lowT}$. Furthermore, $i_s = i_t$. Item i) follows by Lemma 1; item ii) follows from Hypothesis 4; item iii) follows from the fact that $h_{s'} = \langle ctid_s, k_s\rangle :: h_s$ and $h_{t'} = \langle ctid_t, k_t\rangle :: h_t$.
In the second case, the result is a direct consequence of the definitions.
– if $s, h_s \rightsquigarrow_{\mathrm{conc}} s_1, h_{s_1} \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} s_2, h_{s_2} \rightsquigarrow_{\mathrm{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\mathrm{conc}} t_1, h_{t_1} \rightsquigarrow^{\mathsf{hid}}_{\mathrm{conc}} t_2, h_{t_2} \rightsquigarrow_{\mathrm{conc}} t', h_{t'}$. In this case, we must have $ctid_s = ctid_t$ (so we drop subscripts) and $k_s = k_t$, and furthermore $s_1.\mathsf{lowT} = t_1.\mathsf{lowT}$ and $i_s = i_t$. We apply Lemma 1 to s and t to conclude that $s_1 \sim t_1$ and $h_{s_1} \overset{Hist}{\sim} h_{t_1}$. Furthermore, $s_1 \overset{\mathsf{pc}}{\sim} t_1$, and by **NeP3**, $\mathsf{next}(s_1) = \mathsf{next}(t_1)$.
By applying Lemma 5 to s_1, s_2 and t_1, we conclude that $s_2 \sim t_1$, and $h_{s_1} \overset{Hist}{\sim} h_{s_2}$, and $s_2 \overset{\mathsf{pc}}{\sim} t_1$, and $\mathsf{next}(s_1) = \mathsf{next}(s_2)$. Using these facts and by applying Lemma 5 on t_1, t_2 and s_2, we conclude that $t_2 \sim s_2$, and $h_{t_1} \overset{Hist}{\sim} h_{t_2}$, and $t_2 \overset{\mathsf{pc}}{\sim} s_2$, and $\mathsf{next}(t_1) = \mathsf{next}(t_2)$.
To prove that $s' \sim t'$, we apply Hypothesis 3 to conclude that $s_2(ctid) \sim_l s'(ctid)$ and $s_2 \overset{\mathsf{gmem}}{\sim} s'$. Likewise, $t_2(ctid) \sim_l t'(ctid)$ and $t_2 \overset{\mathsf{gmem}}{\sim} t'$. By Hypothesis 5, we also have $s_2(ctid) \sim_l t_2(ctid)$, and hence $s'(ctid) \sim_l t'(ctid)$, from which it is easy to conclude.
To conclude that $s' \overset{\mathsf{pc}}{\sim} t'$, we use the fact that $\mathsf{next}(s_2) = \mathsf{next}(t_2)$ and apply **NeP2**. To conclude that $h_{s'} \overset{Hist}{\sim} h_{t'}$, we use the fact that $h_{s_1} \overset{Hist}{\sim} h_{t_1}$ and that $h_{s_1} \overset{Hist}{\sim} h_{s'}$ and $h_{t_1} \overset{Hist}{\sim} h_{t'}$.

□

We can now conclude the proof of soundness (Theorem 1) by repeatedly applying Lemma 6, and by Lemma 4.

Soundness of the instantiation

Definition 18 (Source labels and control flows). *Natural numbers are added as labels to the source syntax to identify program points where control flow can branch. Therefore, commands are described by the following grammar:*

$$c ::= [x := e]^n \mid c; c \mid [\texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c]^n \mid [\texttt{while}\ e\ \texttt{do}\ c]^n \mid [\texttt{fork}(c)]^n$$

Definition 19 (Branching commands). *The branching commands are those of the form* $\texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c$ *and* $\texttt{while}\ e\ \texttt{do}\ c$ *. The set* $\mathcal{LL}^{\#}$ *consists on all the labels of branching commands in the program.*

We define a notion of contexts to refer to instructions inside of programs.

Definition 20 (Contexts). *A context C for commands is defined as an element of the following grammar:*

$$\begin{array}{l} C ::= \bullet \mid [x := e]^n \mid [\texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ C]^n \mid [\texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ c]^n \mid \\ \quad [\texttt{while}\ e\ \texttt{do}\ C]^n \mid \underline{c}; C \mid \underline{C}; c \mid [\texttt{fork}(C)]^n \end{array}$$

where e is an expression, c is a command, $\underline{c}$ is a single command, and $\underline{C}$ is a context denoting a single command.

The definition of contexts for unlabeled commands is very similar to Definition 20. Therefore, we abuse of notation and denote C as contexts for labeled or unlabeled commands. We define the size of context as follows.
Our compilation function $\mathcal{S}$ takes two arguments: the code to compiled and the compiled code belonging to different threads. For technical reasons, it is necessary to identify what is the value of the second argument when the compilation of commands are performed. We then introduce the following judgment.

Definition 21. *Given commands c and c', and sequences of compiled instructions T, and T', the judgment $\mathcal{S}(c,T) ::\vdash \mathcal{S}(c',T')$ is defined by the following rules.*

$$\frac{}{\mathcal{S}(c,T) ::\vdash \mathcal{S}(c,T)}[REFL] \qquad \frac{\mathcal{S}(c_1,T) ::\vdash \mathcal{S}(c',T')}{\mathcal{S}(c_1;c_2,T) ::\vdash \mathcal{S}(c',T')}[SEQ_1]$$

$$\frac{(lc_1,T_1) = \mathcal{S}(c_1,T) \qquad \mathcal{S}(c_2,T_1) ::\vdash \mathcal{S}(c',T')}{\mathcal{S}(c_1;c_2,T) ::\vdash \mathcal{S}(c',T')}[SEQ_2]$$

$$\frac{\mathcal{S}(c_1,T) ::\vdash \mathcal{S}(c,T)}{\mathcal{S}([\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]^n,T) ::\vdash \mathcal{S}(c,T)}[IF_1]$$

$$\frac{(lc_1,T_1) = \mathcal{S}(c_1,T) \qquad \mathcal{S}(c_2,T_1) ::\vdash \mathcal{S}(c,T)}{\mathcal{S}([\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]^n,T) ::\vdash \mathcal{S}(c,T)}[IF_2]$$

$$\frac{\mathcal{S}(c,T) ::\vdash \mathcal{S}(c',T')}{\mathcal{S}([\texttt{while}\ e\ \texttt{do}\ c]^n,T) ::\vdash \mathcal{S}(c',T')}[WHL] \qquad \frac{\mathcal{S}(c,T) ::\vdash \mathcal{S}(c',T')}{\mathcal{S}(\texttt{fork}(c),T) ::\vdash \mathcal{S}(c',T')}[FRK]$$

$$\frac{\mathcal{S}(c,T) ::\vdash \mathcal{S}(c',T') \qquad \mathcal{S}(c',T') ::\vdash \mathcal{S}(c'',T'')}{\mathcal{S}(c,T) ::\vdash \mathcal{S}(c'',T'')}[TRANS]$$

Intuitively, $\mathcal{S}(c,T) ::\vdash \mathcal{S}(c',T')$ denotes the fact that when compiling the command c, the function $\mathcal{S}$ receives T' as a second argument when compiling c'. For simplicity, we write $\mathcal{S}(c,[]) ::\vdash \mathcal{S}(c',T')$ as $\mathcal{C}(c) ::\vdash \mathcal{S}(c',T')$.
We assume that two instructions are the same iff they are located in the same position of in the compiled code. The following function is useful to define regions at the target code.

Definition 22 (Function $\odot$). *Given a program P and its compilation $l_p = \mathcal{C}(P)$, the function $\odot :: l_p \rightarrow [1..\#l_p]$ is defined as $\odot(i) =$ the position of the instruction i in the sequence l_p.*

We then define regions in the target code.

Definition 23 (Compiler regions). *Given a branching command $[c]^n$ in a source program P. Then, we define* $\mathsf{tregion}(n)$ *as follow:*

$[c]^n = [\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2]^n$*)*

$$\mathsf{tregion}(n) = (\bigcup_{n' \in \mathsf{sregion}(n)} \{\odot i \mid \exists T.\ i \in \mathbf{fst}(\mathcal{S}([c']^{n'}, T)), \mathcal{C}(P) ::\vdash \mathcal{S}([c']^{n'}, T)\}) \bigcup \{\odot \texttt{goto}^n_{else}\}$$

where $\texttt{goto}^n_{else}$ *denotes the* $\texttt{goto}$ *instructions placed after the compilation of command c_2 – see Figure 6.*

$[c]^n = [\texttt{while } e \texttt{ do } c]^n$*)*

$$\mathsf{tregion}(n) = (\bigcup_{n' \in \mathsf{sregion}(n)} \{\odot i \mid \exists T.\ i \in \mathbf{fst}(\mathcal{S}([c']^{n'}, T)), \mathcal{C}(P) ::\vdash \mathcal{S}([c']^{n'}, T)\}) \bigcup \{\odot(\texttt{ifeq}^n_w)\} \bigcup \{\odot i \mid i \in \mathsf{e}^n_w\}$$

where e^n_w *and* $\texttt{ifeq}^n_w$ *denote the sequence of instructions obtaining by compiling the guard e and the* ifeq *instruction generated by compiling the while itself, respectively – see Figure 6.*

We indicate how to determine security environment se from the functions E and F described in Figure 7.

Definition 24 (se determined by E and F). *Given a program P and an instruction $i \in \mathcal{C}(P)$, we define $se(\odot(i))$ as follows:*

- *If $i = \mathsf{start}$ and $\odot(i) = 1$, then $se(\odot(i)) = L$.*
- *If $i = \mathsf{return}$ and $\odot(i) = \#\mathcal{C}(P)$, then $se(\odot(i)) = L$.*
- *If $i = \mathsf{return}$ and $[\texttt{fork}(c)]^n$ is the smallest* $\texttt{fork}$ *such that $i \in$* ***snd***$(\mathcal{S}([\texttt{fork}(c)]^n, T))$ *and $\mathcal{C}(P) ::\vdash \mathcal{S}([\texttt{fork}(c)]^n, T)$, then $se(\odot(i)) = E(n)$.*
- *If $i = \texttt{goto}^n_{else}$, then $se(\odot(i)) = F(n)$.*
- *If $i = \texttt{ifeq}^n_w$, then $se(\odot(i)) = F(n)$.*
- *If $i \in \mathsf{e}^n_w$, then $se(\odot(i)) = F(n)$.*
- *Otherwise, $se(\odot(i)) = E(n)$, where $[c]^n$ is the smallest command in P such that $i \in$* ***fst***$(\mathcal{S}([c]^n, T))$ *and $\mathcal{C}(P) ::\vdash \mathcal{S}([c]^n, T)$.*

The following two lemmas are important to prove **NePd**. The first one indicates that it is always possible to reach a low instruction after getting out of a conditional whose guard contains secrets.

Lemma 7. *Given se obtained as described in Definition 24, program P, a context C, a branching command $[c]^n$ such that $P = C[[c]^n]$, $\vdash_\circ P$, $\vdash_\circ [c]^n_\bullet$ is in the type derivation of P, $l_p = \mathcal{C}(P)$, $i \in \mathsf{tregion}(n)$; then $\exists j \in l_p. i \mapsto^* \odot(j) \wedge se(\odot(j)) = L$*

The next lemma indicates that instructions, which their security environment is high, are placed inside of high branches.

Lemma 8 (From inside of top-level-branches). *Given commands d and c, and label n such that $\vdash_\circ d$, $\vdash_\bullet [c]^n_\bullet$ is in the type derivation of $\vdash_\circ d$ and $n \in \mathsf{labels}(d)$, then there exists a command c' and a label k such that $k \in \mathsf{labels}(d)$, $\vdash_\circ [c']^k_\bullet$ is in the type derivation of $\vdash_\circ d$, and $n \in \mathsf{labels}(c')$.*

Theorem 4 (NePd). $\mathsf{Dom}(\mathsf{next}) = \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\, i \mapsto^\star j \wedge \neg H(j)\}$

Proof. In order to prove this equality, we need to prove inclusion of sets. Firstly, $\mathsf{Dom}(\mathsf{next}) \subseteq \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\, i \mapsto^\star j \wedge \neg H(j)\}$) is proved by inspecting Definition 17, and Lemma 7. Lastly, $\{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\, i \mapsto^\star j \wedge \neg H(j)\} \subseteq \mathsf{Dom}(\mathsf{next})$) is proved by contradiction. We assume that $k \in \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\, i \mapsto^\star j \wedge \neg H(j)\} \wedge k \notin \mathsf{Dom}(\mathsf{next})$. Then, we do case analysis on the command which compilation generated the instruction i such that $\odot(k) = i$, and based on Lemma 8, contradictions are obtained. $\square$

We show a property regarding next that is important to prove **NeP1**.

Lemma 9 (next is not defined for join points). *Given a branching point $[c]^n$ in the typable source program $\vdash_\circ P$ such that $\vdash_\circ [c]^n_\bullet$, then $\mathsf{next}(\mathsf{jun}(n))$ is undefined.*

Proof. By contradiction. $\square$

Theorem 5 (NeP1). $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = \mathsf{next}(j)$

Proof. Since $i \in \mathsf{Dom}(\mathsf{next})$, there exists a command c_i and a number n_i such that $\vdash_\circ [c_i]^{n_i}_\bullet$ and $\forall k \in \mathsf{tregion}(n_i).\mathsf{next}(k) = \mathsf{jun}(n_i)$. On the other hand, since $j \in \mathsf{Dom}(\mathsf{next})$, there exists another command c_j and a number n_j such that $\vdash_\circ [c_j]^{n_j}_\bullet$ and $\forall k \in \mathsf{tregion}(n_j).\mathsf{next}(k) = \mathsf{jun}(n_j)$. By instantiating SOAP 1 with i and j, we have that $i \mapsto j \wedge (i = n_i \vee i \in \mathsf{tregion}(n_i)) \Rightarrow j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i)$, which we split into:

$$i \mapsto j \wedge i = n_i \Rightarrow j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i) \vee \qquad (1)$$

$$i \mapsto j \wedge i \in \mathsf{tregion}(n_i) \Rightarrow j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i) \qquad (2)$$

Since $i \in \mathsf{Dom}(\mathsf{next})$, we have that $i \in \mathsf{tregion}(n_i)$. We proceed by doing case analysis on i.

$i = n_i$**)** Observe that this can happen when c_i is a `while`-loop. By (1), we have that $j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i)$.

  $j \in \mathsf{tregion}(n_i)$**)** By definition of $\mathsf{tregion}(n_i)$, we have that $\mathsf{next}(j) = \mathsf{jun}(n_i)$, which implies that $\mathsf{next}(i) = \mathsf{next}(j)$.

  $j = \mathsf{jun}(n_i)$**)** By Lemma 9, $\mathsf{next}(j)$ is undefined. However, $j \in \mathsf{Dom}(\mathsf{next})$ by Hypothesis, which implies that $\mathsf{next}(j)$ is defined. Contradiction.

$i \in \mathsf{tregion}(n_i)$**)** By (2), we have that $j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i)$.

  $j \in \mathsf{tregion}(n_i)$**)** By definition of $\mathsf{tregion}(n_i)$, we have that $\mathsf{next}(j) = \mathsf{jun}(n_i)$, which implies that $\mathsf{next}(i) = \mathsf{next}(j)$.

  $j = \mathsf{jun}(n_i)$**)** By Lemma 9, $\mathsf{next}(j)$ is undefined. However, $j \in \mathsf{Dom}(\mathsf{next})$ by Hypothesis, which implies that $\mathsf{next}(j)$ is defined. Contradiction.

□

Theorem 6 (NeP2). $i \in \mathsf{Dom}(\mathsf{next}) \wedge j \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = j$

Proof. Since $i \in \mathsf{Dom}(\mathsf{next})$, there exists a command c and a number n such that $\vdash_\circ [c]^n_\bullet$ and $\forall k \in \mathsf{tregion}(n).\mathsf{next}(k) = \mathsf{jun}(n)$. We also know that $i \in \mathsf{tregion}(n)$. By instantiating SOAP 1 with i and j, we have that $i \mapsto j \wedge (i = n \vee i \in \mathsf{tregion}(n)) \Rightarrow j \in \mathsf{tregion}(n) \vee j = \mathsf{jun}(n)$, which we split into:

$$i \mapsto j \wedge i = n \Rightarrow j \in \mathsf{tregion}(n) \vee j = \mathsf{jun}(n_i) \vee \quad (3)$$
$$i \mapsto j \wedge i \in \mathsf{tregion}(n) \Rightarrow j \in \mathsf{tregion}(n) \vee j = \mathsf{jun}(n) \quad (4)$$

We proceed by doing case analysis on i.

$i = n_i$) Observe that this can happen when c_i is a `while`-loop. By (3), we have that $j \in \mathsf{tregion}(n) \vee j = \mathsf{jun}(n)$.
 $j \in \mathsf{tregion}(n)$) It cannot happen. Observe that we assume that $j \notin \mathsf{tregion}(n)$ by Hypothesis.
 $j = \mathsf{jun}(n)$) Since $i \in \mathsf{tregion}(n)$, we know that $\mathsf{next}(i) = \mathsf{jun}(n)$ and $j = \mathsf{jun}(n)$, which implies that $\mathsf{next}(i) = j$ as expected.
$i \in \mathsf{tregion}(n_i)$) It proceeds similarly as when $i = n_i$ but applying (4) instead.

□

In order to prove **NeP3**, we firstly need to show that the compilation function in Figure 6 preserves inclusion of regions. More precisely, we have that

Lemma 10. *Given a command c_p with two branching instructions $[c_1]^{n_1}$ $[c_2]^{n_2}$, where $n_1 \neq n_2$, and two contexts C_1 and C_2 with only a hole each one. If $c_p = C_1[[c_1]^{n_1}]$ and $c_p = C_2[[c_2]^{n_2}]$, then*

- *If* $\mathsf{sregion}(n_1) \subset \mathsf{sregion}(n_2)$, *then* $\mathsf{tregion}(n_1) \subset \mathsf{tregion}(n_2)$.
- *If* $\mathsf{sregion}(n_2) \subset \mathsf{sregion}(n_1)$, *then* $\mathsf{tregion}(n_2) \subset \mathsf{tregion}(n_1)$.
- *If* $\mathsf{sregion}(n_1) \cap \mathsf{sregion}(n_2) = \emptyset$, *then* $\mathsf{tregion}(n_1) \cap \mathsf{tregion}(n_2) = \emptyset$.

The following two lemmas are very similar to the statement of **NeP3**, but they include some assumptions about the typing of branching point $[c]^n$.

Lemma 11. *Given program P and command $[c]^i$ such that $\vdash_\alpha [c]^i_\alpha$ is in the type derivation of $\vdash_\circ P$, $j, k \in \mathsf{Dom}(\mathsf{next}), i \notin \mathsf{Dom}(\mathsf{next}), i \mapsto j, i \mapsto k$, and $j \neq k$, where $\alpha \in \{\circ, \bullet\}$, then* $\mathsf{next}(k) = \mathsf{next}(j)$.

Proof. It consists on proving that the hypothesis does not hold. To do that, we consider, based on Lemma 10, how target regions associated to k and i are included. □

Lemma 12. *Given program P and command $[c]^i$ such that $\vdash_\circ [c]^i_\bullet$ is in the type derivation of $\vdash_\circ P$, $j, k \in \mathsf{Dom}(\mathsf{next}), i \notin \mathsf{Dom}(\mathsf{next}), i \mapsto j, i \mapsto k$, and $j \neq k$, then* $\mathsf{next}(k) = \mathsf{next}(j)$.

Proof. By case analysis on $[c]^i$. □

Theorem 7 (NeP3). $j, k \in \mathsf{Dom}(\mathsf{next}) \wedge i \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = \mathsf{next}(k)$

Proof. We have that i is a branching command. Consequently, $[c]^i$ can be typed as $\vdash_\circ [c]^i_\circ$, $\vdash_\bullet [c]^i_\bullet$, or $\vdash_\circ [c]^i_\bullet$ in the type derivation of the program. By case analysis on the typing of $[c]^i$, the result follows based on Lemmas 11 and 12. □

Theorem 8 (NeP4). $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge k \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = k$

Proof. By applying Theorem 5 with i and j, we obtain that $\mathsf{next}(i) = \mathsf{next}(j)$. By applying Theorem 6 with i and k, we obtain that $\mathsf{next}(i) = k$. Therefore, we have that $\mathsf{next}(j) = \mathsf{next}(i) = k$. □

Theorem 6.1 *Definition 17 satisfies properties **NePd** and **NeP1–4**.*

Proof. The proof easily follows from Theorems 4, 5, 6, 7, and 8. □

Typability Preservation

The following lemma claims that a source expression typable compiles to typable target code. Besides the conclusion of typability for the target code, the lemma also states that the final security operand stacks are of the form $k : st$, with k being the type of the source expression, and st being the initial operand stack used in the transfer rules for the compiled code. Since the new concurrent features of source and target languages in this paper do not include new expressions, this lemma is equivalent to previous work for sequential languages, and a proof can be found in [Rez06].

Lemma 13. *Let e be an expression in $[c]^n$ such that $[c]^n$ is the inner-most command that encloses e and $\Gamma \vdash c : E, F$ and $\Gamma \vdash e : k$, and $\mathcal{S}(c)[i..j] = \mathcal{E}(e)$. Let se be the security environment determined by E, F. Then for any $st_i \in \mathcal{ST}$ there exist $st_{i+1}, ..st_j$ such that the following hold:*

1. *for every $l \mapsto l'$ in $i..j$ then $l, se \vdash st_l \Rightarrow st_{l'}$;*
2. *$j, se \vdash st_j \Rightarrow (k \sqcup se(i)) :: st_i$.*

Theorem 6.2 *For a given source-level program c, assume $nf(c)$ is obtained from c by replacing all occurrences of* `fork`*(d) by d. If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [VSI96] then $se, \mathcal{S} \vdash \mathcal{C}(c)$ for some se and $\mathcal{S}$.*

Proof. First we define how to obtain intermediate typing from the high level typing as follows: Let D be a typing derivation for a source program SP in the high level type system. Define security environments E and F as follows:

– If n belongs to some region of a branching label n' of a command c' in SP such that the intro judgement for c' types it with write effect H, then $E(n)$ and $F(n)$ are defined as the write level of the intro judgement for $[c]^n$ in D. That is, if $D :: \vdash c : H$ then $F(n) = E(n) = H$.

– Otherwise, $E(n) = L$. If n is a branching label not contained in any region such that its intro judgement for types it with write effect H, then $F(n) = H$.

By induction in the structure of the source commands.

Case: $c \equiv [x := e]^n$. We have to prove that if $[x := e]^n$ is typable:

$$\frac{\Gamma \vdash e : k \qquad k \leq \Gamma(x)}{\Gamma \vdash x := e \ : \Gamma(x)}$$

then the constraint $k \sqcup E(n) \leq \Gamma(x)$ from its corresponding intermediate typing rule holds. By definition of E above, if $\Gamma(x) = H$ and n is inside a high region then $E(n) = H$. Otherwise $E(n) = L$. So the constraint $k \sqcup E(n) \leq \Gamma(x)$ is satisfied because of the contraint of the $(Assign)$ high level typing rule for, $k \leq \Gamma(x)$. By Definition 24, for all program points j included in the compilation of $[x := e]^n$, then $se(j) = E(n)$. By Lemma 13, we have that typability of compilation of expression e leads to an security operand stack of the form $(k \sqcup se(i)) :: st_i$. Typability of the store instruction in n follows by constraint $k \sqcup E(n) \leq \Gamma(x)$ and the fact that $E(n) = se(j)$ for all j in n.

Case: $c \equiv [\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$. In the high level type system:

$$\textsc{Cond}\quad \frac{\vdash e : k \qquad \vdash c_1 : k \qquad \vdash c_2 : k}{\vdash [\mathsf{if}\ e\ \mathsf{then}\ \ c\ \mathsf{else}\ c']^n \ : \ k}$$

We need to show that the if command is typable by the intermediate type system with the definition of E, F given above. We need to show that if $\vdash c_i : H$ then $\vdash_\alpha c_i : E, F$. This follows by inductive hypothesis. Furthermore if n does not belong to any high region, then we need to show the hypothesis of the $TOP - H - COND$ rule on E, that is $F(n) = L$. This follows by definition of F above. To prove that compilation of command n is typable, recall that by definition of source regions, c_1 and c_2 are included in the region of n and then by definition of E above $F(n') = E(n') = H$ for all program points n' inside a high region. By Definition 24, $se(j) = H$ for all instructions j inside the high region of n. Thus the constraint of the target typing rule $se = \mathsf{lift}_k(se)$ holds. By Lemma 13, we have that typability of compilation of expression e leads to an security operand stack of the form $(k \sqcup se(i)) :: \epsilon$. By semantics of the if instruction, the operand stack is empty. Thus the lift of the security operand stack holds and we conclude.

Case: $c \equiv [\mathsf{while}\ e\ \mathsf{do}\ c1]^n$. The proof is analog to the if case.

Case: $c \equiv c'; c''$. By inductive hypothesis.

Case: $c \equiv [\mathsf{fork}\ d]^n$. By hypothesis, fork d is tranformed into command d. Typability in the intermediate type system follows by inductive hypothesis. Recall that compilation of fork d gives a start instruction for the current thread and compilation of d for another thread. Typability of d follows by the fact that d is typable applying inductive hypothesis. To prove typability of compilation of start, we need to verify the following typing rule:

$$\frac{P[i] = \mathsf{start}\ pc \quad se(i) \leq se(pc)}{se, i \vdash st \Rightarrow st}$$

By Definition 24, $se(\odot(\mathsf{start}) = E(n)$, where n corresponds to a skip command. Since i belongs to the compilation of n then $E(n) = se(i)$. We have that $se(i) \leq se(pc)$ and we conclude. $\square$

CHAPTER

4

Security for Multithreaded Programs under Cooperative Scheduling

In Proceedings of the Andrei Ershov International Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia, June 27-30, 2006. LNCS, Springer-Verlag.

Security for Multithreaded Programs under Cooperative Scheduling

Alejandro Russo and Andrei Sabelfeld

Dept.of Computer Science and Engineering, Chalmers University of Technology
412 96 Göteborg, Sweden, Fax: +46 31 772 3663

Abstract. Information flow exhibited by multithreaded programs is subtle because the attacker may exploit scheduler properties when deducing secret information from publicly observable outputs. Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing sensitive timing behavior of protected commands and therefore prevents undesired information flows. While a useful construct, `protect` is nonstandard and difficult to implement. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

1 Introduction

Information-flow security specifications and enforcement mechanisms for sequential programs have been developed for several years. Unfortunately, they do not naturally generalize to multithreaded programs [SV98]. Information flow in multithreaded programs remains an important open challenge [SM03]. Furthermore, otherwise significant efforts (such as Jif [MZZ$^+$06] and Flow Caml [Sim03]) in extending programming languages (such as Java and Caml) with information flow controls have sidestepped multithreading issues. Nevertheless, concurrency and multithreading are important in the context of security because environments of mutual distrust are often concurrent. As result, the need for controlling information flow in multithreaded programs has become a necessity.
This paper is focused on preventing attacks that exploit scheduler properties to deduce secret information from publicly observable outputs. Suppose h is a secret (or $high$) variable and l is a public (or low) one. Consider threads c_1 and c_2:

$$c_1 : (\texttt{if}\ h > 0\ \texttt{then}\ \texttt{sleep}(100)\ \texttt{else}\ \texttt{skip});\ l := 1$$
$$c_2 : \texttt{sleep}(50);\ l := 0$$

Although these threads do not exhibit insecure information flow in isolation (because 1 is always the outcome for l in c_1, and 0 is always the outcome for l in c_2), there is a race between assignments $l := 1$ and $l := 0$, whose outcome depends on secret h. If h is originally positive, then—under many schedulers—it is likely that the final value of l is 1. If h is not positive, then it is likely that the final value of l is 0. It is the timing behavior of thread c_1 that leaks—via the scheduler—secret information into l. This

$$\frac{\langle c_i, m\rangle \xrightarrow{\alpha} \langle c'_i, m'\rangle \qquad \alpha \in \{\epsilon, \vec{d}\} \qquad \sigma = i}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma, \langle c_1 \ldots c_{i-1} c'_i \alpha c_{i+1} \ldots c_n\rangle, m'\rangle}$$

$$\frac{\langle c_i, m\rangle \xrightarrow{\alpha} \langle \mathtt{stop}, m'\rangle \qquad \sigma = i}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma, \langle c_1 \ldots c_{i-1} c_{i+1} \ldots c_n\rangle, m'\rangle}$$

$$\frac{\langle c_i, m\rangle \xrightarrow{\not\leadsto} \langle c'_i, m\rangle \qquad \sigma = i \qquad \sigma' = (i \bmod n) + 1 \qquad c'_i \neq \mathtt{stop}}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma', \langle c_1 \ldots c_{i-1} c'_i c_{i+1} \ldots c_n\rangle, m\rangle}$$

Fig. 1. Semantics for threadpools

phenomenon is due to *internal timing*, i.e., timing that is observable to the scheduler. As in [SV98, VS99, Smi01, BC02, Smi03, RS06], we do not consider *external timing*, i.e., timing behavior visible to an attacker with a stopwatch.
Volpano and Smith have introduced a `protect` command that prevents the scheduler from observing the timing behavior of the protected command and therefore prevents undesired information flows. A protected command is executed atomically *by definition*. Although it has been acknowledged [SS00, RS06] that `protect` is hard to implement, no implementation of `protect` has been discussed by approaches that rely on it [VS99, Smi01, Smi03]. This paper presents a transformation that eliminates the need for `protect` under cooperative scheduling. This transformation can be integrated into source-to-source translation that introduces `yield` commands for cooperative schedulers. We show that both termination-insensitive and termination-sensitive security can be enforced by variants of the transformation in a language with dynamic thread creation.

2 Language

We consider a simple imperative language that includes `skip`, assignment, sequential composition, conditionals, and `while`-loops. Its sequential semantics is standard [Win93]. The language also includes dynamic thread creation and a `yield` command. A *command configuration* $\langle c, m\rangle$ consists of a command c and memory m. Memories $m : \mathit{IDs} \rightarrow \mathit{Vals}$ are finite maps from identifier names *IDs* to values *Vals*. Transitions between configurations have form $\langle c, m\rangle \xrightarrow{\alpha} \langle c', m'\rangle$ where α is either ϵ (empty label), $\vec{d}$ (indicating a sequence of newly spawned threads), or $\not\leadsto$. The latter label is used in the transition rule for `yield`:

$$\langle \mathtt{yield}, m\rangle \xrightarrow{\not\leadsto} \langle \mathtt{stop}, m\rangle$$

Labels are then propagated through sequential composition to the threadpool-semantics level. Dynamic thread creation is performed by command `fork`:

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

This has the effect of continuing with thread c while spawning a sequence of fresh threads $\vec{d}$. *Threadpool configurations* have form $\langle\!|\sigma, \langle c_1 \dots c_n\rangle, m|\!\rangle$ where σ is the scheduler's running thread number, $\langle c_1 \dots c_n\rangle$ is a threadpool, and m is a shared memory. Threadpool semantics, describing the behavior of threadpools and their interaction with the scheduler, are displayed in Figure 1. The rules correspond to normal execution of thread i from the threadpool, termination of thread i, and yielding by thread i. Note that due to cooperative scheduling, only termination or a `yield` by a thread may change the decision of the scheduler which thread to run next. Although these semantics model a round-robin scheduler, our approach can be generalized to a wide class of schedulers.
Let $\mathit{cfg} \rightarrow^0 \mathit{cfg}$, for any configuration cfg, and $\mathit{cfg} \rightarrow^v \mathit{cfg}'$, for $v > 0$, if there is a configuration cfg'' such that $\mathit{cfg} \rightarrow \mathit{cfg}''$ and $\mathit{cfg}'' \rightarrow^{v-1} \mathit{cfg}'$. Then, $\mathit{cfg} \rightarrow^* \mathit{cfg}'$ if $\mathit{cfg} \rightarrow^v \mathit{cfg}'$ for some $v \geq 0$. Threadpool configuration cfg *terminates* in memory m (written $\mathit{cfg} \Downarrow m$) if $\mathit{cfg} \rightarrow^* \langle\!|\sigma, \langle\rangle, m|\!\rangle$ for some σ. In particular, $\mathit{cfg} \Downarrow^v m$ is written when $\mathit{cfg} \rightarrow^v \langle\!|\sigma, \langle\rangle, m|\!\rangle$. If $\langle\rangle$ is not finitely reachable from cfg, then cfg *diverges* (written $\mathit{cfg} \Uparrow$). Termination $\Downarrow$ and divergence $\Uparrow$ are defined similarly for command configurations.

3 Security specification

We define two security conditions, termination-insensitive and termination-sensitive security, both based on *noninterference* [GM82]. Suppose *security environment* $\Gamma : \mathit{IDs} \rightarrow \{\mathit{high}, \mathit{low}\}$ specifies a partitioning of variables into high and low ones. Two memories m_1 and m_2 are *low-equal* ($m_1 =_L m_2$) if they agree on low variables, i.e., $\forall x \in \mathit{IDs}.\, \Gamma(x) = \mathit{low} \Longrightarrow m_1(x) = m_2(x)$.
Command c satisfies termination-insensitive noninterference if c's terminating executions on low-equal inputs produce low-equal results.

Definition 1. *Command c satisfies* termination-insensitive security *if*

$$\forall m_1, m_2 . m_1 =_L m_2 \;\;\&\;\; \langle\!|1, \langle c\rangle, m_1|\!\rangle \Downarrow m_1' \;\;\&\;\; \langle\!|1, \langle c\rangle, m_2|\!\rangle \Downarrow m_2' \Longrightarrow m_1' =_L m_2'$$

Command c satisfies termination-sensitive noninterference if c's executions on any two low-equal inputs either both diverge or both terminate in low-equal results.

Definition 2. *Command c satisfies* termination-sensitive security *if*

$$\forall m_1, m_2 . m_1 =_L m_2 \Longrightarrow$$
$$\langle\!|1, \langle c\rangle, m_1|\!\rangle \Downarrow m_1' \;\&\; \langle\!|1, \langle c\rangle, m_2|\!\rangle \Downarrow m_2' \;\&\; m_1' =_L m_2' \;\vee\; \langle\!|1, \langle c\rangle, m_1|\!\rangle \Uparrow \& \; \langle\!|1, \langle c\rangle, m_2|\!\rangle \Uparrow$$

4 Transformation

By performing a simple analysis while injecting `yield` commands, we are able to automatically enforce both termination-insensitive and termination-sensitive security. The transformation rules are presented in Figure 2. They have form $\Gamma \vdash c \hookrightarrow c'$, where command c is transformed into c' under Γ. In order to rule out *explicit flows* [DD77] via assignment, we ensure that expressions assigned to low variables may not depend

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

$$(\text{HCTX})\frac{\text{No } \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash c : \mathit{high}}$$

$$\frac{}{\Gamma \vdash \texttt{skip} \hookrightarrow \texttt{skip}; \texttt{yield}} \qquad \frac{}{\Gamma \vdash \texttt{yield} \hookrightarrow \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \sqsubseteq \Gamma(v)}{\Gamma \vdash v := e \hookrightarrow v := e; \texttt{yield}} \qquad \frac{\Gamma \vdash c_1 \hookrightarrow c_1' \qquad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma \vdash c_1 \hookrightarrow c_1' \qquad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1')\ \texttt{else}\ (\texttt{yield}; c_2')}$$

$$(\text{H-IF})\frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash c_1 : \mathit{high} \qquad \Gamma \vdash c_2 : \mathit{high}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2); \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma \vdash c \hookrightarrow c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ (\texttt{yield}; c')); \texttt{yield}}$$

$$(\text{H-W})\frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash c : \mathit{high}}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ c); \texttt{yield}}$$

$$\frac{\Gamma \vdash c \hookrightarrow c' \qquad \Gamma \vdash d_1 \hookrightarrow d_1' \qquad \ldots \qquad \Gamma \vdash d_n \hookrightarrow d_n'}{\Gamma \vdash \texttt{fork}(c, d_1 \ldots d_n) \hookrightarrow \texttt{fork}(c', d_1' \ldots d_n')}$$

Fig. 2. Transformation rules

on high data. This is enforced by demanding the type of the assigned variable to be at least as restrictive as the type of the expression that is to be assigned. Restrictiveness relation $\sqsubseteq$ on security levels is defined by $\mathit{low} \sqsubseteq \mathit{low}$, $\mathit{high} \sqsubseteq \mathit{high}$, $\mathit{low} \sqsubseteq \mathit{high}$ and $\mathit{high} \not\sqsubseteq \mathit{low}$. In order to reject *implicit flows* [DD77] via control flow, we guarantee that `if`'s and `while`'s with high guards may not have assignments to low variables in their bodies. These two techniques are well known [DD77, VSI96] and do not require code transformation.

The transformation injects `yield` commands in such a way that threads may not yield whenever their timing information depends on secret data. This is achieved by a requirement that `if`'s and `while`'s with high guards may not contain `yield` commands. In addition, such control flow statements may not contain `fork`. The rationale is that if secrets influence the number of threads, then it is possible for some schedulers to leak this difference via races of publicly-observable assignments [SS00, Sab03]. Rules H-IF and H-W enforce the above requirements. The rest of the transformation injects `yield` commands without significant restrictions (but with some obvious liveness guarantees for commands that do not branch on secrets).

The first lemma shows that commands typed under rule HCTX do not affect the low-security variables.

Lemma 1. *Given a command c and memories m and m' so that $\Gamma \vdash c : high$ and $\langle\!| c, m |\!\rangle \Downarrow^v m'$, then $m =_L m'$.*

The following theorem states that pools of transformed threads preserve low-equality on memories:

Theorem 1. *Given two (possibly empty) threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \Downarrow^v m_1'$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \Downarrow^w m_2'$, then $m_1' =_L m_2'$.*

As desired, the transformation enforces termination-insensitive security:

Corollary 1. *If $\Gamma \vdash c \hookrightarrow c'$ then c' satisfies termination-insensitive security.*

The transformation can be adopted to termination-sensitive security in a straightforward way. We write $\Gamma \vdash_{\text{TS}} c \hookrightarrow c'$ whenever $\Gamma \vdash c \hookrightarrow c'$ with the modifications that (i) rule H-W is not used, and (ii) rule HCTX is replaced by:

$$\text{(HCTX')}\ \frac{\text{No } \texttt{while}, \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash_{\text{TS}} c : \mathit{high}}$$

These modifications ensure that loops have low guards and that no loop may appear in an `if` statement with a high guard. These requirements are similar to those of Volpano and Smith [VS99] (except for the requirement on `fork`, which Volpano and Smith lack):

Lemma 2. *Given a command c so that $\Gamma \vdash c : high\ cmd$ for some security environment Γ in Volpano and Smith's type system [VS99]; and given command c' obtained from c by erasing occurrences of* `protect`*, we have $\Gamma \vdash_{TS} c' : high$.*

This allows us to connect the transformation to Volpano and Smith's type system:

Theorem 2. *If command c is typable under security environment Γ in Volpano and Smith's type system [VS99], then there exists command c'' such that $\Gamma \vdash_{TS} c' \hookrightarrow c''$, where c' is obtained from c by erasing occurrences of* `protect`.

We also achieve termination-sensitive security with the above modifications of the transformation. We firstly present some auxiliaries lemmas. The following lemma states that commands typed as *high* terminate and do not affect the low part of the memory:

Lemma 3. *Given a command c and memory m so that $\Gamma \vdash_{TS} c : high$, then $\langle\!| c, m |\!\rangle \Downarrow m'$ and $m =_L m'$.*

In order to show termination-sensitive security, we track the behavior of threadpools after executing some number of `yield` and `fork` commands. We capture this by relation $\rightarrow^*_{y,f}$ so that $cfg \rightarrow^*_{1,0} cfg'$ if there is cfg'' such that $cfg \rightarrow^* cfg''$ where no `yield`'s have been executed, $cfg'' \rightarrow cfg'$ results from executing a `yield` command; and $cfg \rightarrow^*_{y,f} cfg'$ if there is cfg'' such that $cfg \rightarrow^*_{y-1,f} cfg''$ (resp. $cfg \rightarrow^*_{y,f-1} cfg''$) and $cfg'' \rightarrow cfg'$ results from executing a `yield` (resp. `fork`) command.
The next two lemmas state that low-equivalence between memories is preserved after executing some number of `yield` and `fork` commands:

Lemma 4. *Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash_{TS} c_i \hookrightarrow c'_i$ where $c_i \in \vec{c}$ and $c'_i \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_1|\!\rangle \rightarrow^*_{1,0} \langle\!|\sigma', \langle\vec{c}\,''\rangle, m'_1|\!\rangle$, then there exists m'_2 such that $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_2|\!\rangle \rightarrow^*_{1,0} \langle\!|\sigma', \langle\vec{c}\,''\rangle, m'_2|\!\rangle$, and $m'_1 =_L m'_2$.*

Lemma 5. *Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, numbers σ, y, and f so that $y + f > 0$, $\Gamma \vdash_{TS} c_i \hookrightarrow c'_i$ where $c_i \in \vec{c}$ and $c'_i \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_1|\!\rangle \rightarrow^*_{y,f} \langle\!|\sigma', \langle\vec{c}\,''\rangle, m'_1|\!\rangle$, then there exists m'_2 such that $\langle\!|\sigma, \langle\vec{c}\,'\rangle, m_2|\!\rangle \rightarrow^*_{y,f} \langle\!|\sigma', \langle\vec{c}\,''\rangle, m'_2|\!\rangle$, and $m'_1 =_L m'_2$.*

The final theorem shows that the transformation eliminates the need for `protect`:

Theorem 3. *If $\Gamma \vdash_{TS} c \hookrightarrow c'$ then c' satisfies termination-sensitive security.*

5 Related work

An general overview of information flow controls for concurrent programs can be found in [SM03]. We briefly mention most closely related work. External timing-sensitive information-flow policies have been addressed for a multithreaded language [SS00], and extended with synchronization [Sab01], message passing [SM02], and declassification [MS04]. Type systems have been investigated for termination-sensitive flows in possibilistic [BC02] and probabilistic [VS99, Smi01, Smi03] settings. Recently, we have presented a type system that guarantees termination-insensitive security with respect to a class of deterministic schedulers [RS06]. Information flow via low determinism, prohibiting races on low variables from the outset, has been addressed in [ZM03, HWS06].

6 Conclusion

We have presented a transformation that prevents timing leaks via cooperative schedulers. We argue that this technique is general: it applies to a wide class of schedulers (although only a round-robin scheduler has been considered here for simplicity).
We have experimented with the GNU Pth [Eng05], a portable thread library for threads in user space. We have modified this library to allow the round-robin scheduling policy from Section 2. We have successfully applied the transformation for source-to-source translation of multithreaded programs without `yield`'s into GNU Pth programs. The security of this translation is ensured by Theorems 1 and 3.

Acknowledgment 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

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

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

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

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

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

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

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

[RS06] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[Sab03] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 260–273. Springer-Verlag, July 2003.

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

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

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

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

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

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

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

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

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

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, 1993.

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

Appendix

Lemma 1. Given a command c and memories m and m' so that $\Gamma \vdash c : high$ and $\langle\!| c, m |\!\rangle \Downarrow^v m'$, then $m =_L m'$.

Proof. By induction on v and case analysis on c. □

Theorem 1. Given two (possibly empty) threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \Downarrow^v m_1'$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \Downarrow^w m_2'$, then $m_1' =_L m_2'$.

Proof. The proof is done by induction on $v + w$ and case analysis on c_σ. Frequently, we need to identify a thread in a given position σ inside of a threadpool $\vec{c}$. In order to do that, we can represent the threadpool $\vec{c} = \langle c_1, c_2, \ldots, c_{\sigma-1}, c_\sigma, c_{\sigma+1}, \ldots, c_m \rangle$ as $c_\sigma \oplus \vec{c}_\sigma$, where $\vec{c}_\sigma = \langle c_1, c_2, \ldots, c_{\sigma-1}, c_{\sigma+1}, \ldots, c_m \rangle$.

$c_\sigma = \mathtt{if\ e\ then\ c_1\ else\ c_2}$) When $\Gamma \vdash e : low$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH afterward. The interesting case is when $\Gamma \vdash e : high$ and $\langle\!| e, m |\!\rangle \downarrow b_1$ and $\langle\!| e, m |\!\rangle \downarrow b_2$, where $b_1 \neq b_2$. Without loosing generality, let us suppose $b_1 = \mathtt{True}$ and $b_2 = \mathtt{False}$. We know that $c_\sigma' = (\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2); \mathtt{yield}$ by applying the transformation to c_σ. By inspecting the semantics for threadpools, we know that

$$\langle\!| \sigma, c_\sigma' \oplus \vec{c}\,'_\sigma, m_1 |\!\rangle \rightharpoonup \langle\!| \sigma, (c_1; \mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1 |\!\rangle$$
$$\langle\!| \sigma, c_\sigma' \oplus \vec{c}\,'_\sigma, m_2 |\!\rangle \rightharpoonup \langle\!| \sigma, (c_2; \mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2 |\!\rangle$$

By inspecting the transformation, we know that $(\Gamma \vdash c_i : high)_{i=1,2}$. By applying Lemma 1 to $(\Gamma \vdash c_i : high)_{i=1,2}$ and by inspecting the semantics for threadpools, we have

$$\langle\!| \sigma, (c_1; \mathtt{yield}) \oplus \vec{c}\,'_\sigma, m_1 |\!\rangle \rightharpoonup^* \langle\!| \sigma, (\mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1'' |\!\rangle \quad (1)$$
$$\langle\!| \sigma, (c_2; \mathtt{yield}) \oplus \vec{c}\,'_\sigma, m_2 |\!\rangle \rightharpoonup^* \langle\!| \sigma, (\mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2'' |\!\rangle \quad (2)$$

where $m_1 =_L m_1''$ and $m_2 =_L m_2''$. Additionally, we know by 1 and 2 that

$$\langle\!| \sigma, (\mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1'' |\!\rangle \rightharpoonup \langle\!| \sigma', \vec{c}\,'_\sigma, m_1'' |\!\rangle \quad (3)$$
$$\langle\!| \sigma, (\mathtt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2'' |\!\rangle \rightharpoonup \langle\!| \sigma', \vec{c}\,'_\sigma, m_2'' |\!\rangle \quad (4)$$

The result follows by applying IH on 3 and 4.

$c_\sigma = \texttt{while e do c}$) The interesting case is when $\Gamma \vdash e : high$, and $\langle\!| e, m |\!\rangle \downarrow b_1$ and $\langle\!| e, m |\!\rangle \downarrow b_2$, where $b_1 \neq b_2$. Without loosing generality, let us suppose $b_1 = \texttt{True}$ and $b_2 = \texttt{False}$. We know that $c'_\sigma = (\texttt{while}\ e\ \texttt{do}\ c); \texttt{yield}$ by applying the transformation to c_σ. By inspecting the semantics for threadpools and by applying Lemma 1, we have that

$$\langle\!| \sigma, c'_\sigma \oplus \vec{c}\,'_\sigma, m_1 |\!\rangle \rightharpoonup^* \langle\!| \sigma, c'_\sigma \oplus \vec{c}\,'_\sigma, m''_1 |\!\rangle \tag{5}$$

where $m''_1 =_L m_1$. The result follows from applying IH to configurations (5) and $\langle\!| \sigma, c'_\sigma \oplus \vec{c}\,'_\sigma, m_2 |\!\rangle$.

$c_\sigma = c_1; c_2$) We assume, by associativity of sequential composition, that c_1 is a single command. Thus, the proof consists on case analysis over c_1 and following the same structure of the proofs for single commands.

□

Corollary 1. If $\Gamma \vdash c \hookrightarrow c'$ then c' satisfies termination-insensitive security.

Proof. By applying Theorem 1 with $\vec{c} = \langle c \rangle$, $\vec{c}\,' = \langle c' \rangle$, and $\sigma = 1$. □

Lemma 2. Given a command c so that $\Gamma \vdash c : high\ cmd$ for some security environment Γ in Volpano and Smith's type system [VS99]; and given command c' obtained from c by erasing occurrences of `protect`, we have $\Gamma \vdash_{\text{TS}} c' : high$.

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

Theorem 2. If command c is typable under security environment Γ in Volpano and Smith's type system [VS99], then there exists command c'' such that $\Gamma \vdash_{\text{TS}} c' \hookrightarrow c''$, where c' is obtained from c by erasing occurrences of `protect`.

Proof. By simple structural induction on the type derivation of c.

$c_1; c_2$) We know that $\Gamma \vdash c_1 : \tau\ cmd$ and $\Gamma \vdash c_2 : \tau\ cmd$ by the type derivation of c. By IH, we have that there exists c'_1, c''_1, c'_2, and c''_2 such that $\Gamma \vdash_{\text{TS}} c'_1 \hookrightarrow c''_1$ and $\Gamma \vdash_{\text{TS}} c'_2 \hookrightarrow c''_2$, where c'_1 and c'_2 are respectively obtained from c_1 and c_2 by erasing the occurrences of `protect`. The result follows by taking $c'' = c''_1; c''_2$.

$\texttt{protect}(c_p)$) We have that $\Gamma \vdash c_p : \tau\ cmd$. By IH, we have that there exists there exists c'_p and c''_p such that $\Gamma \vdash_{\text{TS}} c'_p \hookrightarrow c''_p$, where c'_p is obtained from c_p by erasing the occurrences of `protect`. The result follows by taking $c' = c'_p$ and $c'' = c''_p$.

(CMD^-) **rule)** We know that

$$(CMD^-)\frac{\Gamma \vdash c : \tau_2\ cmd \qquad \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash c : \tau_1\ cmd}$$

By IH, we know that there exists c' and c'' such that $\Gamma \vdash_{\text{TS}} c' \hookrightarrow c''$, where c' is obtained from c by erasing the occurrences of `protect`. The result thus holds trivially.

(IF) **rule)** We know that

$$\text{(IF)}\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash c_1 : \tau\ cmd \qquad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \tau\ cmd}$$

Here, we have to cases.

$\tau = L$) By IH, we have that By IH, we have that there exists c_1', c_1'', c_2', and c_2'' such that $\Gamma \vdash_{\text{TS}} c_1' \hookrightarrow c_1''$ and $\Gamma \vdash_{\text{TS}} c_2' \hookrightarrow c_2''$, where c_1' and c_2' are respectively obtained from c_1 and c_2 by erasing the occurrences of `protect`. Moreover, $\Gamma \vdash_{\text{TS}} e : low$ by our transformation. The result follows by taking $c'' = \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1'')\ \texttt{else}\ (\texttt{yield}; c_2'')$.

$\tau = H$) Since the transformation does not have a subtyping rule for expression, we need to split the proof here in two more cases.

$\Gamma \vdash e : high$) By applying Lemma 2 to c_1 and c_2, we obtain that $\Gamma \vdash_{\text{TS}} c_1' : high$ and $\Gamma \vdash_{\text{TS}} c_2' : high$, where c_1' and c_2' are respectively obtained from c_1 and c_2 by erasing the occurrences of `protect`. The result follows by applying $(H - IF)$ rule in the transformation.

$\Gamma \vdash e : low$) Thus, the type derivation for the conditional has the following form.

$$\frac{\text{(SUBTYPE)}\ \dfrac{\Gamma \vdash e : L}{\Gamma \vdash e : H} \qquad \Gamma \vdash c_1 : H\ cmd \qquad \Gamma \vdash c_2 : H\ cmd}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : H\ cmd}$$

By IH, we have that there exists c_1', c_1'', c_2', and c_2'' such that $\Gamma \vdash_{\text{TS}} c_1' \hookrightarrow c_1''$ and $\Gamma \vdash_{\text{TS}} c_2' \hookrightarrow c_2''$, where c_1' and c_2' are respectively obtained from c_1 and c_2 by erasing the occurrences of `protect`. The result follows from $\Gamma \vdash e : low$ and taking $c'' = \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1'')\ \texttt{else}\ (\texttt{yield}; c_2'')$.

□

Lemma 3. Given a command c and memory m so that $\Gamma \vdash_{\text{TS}} c : high$, then $\langle\!| c, m |\!\rangle \Downarrow m'$ and $m =_L m'$.

Proof. By induction on the size of c. □

Lemma 4. Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, and number σ so that $\Gamma \vdash_{\text{TS}} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \rightarrow^*_{1,0} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.

Proof. By induction on the number of steps of $\rightarrow^*_{1,0}$ and case analysis on c_σ.

$\rightarrow^1_{1,0}$) The only possibilities are that $c_\sigma = \texttt{yield}$. The lemma trivially holds in this case.

$\rightarrow^{v+1}_{1,0}, v \geq 1$)

$c_\sigma = \texttt{if e then c}_1\texttt{ else c}_2$) When $\Gamma \vdash e : low$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH afterwards. The interesting case is when $\Gamma \vdash e : high$ and $\langle\!| e, m |\!\rangle \downarrow b_1$ and $\langle\!| e, m |\!\rangle \downarrow b_2$, where $b_1 \neq b_2$. Without loosing generality, let us suppose $b_1 =$ `True` and $b_2 =$ `False`. We know that

$$\langle\!| \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_1 |\!\rangle \rightharpoonup_{0,0} \langle\!| \sigma, (c_1; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_1 |\!\rangle \tag{6}$$
$$\langle\!| \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_2 |\!\rangle \rightharpoonup_{0,0} \langle\!| \sigma, (c_2; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_2 |\!\rangle \tag{7}$$

By H, we know that $\Gamma \vdash_{\text{TS}} c_i : high$. Thus, we can apply Lemma 1 to obtain that

$$\langle\!| \sigma, (c_1; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_1 |\!\rangle \rightharpoonup^*_{0,0} \langle\!| \sigma, (\texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_1^* |\!\rangle \tag{8}$$
$$\langle\!| \sigma, (c_2; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_2 |\!\rangle \rightharpoonup^*_{0,0} \langle\!| \sigma, (\texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_2^* |\!\rangle \tag{9}$$

where $m_1 =_L m_1^*$ and $m_2 =_L m_2^*$. By executing `yield`s in (8) and (9) and by H, we have that

$$\langle\!| \sigma', \vec{c}'_\sigma, m_1^* |\!\rangle \rightharpoonup^k_{1,0} \langle\!| \sigma'', \vec{c}'', m_1' |\!\rangle \tag{10}$$
$$\langle\!| \sigma', \vec{c}'_\sigma, m_2^* |\!\rangle \rightharpoonup^k_{1,0} \langle\!| \sigma'', \vec{c}'', m_2' |\!\rangle \tag{11}$$

where $k < v$. The result follows from applying IH to (10) and (11) together with $m_1 =_L m_1^*$ and $m_2 =_L m_2^*$.

$c_\sigma = c_1; c_2$) We assume, by associativity of sequential composition, that c_1 is a single command. Thus, the proof consists on case analysis over c_1 and following the same structure of the proofs for single commands.

□

Lemma 5. Given two non-empty threadpools $\vec{c}$ and $\vec{c}\,'$ of equal size, memories m_1 and m_2, numbers σ, y, and f so that $y + f > 0$, $\Gamma \vdash_{\text{TS}} c_i \hookrightarrow c_i'$ where $c_i \in \vec{c}$ and $c_i' \in \vec{c}\,'$, $m_1 =_L m_2$, and $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_1 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle$, then there exists m_2' such that $\langle\!| \sigma, \langle \vec{c}\,' \rangle, m_2 |\!\rangle \rightarrow^*_{y,f} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_2' |\!\rangle$, and $m_1' =_L m_2'$.

Proof. By induction on $y + f$, case analysis on c_σ, and by applying Lemmas 3 and 4 when necessary.

$y = 1, f = 0$) It holds by Lemma 4.

$y = 0, f = 1$) It cannot happen since executing a `fork` implies to executed `yield`s. Observe that the transformation rule for `fork` inserts at least one `yield` in c'.

$y + f = k + 1, k \geq 1$)

$c_\sigma = \texttt{if e then c}_1\texttt{ else c}_2$) When $\Gamma \vdash e : low$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH afterwards. The interesting case is when $\Gamma \vdash e : high$ and $\langle\!| e, m |\!\rangle \downarrow b_1$ and $\langle\!| e, m |\!\rangle \downarrow b_2$, where $b_1 \neq b_2$. Without loosing generality, let us suppose $b_1 =$ `True` and $b_2 =$ `False`. We know that

$$(\langle\!| \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_i |\!\rangle \rightharpoonup \langle\!| \sigma, (c_i; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_i |\!\rangle)_{i=1,2}$$

By H, we know that $(\Gamma \vdash_{\mathrm{TS}} c_i : \mathit{high})_{i=1,2}$. Thus, we can apply Lemma 3 to obtain that

$$(\langle\!| \sigma, (c_i; \texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_i |\!\rangle \rightharpoonup^* \langle\!| \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_i^* |\!\rangle)_{i=1,2}$$

By executing `yield`, we have that

$$(\langle\!| \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_i^* |\!\rangle \rightharpoonup \langle\!| \sigma^*, \vec{c}\,'_\sigma, m_i^* |\!\rangle)_{i=1,2}$$

By H, we know that

$$\langle\!| \sigma^*, \vec{c}\,'_\sigma, m_1^* |\!\rangle \rightharpoonup^*_{y-1,f} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle \tag{12}$$

The result follows by applying IH on (12), and because $(m_i =_L m_i^*)_{i=1,2}$.

$c_\sigma =$ `while e do c`) Loops with secrets on guards are not allowed by the transformation. Thus, the only possible case is when $\Gamma \vdash e : \mathit{low}$. The guard e needs to be evaluated to `True` since otherwise `yield` is executed only once, which contradicts the hypothesis. The proof for when $(\langle\!| e, m_i |\!\rangle \downarrow \texttt{True})_{i=1,2}$ consists on reducing the command `while` once and then apply IH.

$c_\sigma = \texttt{fork}(c, \vec{d})$) We know that $c'_\sigma = \texttt{fork}(c', \vec{d'})$, where $\Gamma \vdash c \hookrightarrow c'$ and $\Gamma \vdash \vec{d} \hookrightarrow \vec{d}\,'$. By executing the command `fork`, we have that

$$(\langle\!| \sigma, (\texttt{fork}(c', \vec{d}\,'))_\sigma \oplus \vec{c}\,'_\sigma, m_i |\!\rangle \rightharpoonup \langle\!| \sigma, (c')_\sigma \oplus \vec{c}\,'_\sigma \oplus \vec{d}\,', m_i |\!\rangle) \tag{13}$$

By H, we also know that

$$\langle\!| \sigma, (c')_\sigma \oplus \vec{c}\,'_\sigma \oplus \vec{d}\,', m_1 |\!\rangle \rightharpoonup^*_{y,f-1} \langle\!| \sigma', \langle \vec{c}\,'' \rangle, m_1' |\!\rangle \tag{14}$$

The result follows by (13) and by applying IH to (14).

$c_\sigma = c_1; c_2$) We assume, by associativity of sequential composition, that c_1 is a single command. Thus, the proof consists on case analysis over c_1 and following the same structure of the proofs for single commands.

□

Theorem 3. If $\Gamma \vdash_{\mathrm{TS}} c \hookrightarrow c'$ then c' satisfies termination-sensitive security.

Proof. By applying Lemma 5 with $\vec{c} = \langle c \rangle$, $\vec{c}\,' = \langle c' \rangle$, and $\sigma = 1$ and observing that a divergent configuration (originating from c') performs an infinite number of `yield`'s. □

CHAPTER

5

Closing Internal Timing Channels by Transformation

In Proceedings of the 11th Annual Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006. LNCS, Springer-Verlag.

Closing Internal Timing Channels by Transformation

Alejandro Russo[1], John Hughes[1], David Naumann[2], and Andrei Sabelfeld[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology, 412 96 Göteborg, Sweden, Fax: +46 31 772 3663
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken, New Jersey 07030, USA

Abstract. A major difficulty for tracking information flow in multithreaded programs is due to the *internal timing* covert channel. Information is leaked via this channel when secrets affect the timing behavior of a thread, which, via the scheduler, affects the interleaving of assignments to public variables. This channel is particularly dangerous because, in contrast to external timing, the attacker does not need to observe the actual execution time. This paper presents a compositional transformation that closes the internal timing channel for multithreaded programs (or rejects the program if there are symptoms of other flows). The transformation is based on spawning dedicated threads, whenever computation may affect secrets, and carefully synchronizing them. The target language features semaphores, which have not been previously considered in the context of termination-insensitive security.

1 Introduction

An active area of research is focused on information flow controls in multithreaded programs [SM03]. Multithreading opens new covert channels by which information can be leaked to an attacker. As a consequence, the machinery for enforcing secure information flow in sequential programs is not sufficient for multithreaded languages [SV98]. One particularly dangerous channel is the *internal timing* covert channel. Information is leaked via this channel when secrets affect the timing behavior of a thread, which, via the scheduler, affects the interleaving of assignments to public variables.
Suppose that h is a secret variable, and k and l are public ones. Assuming that $\|$ denotes parallel composition, consider a simple example of an internal timing leak:

$$\begin{array}{l} \texttt{if}\ h \geq k\ \texttt{then skip; skip else skip;} \\ l := 1 \end{array} \quad \| \quad \begin{array}{l} \texttt{skip;} \\ \texttt{skip;} \\ l := 0 \end{array} \qquad \text{(Internal timing leak)}$$

Under a one-step round-robin scheduler (and a wide class of other reasonable schedulers), if $h \geq k$ then by the time assignment $l := 1$ is reached in the first thread, the second thread has terminated. Therefore, the last assignment to execute is $l := 1$. On the other hand, if $h < k$ then by the time assignment $l := 0$ is reached in the second thread, the first thread has terminated. Therefore, the last assignment to execute is $l := 0$. Hence, the truth value of $h \geq k$ is leaked into l. Programs with dynamic thread creation are vulnerable to similar leaks. For example, a direct encoding of the example above is depicted in Fig. 1 (where $\texttt{fork}(c)$ spawns a new thread c).

This program also leaks whether $h \geq k$ is true, under many schedulers. Internal timing leaks are particularly dangerous because, in contrast to *external* timing, the attacker does not need to observe the actual execution time. Moreover, leaks similar to those considered so far can be magnified via loops as shown in Fig. 2 (where k, l, n, and p are public; and h is an n-bit secret integer). Each iteration of the loop leaks one bit of h. As a result, the entire value of h is copied into p. Although this example assumes a round-robin scheduler, similar examples can be easily constructed where secrets are copied into public variables under any fair scheduler [SV98].

```
fork(skip; skip; $l := 0$);
if $h \geq k$
  then skip; skip else skip;
$l := 1$
```

Fig. 1. Internal timing leak with `fork`

```
$p := 0$;
while $n \geq 0$ do
  $k := 2^{n-1}$;
  fork(skip; skip; $l := 0$);
  if $h \geq k$
    then skip; skip else skip;
  $l := 1$;
  if $l = 1$
    then $h := h - k; p := p + k$
    else skip;
  $n := n - 1$
```

Fig. 2. Internal timing leak magnified

Existing proposals to tackling internal timing flows heavily rely on the modification of run-time environment. (A more detailed discussion of related work is deferred to Section 8.) A series of work by Volpano and Smith [SV98, VS99, Smi01, Smi03] suggests a special `protect`(c) statement that, by definition, takes one atomic computation step with the effect of running command c to the end. Internal timing leaks are made invisible because `protect`()-based security typed systems ensure that computation that branches on secrets is wrapped by `protect`() commands. However, implementing `protect`() is a major challenge [SS00, Sab01, RS06a] because while a thread runs `protect`(), the other threads must be instantly blocked. Russo and Sabelfeld argue that standard synchronization primitives are not sufficient and resort to primitives for direct interaction with scheduler in order to enable instant blocking [RS06a]. However, a drawback of this approach (and, arguably, any approach that implements `protect`() by instant blocking) is that it relies on the modification of run-time environment: the scheduler must be able to immediately suspend all threads that might potentially assign to public variables while a protected segment of code is run, which limits concurrency in the program.

This paper eliminates the need for modifying the run-time environment for a class of round-robin schedulers. We give a transformation that closes internal timing leaks by spawning dedicated threads for segments of code that may affect secrets. There are no internal timing leaks in transformed programs because the timing for reaching assignments to public variables does not depend on secrets. The transformation carefully synchronizes the dedicated threads in order not to introduce undesired interleavings in the semantics of the original program. Despite the introduced synchronization, threads that operate on public data are not prevented from progress by threads that operate on secret data, which gives more concurrency than in [SV98, VS99, Smi01, Smi03, RS06a].

For a program with internal timing leaks under a particular deterministic scheduler, the elimination of leaks necessarily changes the interleavings and so possibly the final result. What thread synchronization allows us to achieve is refinement of results under nondeterministic scheduling: the result of the transformed program (under round-robin) is a possible result of the source program under nondeterministic scheduling. Although an attacker would seek to exploit information about the specific scheduler in use, good software engineering practice suggests that a program's functional behavior should not be dependent on specific properties of a scheduler beyond such properties as fairness.
The transformation does not reject programs unless they have symptoms that would already reject sequential programs [DD77, VSI96]. The transformation ensures that the rest of insecurities (due to internal timing) are repaired.
It is seemingly possible to remove internal timing leaks by applying the following naive transformation. Suppose a command (program) c only has two variables h and l to store a secret and a public value, respectively. Assume that c does not have insecurities other than due to internal timing (this can be achieved by disallowing explicit and implicit flows, defined later in the paper). Then the following program does not leak any information about h, while it computes output as intended for c (or diverges):

$$h_i := h;\ l_i := l;\ h := 0;\ c;\ bar;\ l_o := l;\ h := h_i;\ l := l_i;\ c;\ bar;\ l := l_o$$

where bar is a barrier command that ensures that all other threads have terminated before proceeding. This transformation suffers from at least two drawbacks. Firstly, the program c is run twice, which is inefficient. Secondly, it is hard to ensure that any kind of nondeterminism (e.g., due to the scheduler, random number generator, or input channels) in c is resolved in the same way in both copies. For example, the transformation does not scale up naturally when c uses input channels. It is not obvious how to communicate inputs between the two copies of the program.
Another attempt to remove internal timing leaks could be done by applying slicing techniques, which can automatically split the original program into low and high parts. Unfortunately, these techniques in presence of concurrency are not enough to preserve the semantics of the original program. The reason for that is simple: public variables, which are updated by threads, might affect the computation of secrets. Therefore, an explicit communication of public values to the high part is required.

2 Language

Although our technique is applicable to fully-fledged programming languages, we use a simple imperative language to formalize the transformation. The language includes a command $\mathtt{fork}((\lambda\vec{x}.c) \mathbin{@} \vec{e})$, which dynamically creates and runs a new thread with local variables $\vec{x}$ with initial values given by the expressions $\vec{e}$. When the list of local variables is empty, we sometimes use simpler notation: $\mathtt{fork}(c)$. The command c may also use the program's global variables. The transformation requires dynamically allocated semaphores, so these too are included in the language defined in this section. Without making it precise, we assume that each variable is of type integer or type semaphore. There are no expressions of type semaphore other than semaphore variables. A main program is a single command c, in the grammar of Fig. 3. Its free vari-

$$\begin{array}{l} c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{fork}((\lambda\vec{x}.c) \mathbin{@} \vec{e}) \\ \quad \mid \texttt{stop} \mid \mathit{sem} := \texttt{newSem}(n) \mid \texttt{wait}(\mathit{sem}) \mid \texttt{signal}(\mathit{sem}) \end{array}$$

Fig. 3. Command syntax (with x and sem ranging over variables, and n over integer literals)

$$\frac{(\vec{e}, m) \downarrow \vec{v}}{\langle\!| \texttt{fork}((\lambda\vec{x}.d) \mathbin{@} \vec{e}), m |\!\rangle h \overset{\lambda\vec{x}.d,\vec{v}}{\rightharpoonup} \langle\!| \texttt{stop}, m |\!\rangle h} \qquad \frac{(\mathit{sem}, m) \downarrow r \qquad h(r).\mathit{cnt} = 0}{\langle\!| \texttt{wait}(\mathit{sem}), m |\!\rangle h \overset{\otimes r}{\rightharpoonup} \langle\!| \texttt{stop}, m |\!\rangle h}$$

$$\frac{(\mathit{sem}, m) \downarrow r \qquad h(r).\mathit{cnt} > 0 \qquad h' = h[r.\mathit{cnt} := r.\mathit{cnt} - 1]}{\langle\!| \texttt{wait}(\mathit{sem}), m |\!\rangle h \rightharpoonup \langle\!| \texttt{stop}, m |\!\rangle h'}$$

$$\frac{(\mathit{sem}, m) \downarrow r}{\langle\!| \texttt{signal}(\mathit{sem}), m |\!\rangle h \overset{\odot r}{\rightharpoonup} \langle\!| \texttt{stop}, m |\!\rangle h}$$

$$\frac{i = \mathit{max}(\mathit{dom}(h)) + 1 \qquad h' = h \cup \{i \mapsto (\text{cnt} = n, \text{que} = \langle\rangle)\}}{\langle\!| s := \texttt{newSem}(n), m |\!\rangle h \rightharpoonup \langle\!| \texttt{stop}, m[s := i] |\!\rangle h'}$$

Fig. 4. Commands semantics

ables comprise the *globals* of the program. The *source language* is the subset in which there are no `stop` commands, no semaphore variables and therefore no semaphore allocations or operations. Moreover, the list of local variables in every `fork` must be empty. Locals are needed for the transformation, but locals in source code would complicate the transformation (because each source thread is split into multiple threads, and locals are not shared between threads).

3 Semantics

The formal semantics is defined in two levels: individual command and threadpool semantics. The small-step semantics for sequential commands is standard [Win93], and we thus omit these rules. The rules for concurrent commands are given in Fig. 4.
Configurations have the form $\langle\!| c, m |\!\rangle h$, where c is a command, m is a memory (mapping variables to their values), and h is a heap for dynamically allocated semaphores. The expression language does not include dereferencing of semaphore references, so evaluation of expressions does not depend on the heap. We write $(e, m) \downarrow n$ to say that n is the value of e in memory m. A *heap* is a finite mapping from semaphore references (which we take to be naturals) to records of the form $(\text{cnt} = n, \text{que} = ws)$ where n is a natural number and ws is the list of blocked thread states.
Let α range over the following *events*, which label command transitions for use in the threadpool semantics: $\odot r$, to indicate the semaphore at reference r is signaled; $\otimes r$, to indicate it is waited; or a pair $\lambda\vec{x}.c, \vec{v}$ where $\vec{v}$ is a sequence of values that match $\vec{x}$.

Threadpool configurations have the form $\langle\!|\langle(c_0,m_0)\dots(c_i,m_i)\dots(c_{n-1},m_{n-1})\rangle, g, h, j|\!\rangle$, where each (c_i,m_i) is the state of thread i which is not blocked, g maps global variables to their values, h is the heap, $j \in 0\dots n-1$ is the index of the thread that will take the next step. For all i, $dom(m_i)$ is disjoint from $dom(g)$. Numbering threads $0\dots n-1$ slightly simplifies some definitions related to round-robin scheduling.

The threadpool semantics is defined for any scheduler relation SC. We interpret $(i, n, n', i') \in SC$ to mean that i is the current thread taking a step, n is the current pool size, n' is the size of the pool after that step, and i' is the next thread chosen by the scheduler. This model is adequate to define a round-robin scheduler for which thread activation, suspension, and termination do not affect the interleaving of other threads, and also to model full nondeterminism. The fully nondeterministic scheduler ND is defined by $(i,n,n',i') \in ND$ if and only if $0 \le i < n$ and $0 \le i' < n'$.

A little care is needed with round-robin to maintain the order when threads are blocked or terminated. The definition relies on some details of the threadpool semantics, e.g., when a step by thread i removes a thread from the pool (by termination or blocking), that thread is i itself. Define the round-robin scheduler RR by $(i,n,n',i') \in RR$ if and only if $0 \le i < n$ and equation (1) holds.

$$\begin{aligned} i' &= i, && \text{if } n' < n \text{ and } i < n-1 \\ &= 0, && \text{if } n' < n \text{ and } i = n-1 \\ &= (i+1) \mathbf{\,mod\,} n', && \text{otherwise} \end{aligned} \tag{1}$$

The threadpool semantics is given in Fig. 5. Note that memories in command configurations are disjoint unions $m_i \cup g$, where m_i is the thread-local memory, and g is the global one. We write $h[r.\text{que} := (r.\text{que} :: (c,m))]$ to abbreviate an update of the record at r in h to change its que field by appending (c,m) at the tail. Although semaphores are stored in a heap, we streamline the semantics by not including a null reference. Thus, an initial heap is needed. It is defined to initialize semaphores to 1, which is an arbitrary choice. The security condition defined later refers to initial values for all global variables, for simplicity, but only integer inputs matter.

Definition 1. *The* initial heap of size k *is the mapping* h_k *with domain* $1\dots k$ *that maps each* i *to the semaphore state* $(cnt = 1, que = \langle\rangle)$*. Suppose that* k *of the globals have type semaphore. Given a global memory* g*, the* initial global memory g_k *agrees with* g *on integer variables, and the* i*th semaphore variable (under some enumeration) is mapped to* i *(*$i \in dom(h_k)$*).*

Define $(c,g) \Downarrow g'$ *if and only if* $\langle\!|(c,m), g_k, h_k|\!\rangle 0 \rightarrow^* \langle\!|, g', h'|\!\rangle j$*, for some* h' *and* j*, where* $\rightarrow^*$ *is the reflexive and transitive closure of the transition relation* $\rightarrow$*, and* m *is the empty function (since the initial thread* c *has no local variables).*

Note that the definitions of $\rightarrow^*$ and $\Downarrow$ depend on the choice of scheduler, but this is elided in the notation.

$$\frac{\langle c_i, m_i \cup g \rangle h \rightharpoonup \langle c_i', m_i' \cup g' \rangle h' \qquad (i, n, n, j) \in SC}{\langle\!| \ldots (c_i, m_i) \ldots, g, h |\!\rangle i \to \langle\!| \ldots (c_i', m_i') \ldots, g', h' |\!\rangle j}$$

$$\frac{c_i = \mathtt{stop} \qquad (i, n, n-1, j) \in SC}{\langle\!| \ldots (c_i, m_i) \ldots, g, h |\!\rangle i \to \langle\!| \ldots (c_{i-1}, m_{i-1})(c_{i+1}, m_{i+1}) \ldots, g, h |\!\rangle j}$$

$$\frac{\langle c_i, m_i \cup g \rangle h \overset{\lambda \vec{x}.d, \vec{v}}{\rightharpoonup} \langle c_i', m_i' \cup g' \rangle h' \qquad m = \{\vec{x} \mapsto \vec{v}\} \qquad (i, n, n+1, j) \in SC}{\langle\!| \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1}), g, h |\!\rangle i \to \langle\!| \ldots (c_i', m_i') \ldots (c_{n-1}, m_{n-1})(d, m), g', h' |\!\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g \rangle h \overset{\otimes r}{\rightharpoonup} \langle c_i', m_i' \cup g' \rangle h' \\ h'' = h'[r.\text{que} := (r.\text{que} :: (c_i', m_i'))] \qquad (i, n, n-1, j) \in SC\end{array}}{\langle\!| \ldots (c_i, m_i) \ldots, g, h |\!\rangle i \to \langle\!| \ldots (c_{i-1}, m_{i-1})(c_{i+1}, m_{i+1}) \ldots, g', h'' |\!\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g \rangle h \overset{\odot r}{\rightharpoonup} \langle c_i', m_i' \cup g' \rangle h' \\ h'(r).\text{que} = (c, m) :: ws \qquad h'' = h'[r.\text{que} := ws] \qquad (i, n, n+1, j) \in SC\end{array}}{\langle\!| \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1}), g, h |\!\rangle i \to \langle\!| \ldots (c_i', m_i') \ldots (c_{n-1}, m_{n-1})(c, m), g', h'' |\!\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g \rangle h \overset{\odot r}{\rightharpoonup} \langle c_i', m_i' \cup g' \rangle h' \\ h'(r).\text{que} = \langle\rangle \qquad h'' = h'[r.\text{cnt} := r.\text{cnt} + 1] \qquad (i, n, n, j) \in SC\end{array}}{\langle\!| \ldots (c_i, m_i) \ldots, g, h |\!\rangle i \to \langle\!| \ldots (c_i', m_i') \ldots, g', h'' |\!\rangle j}$$

Fig. 5. Threadpool semantics (for scheduler SC)

4 Security specification

Assume that all global non-semaphore variables are labeled with *low* or *high* security levels to represent public and secret data, respectively. We label all semaphore variables as high in the target code (recall that the source program has no semaphore variables). To define the security condition, it suffices to define *low equality* of global memories, written $g_1 =_L g_2$, to say that $g_1(x) = g_2(x)$ for all low variables x.

Definition 2. *Program c is secure if for all g_1, g_2 such that $g_1 =_L g_2$, if $(c, g_1) \Downarrow g_1'$ and $(c, g_2) \Downarrow g_2'$ then $g_1' =_L g_2'$, where $\Downarrow$ refers to the round-robin scheduler RR.*

The definition says that low equality of initial global memories implies low equality of final global memories. Note that this definition is termination-insensitive [SM03], in the sense that nonterminating runs are ignored.
Observe that the examples from the introduction are rejected by the above definition because the changes in the final values of low variables break low equality. Consider another example (where k and l are low; and h is high):

$$\mathtt{if}\ (h \geq k)\ \mathtt{then\ skip}; \mathtt{skip\ else\ skip} \parallel l := 0 \parallel l := 1$$

This program is secure because the timing of the first thread does not affect how the race between assignments in the second and third threads is resolved. This holds for round-robin schedulers that run each thread for a fixed number of steps (which covers the case of a one-step round-robin scheduler RR), machine instructions, or even calls to the `fork` primitive. Note, however, that schedulers that are able to change the order of scheduled threads depending on the number of live threads would not necessarily guarantee secure execution of the above program. For example, consider a scheduler that runs the first thread for two steps and then checks the number of live threads. If this number is two then the second thread is scheduled; otherwise the third thread is scheduled. This leaks the truth value of $h \geq k$ into l. Round-robin schedulers are not only practical but also in this sense more secure, which motivates our choice to adopt them in the semantics.

5 Transformation

In this section, we give a transformation that rules out *explicit* and *implicit* flows [DD77] and closes internal timing leaks under round-robin schedulers. The transformation rules have the form $\Gamma; w, s, a, b, m \vdash c \hookrightarrow c'$, where command c is transformed into c' under the security type environment Γ, which maps variables to their security levels, and special semaphore variables w, s, a, b, and m needed for synchronization. Moreover, a fresh high variable h_x is introduced for each low variable x in the source code. The transformation comprises the rules presented in Figs. 6 and 7, and the top-level rule:

$$\frac{\Gamma; w, s, a, b, m \vdash c \hookrightarrow c' \qquad w, s\ \textit{fresh}}{\Gamma \vdash c \hookrightarrow_t m := \mathtt{newSem}(1); a := \mathtt{newSem}(1); w := \mathtt{newSem}(1); \vec{h_l} := \vec{l}; c'} \tag{2}$$

where $\vec{h_l} := \vec{l}$ stands for copying all low variables l into fresh high variables h_l.
Define *low assignments* to be assignments to low variables. Explicit flows are prevented by not allowing high variables to occur in low assignments (see rule L-ASG). Define *high conditionals (loops)* to be conditionals (loops) that branch on expressions that contain high variables. Implicit flows for high conditionals and loops are prevented by rules of the form $\Gamma \vdash c \looparrowright c'$, where command c is transformed into c' under Γ. These rules guarantee that high `if`'s and `while`'s do not have assignments to low variables in their bodies. These rules for tracking explicit and implicit flows are adopted from security-type systems for sequential programs [VSI96].
As illustrated by previous examples, internal timing channels are introduced by low assignments after high conditionals and loops. To close these channels, the transformation introduces a `fork` whenever the source code branches on high data (see rules (H-IF) and (H-W)). Since such computations are now spawned in new threads, the number of executed instructions before low assignments does not depend on secrets. However, new threads open up possibilities for new races between high variables, which can unexpectedly change the semantics of the program. To ensure that such races are avoided, the transformation spawns dedicated threads for all computations that might affect high data (see rules (H-ASG) and (L-ASG)) and carefully places synchronization primitives in the transformed program. We will illustrate this, and other interesting aspects of the transformation, through examples.
Consider the following simple program that suffers from an internal timing leak:

$$(\mathtt{if}\ h_1\ \mathtt{then}\ \mathtt{skip}; \mathtt{skip}\ \mathtt{else}\ \mathtt{skip}); l := 1 \parallel d \tag{3}$$

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

$$\frac{}{\Gamma; w, s, a, b, m \vdash \texttt{skip} \hookrightarrow \texttt{skip}} \qquad \frac{(\Gamma; w, s, a, b, m \vdash c_i \hookrightarrow c_i')_{i=1,2}}{\Gamma; w, s, a, b, m \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$(\text{H-ASG})\ \frac{\Gamma \vdash e \looparrowright e' \qquad \Gamma(x) = \mathit{high}}{\begin{array}{ll}\Gamma; w, s, a, b, m \vdash x := e \hookrightarrow & s := \texttt{newSem}(0);\\ & \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); x := e'; \texttt{signal}(\hat{s}))\ @\ ws);\\ & w := s\end{array}}$$

$$(\text{L-ASG})\ \frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma(x) = \mathit{low} \qquad \Gamma \vdash e \looparrowright e'}{\begin{array}{ll}\Gamma; w, s, a, b \vdash x := e \hookrightarrow & s := \texttt{newSem}(0);\\ & \texttt{wait}(m);\ x := e;\ b := \texttt{newSem}(0);\\ & \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{wait}(\hat{w}); \texttt{wait}(\hat{a}); h_x := e';\\ & \qquad \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s}))@wsab);\\ & a := b; \texttt{signal}(m);\\ & w := s\end{array}}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma; w, s, a, b, m \vdash c \hookrightarrow c'}{\Gamma; w, s, a, b, m \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow \texttt{while}\ e\ \texttt{do}\ c'}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad (\Gamma; w, s, a, b, m \vdash c_i \hookrightarrow c_i')_{i=1,2}}{\Gamma; w, s, a, b, m \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}$$

$$(\text{H-IF})\ \frac{\begin{array}{cc}\Gamma \vdash e : \mathit{high} & \Gamma \vdash e \looparrowright e'\\ (\Gamma \vdash c_i \looparrowright c_i')_{i=1,2} & c_t = \texttt{if}\ e'\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'\end{array}}{\begin{array}{ll}\Gamma; w, s, a, b, m \vdash & \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\\ \hookrightarrow & s := \texttt{newSem}(0);\\ & \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); c_t; \texttt{signal}(\hat{s}))\ @\ ws);\\ & w := s\end{array}}$$

$$(\text{H-W})\ \frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash e \looparrowright e' \qquad \Gamma \vdash c \looparrowright c' \qquad c_t = \texttt{while}\ e'\ \texttt{do}\ c'}{\begin{array}{ll}\Gamma; w, s, a, b, m \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow & s := \texttt{newSem}(0);\\ & \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); c_t; \texttt{signal}(\hat{s}))\ @\ ws);\\ & w := s\end{array}}$$

$$\frac{\begin{array}{c}\Gamma; w', s', a, b, m \vdash d \hookrightarrow d' \qquad w', s'\ \mathit{fresh}\\ c_t = \texttt{fork}((\lambda \hat{w}\hat{s}\hat{w}'.\texttt{wait}(\hat{w}); \texttt{signal}(\hat{w}); \texttt{signal}(\hat{s}); \texttt{signal}(\hat{w}'))\ @\ \hat{w}\hat{s}w')\end{array}}{\begin{array}{ll}\Gamma; w, s, a, b, m \vdash \texttt{fork}(d) \hookrightarrow & s := \texttt{newSem}(0);\\ & \texttt{fork}((\lambda \hat{w}\hat{s}.w' := \texttt{newSem}(0); c_t; d')\ @\ ws);\\ & w := s\end{array}}$$

Fig. 6. Transformation rules I

$$\frac{}{\Gamma \vdash e \looparrowright e[h_x/x]_{\Gamma(x)=low}} \qquad \frac{}{\Gamma \vdash \texttt{skip} \looparrowright \texttt{skip}}$$

$$\frac{\Gamma(v) = high \quad \Gamma \vdash e \looparrowright e'}{\Gamma \vdash v := e \looparrowright v := e'} \qquad \frac{(\Gamma \vdash c_i \looparrowright c'_i)_{i=1,2}}{\Gamma \vdash c_1; c_2 \looparrowright c'_1; c'_2}$$

$$\frac{\Gamma \vdash e \looparrowright e' \quad (\Gamma \vdash c_i \looparrowright c'_i)_{i=1,2}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \looparrowright \texttt{if}\ e'\ \texttt{then}\ c'_1\ \texttt{else}\ c'_2}$$

$$\frac{\Gamma \vdash d \looparrowright d'}{\Gamma \vdash \texttt{fork}(d) \looparrowright \texttt{fork}(d')} \qquad \frac{\Gamma \vdash e \looparrowright e' \quad \Gamma \vdash c \looparrowright c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \looparrowright \texttt{while}\ e'\ \texttt{do}\ c'}$$

Fig. 7. Transformation rules II

where d abbreviates command $\texttt{skip}; \texttt{skip}; l := 0$. The assignment $l := 1$ may be reached in three or two steps depending on h_1. However, by spawning the high conditional in a new thread, the number of instructions to execute it will no longer affect when $l := 1$ is reached. More precisely, program (3) can be rewritten as $\texttt{fork}(\texttt{if}\ h_1\ \texttt{then}\ \texttt{skip}; \texttt{skip}; \texttt{else}\ \texttt{skip}); l := 1 \parallel d$, where internal timing leaks are not possible. From now on, we assume that the initial values of l and h_2 are always 0. Suppose now that we modify program (3) by:

$$(\texttt{if}\ h_1\ \texttt{then}\ h_2 := 2 * h_2 + l; \texttt{skip}\ \texttt{else}\ \texttt{skip}); l := 1 \parallel d \tag{4}$$

where the final value of h_2 is always 0. This code still suffers from an internal timing leak. Unfortunately, by putting a `fork` around the `if` as before, we introduce 1 as a possible final value for h_2, which was not possible in the original code. This discrepancy originates from an undesired new interleaving of the rewritten program: $l := 1$ can be computed before $h_2 := 2 * h_2 + l$. To prevent such an interleaving, we introduce fresh high variables for every low variable in the code. We call this kind of new variables *high images* of low variables. Since low variables are only read, and not written, by high conditional and loops, it is possible to replace low variables inside of high contexts by their corresponding high images. Then, every time that low variables are updated, their corresponding images will do so but in due course. To illustrate this, let us rewrite the left side of program (4) as in (5). Variable h_l is the corresponding high image of low variable l. Two dedicated threads are spawned with different local snapshots of w and s, written as $\hat{w}$ and $\hat{s}$, respectively. The second dedicated thread, which updates the high image of l to

```
w := newSem(1);   //initialization from top-level rule (2)
s := newSem(0);
fork((λŵŝ.wait(ŵ); (if h₁ then h₂ := 2 * h₂ + h_l; skip
                          else skip); signal(ŝ))
    @ws)
w := s
l := 1; s := newSem(0);
fork((λŵŝ.wait(ŵ); h_l := 1; signal(ŝ)) @ ws)
w := s
```
(5)

1, waits ($\texttt{wait}(\hat{w})$) for the first one to finish, and the first one indicates when the second one should start ($\texttt{signal}(\hat{s})$). By doing so, and by properly updating w and s in the main thread, the command $h_l := 1$ is never executed before the `if` statement. Note that the first dedicated thread does not need to synchronize with previous ones. Hence, the top-level transformation rule, presented at the beginning of the section, initializes the semaphore w to 1.

The thread d also needs to be modified to include an update to h_l. Let us rewrite d as follows:

$$\begin{array}{l} w_d := \texttt{newSem}(1); \texttt{skip}; \texttt{skip}; \\ l := 0; s_d := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w_d}\hat{s_d}.\texttt{wait}(\hat{w_d}); h_l := 0; \texttt{signal}(\hat{s_d}))@w_d s_d); \\ w_d := s_d \end{array} \tag{6}$$

Semaphore variables w_d and s_d do not play any important role here, since just one dedicated thread is spawned. Note that if we run programs (5) and (6) in parallel, it might be possible that the updates of low variables happen in a different order than the updates of their corresponding high images. In order to avoid this, we introduce three global semaphores, called a, b, and m. The final transformed code is shown in Fig. 8, where c'_1 runs in parallel with d'_1. Semaphore variables a and b ensure that the queuing processes update high images in the same order as the low assignments occur. Since a and b are globals, we protect their access with the global semaphore m. As in the original program, h_2 can only have the final value 0. From now on, we assume that the semaphore a is allocated and initialized with value 1 .

Let us modify program (4) by adding assignments to high and low variables:

$$(\texttt{if}\ h_1\ \texttt{then}\ h_2 := 2 * h_2 + l; \texttt{skip}\ \texttt{else}\ \texttt{skip}); l := 1; h_2 := h_2 + 1; l := 3 \parallel d \tag{7}$$

The final value of h_2 is 1. As before, this code still suffers from internal timing leaks. By putting `fork`'s around high conditionals and introducing updates for high images as in program (5), we would introduce 2 as a new possible final value for h_2, when h_1 is positive. The new value arises from executing $h_2 := h_2 + 1$ before the `if` statement.

In order to remove this race, we use synchronization to guarantee that computations on high data are executed in the same order as they appear in the original code. However, this synchronization should not lead to recreating timing leaks: waiting for the `if` to finish before executing $h_2 := h_2 + 1; l := 3$ would imply that the timing of the low assignment $l := 3$ could depend on h_1. We resolve this problem by spawning dedicated threads for assignments to high variables and synchronizing, via semaphores, these threads with other threads that either read from or write to high data. The dedicated thread to compute $h_2 := h_2 + 1$ will wait until the last dedicated thread in c'_1 finishes. The transformed code

$$\begin{array}{ll} c'_2 : & c'_1; s := \texttt{newSem}(0); \\ & \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); h_2 := h_2 + 1; \texttt{signal}(\hat{s})) \\ & \qquad @ws; \\ & w := s; \\ & \texttt{wait}(m); l := 3; b := \texttt{newSem}(0); \\ & \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{wait}(\hat{w}); \texttt{wait}(\hat{a}); h_l := 3; \\ & \qquad \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s}))@wsab); \\ & a := b; \texttt{signal}(m); \\ & \parallel d'_1 \end{array} \tag{8}$$

```
$c_1'$ : $w :=$ newSem$(1)$;
    $s :=$ newSem$(0)$;
    fork$((\lambda\hat{w}\hat{s}.$wait$(\hat{w})$;
            if $h_1$ then $h_2 := 2 * h_2 + h_l$;
                    skip;
                else skip;
              signal$(\hat{s}))@ws)$;
    $w := s$
    $s :=$ newSem$(0)$;
    wait$(m)$; $l := 1$; $b :=$ newSem$(0)$;
    fork$((\lambda\hat{w}\hat{s}\hat{a}\hat{b}.$wait$(\hat{w})$; wait$(\hat{a})$; $h_l := 1$;
          signal$(\hat{b})$; signal$(\hat{s}))@wsab)$;
    $a := b$; signal$(m)$;
    $w := s$
```

```
$d_1'$ : $w_d :=$ newSem$(1)$;
    skip; skip;
    $s_d :=$ newSem$(0)$;
    wait$(m)$; $l := 0$; $b :=$ newSem$(0)$;
    fork$((\lambda\hat{w}_d\hat{s}_d\hat{a}\hat{b}.$wait$(\hat{w}_d)$; wait$(\hat{a})$;
          $h_l := 0$; signal$(\hat{b})$; signal$(\hat{s}_d))$
          $@w_d s_d ab)$;
    $a := b$; signal$(m)$;
    $w_d := s_d$
```

Fig. 8. Transformed code for program (4)

is shown in (8). Note that spawned dedicated threads are executed in the same order as they appear in the main thread.
Let us modify program (7) to introduce a `fork` as follows:

$$\begin{array}{l} \texttt{if } h_1 \texttt{ then } h_2 := 2 * h_2 + l; \texttt{skip else skip}; \\ l := 1; h_2 := h_2 + 1; l := 3; \\ \texttt{fork}(h_2 := 5) \parallel d \end{array} \tag{9}$$

The final value of h_2 is 5. However, the rewritten program will spawn several dedicated threads: for the conditional, for updating high images, $h_2 := h_2 + 1$, and $h_2 := 5$, which need to be synchronized. In particular, $h_2 := 5$ cannot be executed before $h_2 := h_2 + 1$ finishes. Thus, we need to synchronize dedicated threads in the main thread with the dedicated threads from their children. This is addressed by the transformation as follows:

$$\begin{array}{l} c_2'; \\ s := \texttt{newSem}(0); \\ \texttt{fork}((\lambda\hat{w}\hat{s}.w' := \texttt{newSem}(0); \\ \qquad \texttt{fork}((\lambda\underline{\hat{w}}\underline{\hat{s}}\hat{w}'.\texttt{wait}(\underline{\hat{w}}); \texttt{signal}(\underline{\hat{w}}); \texttt{signal}(\hat{s}); \texttt{signal}(\hat{w}'))@\hat{w}\hat{s}w'); d^*)\,@ws); \\ w := s; \parallel d_1' \end{array} \tag{10}$$

where d^* spawns a new thread that waits on w' to perform $h_2 := 5$. In order to be able to receive a signal on w', it is necessary to firstly receive a signal on $\underline{\hat{w}}$, which can be only done after computing $h_2 := h_2 + 1$. Note that the transformation spawns a new thread to wait on $\underline{\hat{w}}$ in order to avoid recreating timing leaks. When a `fork` occurs inside a loop in the source program, there is potentially a number of dynamic threads that need to wait for the previous computation on high data to finish. This is resolved by passing-the-baton technique: whichever thread receives a signal first (`wait`$(\underline{\hat{w}})$) passes it to another thread (`signal`$(\underline{\hat{w}})$).
The examples above show how to close internal timing leaks by spawning dedicated threads that perform computation on high data. We have seen that some synchronization is needed to avoid producing different outputs than intended in the original pro-

```
$hotel_l := nextHotel();$
$hotelLoc_l := getHotelLocation(hotel_l);$
$d_h := distance(hotelLoc_l, userLoc_h);$
$closest_h := hotel_l;$
while $(moreHotels?())$ do
        $hotel_l := nextHotel();$
        $hotelLoc_l := getHotelLocation(hotel_l);$
        $d'_h := distance(hotelLoc_l, userLoc_h);$
        if $(d'_h < d_h)$ then $d_h := d'_h; closest_h := hotel_l$
                    else skip
        $i_h := 0;$

while $(moreTypeRooms?(closest_h))$ do
        $type_h := nextTypeRoom(closest_h);$
        $showTypeRoom(type_h, i_h);$
        $i_h := i_h + 1;$
```

Fig. 9. Geo-localization example

gram. Transformed programs introduce performance overhead related to synchronization. This overhead comes as a price for not modifying the run-time environment when preventing internal timing leaks.

6 Geo-localization example

Inspired by a scenario from mobile computing [Mob06], we give an example of closing timing leaks in a realistic setting. Modern mobile phones are able to compute their geographical positions. The widely used MIDP profile [JSR02] for mobile devices includes API support for obtaining the current position of the handset [JSR03]. Furthermore, geo-localization can be approximated by using the identity of the current base station and the power of its signal. It is desirable that such information can only be used by trusted parties.
Consider the code fragment in Fig. 9. This fragment is part of a program that runs on a mobile phone. Such a program typically uses dynamic thread creation (which is supported by MIDP) to perform time-consuming computation (such as establishing network connections) in separate threads [Knu02, Mah04].
The program searches for the closest hotel in the area where the handset is located. Once found, it displays the types of available rooms at that hotel. Variables have subscripts indicating their security levels (l for low and h for high). Suppose that $hotel_l$ and $hotelLoc_l$ contain the public name and location for a given hotel, respectively. The location of the mobile device is stored in the high variable $userLoc_h$. Variables d_h and d'_h are used to compute the distance to a given hotel. Variable $closest_h$ stores the location of the closest hotel in the area. Variable i_h is used to index the type of rooms at the closest hotel. Variable $type_h$ stores a room type, i.e., single, double, etc.

Function $\mathit{nextHotel}()$ returns the next available hotel in the area (for simplicity, we assume there is always at least one). Function $\mathit{getHotelLocation}()$ provides the location of a given hotel, and function $\mathit{distance}()$ computes the distance between two locations. Function $\mathit{moreHotels?}()$ returns true if there are more hotels for $\mathit{nextHotel}()$ to retrieve. Function $\mathit{moreTypeRooms?}()$ returns true if there are more room types for $\mathit{nextTypeRoom}()$. Function $\mathit{showTypeRoom}()$ displays room types on the screen.
This code may leak information about the location of the mobile phone through the internal timing covert channel. The source of the problem is a conditional that branches on secret data, where the `then` branch performs two assignments while the `else` branch only `skip`. However, internal timing leaks can be closed by the transformation given in Section 5 (provided the transformed program runs under a round-robin scheduler). This example highlights the permissiveness of the transformation. For instance, the type systems by Boudol and Castellani [BC01, BC02] reject the example because both high conditionals and low assignments appear in the body of a loop. Transformations in [SS00, KM06] also reject the example due to the presence of a high loop in the code.

7 Soundness

This section shows that transformed programs are secure. It also states that transformed programs refine source programs in a suitable sense. The details of the proofs for lemmas and theorems shown in this section are to appear in an accompanying technical report.

Security We identify two kinds of threads. *High* threads are dedicated threads introduced by the transformation and threads in the source program spawned inside a high conditional or a high loop. Other threads are *low* threads. We designate high threads by arranging that they have a distinguished local variable called $\hbar$. It is not difficult to modify the transformation in Section 5 to guarantee this.
In order to prove non-interference under round-robin schedulers, we firstly need to exploit some properties of programs produced by the transformation.

Definition 3. *A command c is* syntactically secure *provided that (i) there are no explicit flows, i.e., assignments $x := e$ with high e and low x; (ii) each low thread,* $\texttt{fork}((\lambda\vec{x}.c')\,@\,\vec{e})$*, in c satisfies the following: there are no high conditionals or high loops or* $\texttt{signal}()$ *or* $\texttt{wait}()$ *operations related to synchronize high threads, except inside high threads forked in c'; and (iii) in high threads, there are neither low assignments nor forks of low threads.*

Lemma 1. *If $\Gamma \vdash_t c \hookrightarrow c'$ then c' is syntactically secure.*

We let γ and δ range over threadpool configurations. We assume, for convenience in the notation, that $\gamma = \langle\!|(c_0, m_0)\ldots, g, h|\!\rangle j$. We also define $\gamma.\mathit{pool} = \langle(c_0, m_0)\ldots\rangle$, $\gamma.\mathit{globals} = g$, $\gamma.\mathit{heap} = h$, and $\gamma.\mathit{next} = j$. A program configuration γ is called *syntactically secure* if every command in $\gamma.\mathit{pool}$ and every command in a waiting queue of $\gamma.\mathit{heap}$ is syntactically secure.

A thread configuration (c, m) is low, noted $low?(m)$, if and only if $\hbar \notin dom(m)$. Define $low?(i, \gamma)$ if and only if the ith thread in $\gamma.pool$ is low. Define γ_L as the subsequence of thread configurations (c_i, m_i) in $\gamma.pool$ that are low. For each thread configuration $(c_i, m_i) \in \gamma$ that is low, define $lowpos(i, \gamma)$ (and, for simplicity in the notation, $lowpos(i, \gamma.pool)$) to be the index of the thread but in γ_L. The key property of a round-robin scheduler is that the next low thread to be scheduled is independent of the values of global or local variables, the states of high threads (running or blocked), and even the number of high threads in the configuration. We can formally capture this property as follows. Define $nextlow(\gamma) = j \ \mathbf{mod}\ (\#\gamma.pool)$ where j is the least number such that $j \geq \gamma.next$ and $low?(j \ \mathbf{mod}\ (\#\gamma.pool), \gamma)$.

Definition 4 (Low equality). *Define $P =_L P'$ for threadpools $P = \langle (c_1, m_1) \ldots \rangle$ and $P' = \langle (c'_1, m'_1) \ldots \rangle$ (not necessarily the same length) if and only if $c_i \equiv c'_j$ for all i, j such that $low?(m_i)$, $low?(m'_j)$, and $lowpos(i, P) = lowpos(j, P')$. Define $\gamma =_L \delta$ if and only if γ and δ are syntactically secure, $\gamma.globals =_L \delta.globals$, $\gamma.pool =_L \delta.pool$, $lowpos(nextlow(\gamma), \gamma) = lowpos(nextlow(\delta), \delta)$, and all threads blocked in $\gamma.heap$ and $\delta.heap$ are high.*

Theorem 1. *Let γ and δ be configurations such that $\gamma =_L \delta$. If $\gamma \to^* \gamma'$ and $\delta \to^* \delta'$ where γ', δ' are terminal configurations, then $\gamma' =_L \delta'$. Here $\to^*$ refers to the semantics using the round-robin scheduler RR.*

Corollary 1 (Security). *If $\Gamma \vdash c \hookrightarrow_t c'$ then c' is secure under round-robin scheduling.*

Refinement For programs produced by our transformation, the result from a round-robin computation from any initial state is a result from the original program using the fully nondeterministic scheduler. In fact, any interleaving of the transformed program matches some interleaving of the original code. Then, we have the following claim:

Claim 1. *Suppose $\Gamma \vdash c \hookrightarrow_t c'$ and g'_1 and g'_2 are global memories for c' such that $(c', g'_1) \Downarrow g'_2$ using the nondeterministic scheduler ND. Let g_1 and g_2 be the restrictions of g'_1 and g'_2 to the globals of c. Then $(c, g_1) \Downarrow g_2$ using ND.*

8 Related work

Variants of possibilistic noninterference have been explored in process-calculus settings [HVY00, FG01, Rya01, HY02, Pot02], but without considering the impact of scheduling.

As discussed in the introduction, a series of work by Volpano and Smith [SV98, VS99, Smi01, Smi03] suggests a special `protect`(c) statement to hide the internal timing of command c in the semantics. In contrast to this work, we are not dependent on the randomization of the scheduler. To the best of our knowledge, no proposals for `protect`$()$ implementation avoid significantly changing the scheduler (unless the scheduler is cooperative [RS06b]).

Boudol and Castellani [BC01, BC02] suggest explicit modeling of schedulers as programs. Their type systems, however, reject source programs where assignments to public variables follow computation that branches on secrets.

Smith and Thober [ST06] suggest a transformation to split a program into high and low components. Jif/split [ZCMZ03] partitions sequential programs into distributed code on different hosts. However, the main focus is on security when some trusted hosts are compromised. Neither approach provides any formal notion of security.

A possibility to resolve the internal timing problem is by considering external timing. Definitions sensitive to external timing consider stronger attackers, namely those that are able to observe the actual execution time. External timing-sensitive security definitions have been explored for multithreaded languages by Sabelfeld and Sands [SS00] as well as languages with synchronization [Sab01] by Sabelfeld and message passing [SM02] by Sabelfeld and Mantel. Typically, padding techniques [Aga00, SS00, KM06] are used to ensure that the timing behavior of a program is independent of secrets. Naturally, a stronger attacker model implies more restrictions on programs. For example, loops branching on secrets are disallowed in the above approaches. Further, padding might introduce slow-down and, in the worst case, nontermination.

Another possibility to prevent internal timing leaks in programs is by disallowing any races on public data, as pursued by Zdancewic and Myers [ZM03] and improved by Huisman et al. [HWS06]. However, such an approach rejects innocent programs such as $l := 0 \parallel l := 1$ where l is a public variable.

9 Conclusion

We have presented a transformation that closes internal timing leaks in programs with dynamic thread creation. In contrast to existing approaches, we have not appealed to nonstandard semantics (cf. the discussion on `protect`()) or to modifying the run-time environment (cf. the discussion on interaction with schedulers). Importantly, the transformation is not overrestrictive: programs are not rejected unless they have symptoms of flows inherent to sequential programs. The transformation ensures that the rest of insecurities (due to internal timing) are repaired. Our target language includes semaphores, which have not been considered in the context of termination-insensitive security.

Future work includes introducing synchronization and declassification primitives into the source language and improving the efficiency of the transformation: instead of dynamically spawning dedicated threads, one could refactor the program into high and low parts and explicitly communicate low data to the high part, when needed (and high data to the low part, when prescribed by declassification).

Acknowledgments This work was funded in part by the Swedish Emergency Management Agency and in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 Mobius project.

References

[Aga00] J. Agat. Transforming out timing leaks. In *Proc. POPL'02*, pages 40–53, January 2000.

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

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

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

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

[HVY00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

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

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

[JSR02] JSR 118 Expert Group. Mobile information device profile (MIDP), version 2.0. Java specification request, Java Community Process, November 2002.

[JSR03] JSR 179 Expert Group. Location API for J2ME. Java specification request, Java Community Process, September 2003.

[KM06] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *FAST'05*, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

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

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

[Mob06] Report on resource and information flow security requirements, March 2006. Deliverable D1.1 of the EU IST FET GC2 MOBIUS project, `http://mobius.inria.fr/`.

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

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

[RS06b] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. PSI'06*, volume 4378 of *LNCS*. Springer-Verlag, June 2006.

[Rya01] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *FOSAD*, volume 2171 of *LNCS*. Springer-Verlag, 2001.

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. PSI'01*, volume 2244 of *LNCS*. Springer-Verlag, July 2001.

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

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

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

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

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

[ST06] Scott F. Smith and Mark Thober. Refactoring programs to secure information flows. In *PLAS '06*, pages 75–84, New York, NY, USA, 2006. ACM Press.

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

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

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

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[ZCMZ03] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

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

CHAPTER

6

A Library for Secure Multi-threaded Information-Flow in Haskell

In Proceedings of the 20th IEEE Computer Security Foundations Symposium, Venice, Italy, July 6-8, 2007. IEEE Computer Society Press.

A Library for Secure Multithreaded Information-Flow in Haskell

Ta-chung Tsai, Alejandro Russo, and John Hughes

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

Abstract. Li and Zdancewic have recently proposed an approach to provide information-flow security via a library rather than producing a new language from the scratch. They show how to implement such a library in Haskell by using arrow combinators. However, their approach only works with computations that have no side-effects. In fact, they leave as an open question how their library, and the mechanisms in it, need to be modified to consider these kind of effects. Another absent feature in the library is support for multithreaded programs. Information-flow in multithreaded programs still remains as a challenge, and no support for that has been implemented yet. In this light, it is not surprising that the two main stream compilers that provide information-flow security, Jif and FlowCaml, lack support for multithreading.
Following ideas taken from literature, this paper presents an extension to Li and Zdancewic's library that provides information-flow security in presence of reference manipulation and multithreaded programs. Moreover, an online-shopping case study has been implemented to evaluate the proposed techniques. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice. To the best of our knowledge, this is the first implemented tool to guarantee information-flow security in concurrent programs and the first implementation of a case study that involves concurrency and information-flow policies.

1 Introduction

Language-based information flow security aims to guarantee that programs do not leak confidential data. It is commonly achieved by some form of static analysis which rejects programs that would leak, before they are run. Over the years, a great many such systems have been presented, supporting a wide variety of programming constructs [SM03]. However, the impact on programming practice has been rather limited.
One possible reason is that most systems are presented in the context of a simple, elegant, and minimal language, with a well-defined semantics to make proofs of soundness possible. Yet such systems cannot immediately be adopted by programmers—they must first be embedded in a real programming language with a real compiler, which is a major task in its own right. Only two such languages have been developed—Jif [Mye99, MZZ$^+$06] (based on Java) and FlowCaml [PS02, Sim03] (based on Caml).
Yet when a system implementor chooses a programming language, information flow security is only one factor among many. While Jif or FlowCaml might offer the desired

security guarantees, they may be unsuitable for other reasons, and thus not adopted. This motivated Li and Zdancewic to propose an alternative approach, whereby information flow security is provided via a *library* in an existing programming language [LZ06]. Constructing such a library is a much simpler task than designing and implementing a new programming language, and moreover leaves system implementors free to choose any language for which such a library exists.

Li and Zdancewic showed how to construct such a library for the functional programming language Haskell. The library provides an abstract type of secure programs, which are composed from underlying Haskell functions using operators that impose information-flow constraints. Secure programs are *certified*, by checking that all constraints are satisfied, before the underlying functions are invoked—thus guaranteeing that no secret information leaks. While secure programs are a little more awkward to write than ordinary Haskell functions, Li and Zdancewic argue that typically only a small part of a system need manipulate secret data—for example, an authentication module—and only this part need be programmed using their library.

However, Li and Zdancewic's library does impose quite severe restrictions on what a secure program fragment may do. In particular, these fragments may have no effects of any sort, since the library only tracks information flow through the inputs and outputs of each fragment. While absence of side-effects can be guaranteed in Haskell (via the type system), this is still a strong restriction. Our purpose in this paper is to show that the same idea can be applied to support secure programs with a much richer set of effects—namely updateable references in the presence of (cooperative) concurrency. The underlying methods we use—an information-flow type-system for references, a restriction on the scheduler—are taken from the literature; what we show here is how to *implement* them for a real programming language following Li and Zdancewic's approach.

The rest of this paper is structured as follows. In the next section we explain Li and Zdancewic's approach in more detail. One restriction of their approach is that data-structures are assigned a *single* security level—so if any part of the output of a secure program is secret, then the entire output must be classified as secret. We need to lift this restriction in our work, allowing data-structures with mixed security levels, and in Section 3 we show how. This enables us to add references in Section 4. We then introduce concurrency, reviewing approaches to secure information flow in this context in Section 5, in particular ways to close the *internal timing* covert channel, and in Section 6 we describe the implementation of our chosen approach. In Section 7 we present a concurrent case study involving online shopping. With no countermeasures, an attack based on internal timing leaks can obtain a credit-card number with high probability in about two minutes. We show that our library successfully defends against this attack. Finally, in Section 8, we draw our conclusions.

2 Encoding information-flow in Haskell

Li and Zdancewic's approach represents secure program fragments as *arrows* in Haskell [Hug00]. Arrows can be visualised as dataflow networks, mapping inputs on the left to outputs on the right. Arrows are constructed from Haskell functions using combinators,

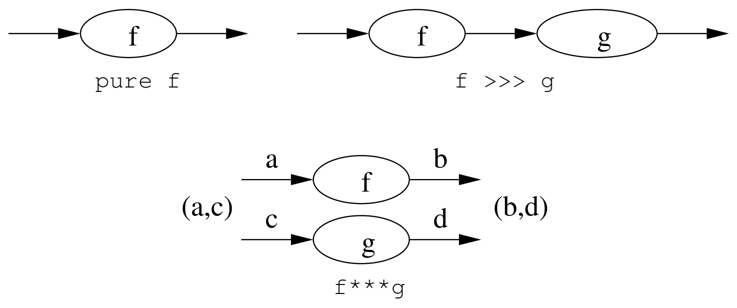

Fig. 1. Basic arrow combinators.

of which the most important are illustrated in Figure 1—`pure` converts a Haskell function to an arrow, `(>>>)` sequences two arrows, and `(***)` pairs arrows together. Any required left-to-right static dataflow can be implemented using these combinators—for example, an arrow that computes the average of a list could be constructed as

```
squareA = pure tee >>>
          (pure sum *** pure length) >>>
          pure divide
  where tee x = (x,x)
        divide (x,y) = x/y
```

Its effect is illustrated in Figure 2. To express a dynamic choice between two arrows, there is an additional combinator `f|||g`, whose input is of Haskell's sum type:

```
data Either a b = Left a | Right b
```

Its effect is illustrated in Figure 3.
Haskell allows any suitable type to be declared to be an arrow, by providing implementations for the basic arrow combinators. This is usually used to encapsulate some kind of effects. For example, we might define an arrow for programming with references, by declaring `ArrowRef a b` to be the type of arrows from `a` to `b`, implementing the basic combinators, and then providing arrows

```
createRefA :: ArrowRef a (Ref a)
readRefA   :: ArrowRef (Ref a) a
writeRefA  :: ArrowRef (Ref a,a) ()
```

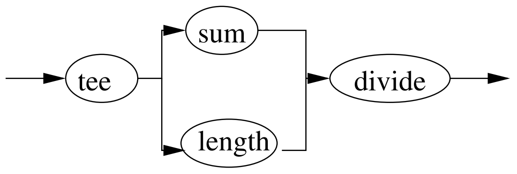

Fig. 2. Average of a list.

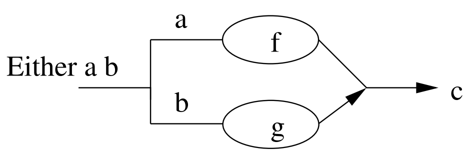

Fig. 3. Choice between `f` and `g`.

to perform the basic operations on references. With these definitions, we can write side-effecting programs in a dataflow style. For example, an arrow to increment the contents of a reference could be programmed as

```
incrRefA :: ArrowRef (Ref Int) ()
incrRefA =
  (pure id &&& (readRefA >>> pure (+1)))
  >>> writeRefA
  where f &&& g = pure tee >>> (f***g)
```

Li and Zdancewic found another use for arrows: they realised that, since all the data- and control-flow in an arrow program is expressed using the arrow combinators, then they could define a type of *flow arrows*, whose primitive arrow combinators implement the type checking of an information flow type system. Their type system assigns a *security label* drawn from a suitable lattice, such as

```
data Label = LOW | MEDIUM | HIGH
  deriving (Eq, Ord)
```

to the input and output of each arrow (where the `deriving` clause declares that `LOW`≤`MEDIUM`≤`HIGH`). Their arrows themselves are represented by the Haskell type `FlowArrow l arr a b`, which is actually an *arrow transformer*: the type `l` is the security lattice, `a` and `b` are the input and output types, and `arr` is an *underlying arrow* type such as `ArrowRef`. Flow arrows *contain* arrows of type `arr a b`, together with flow information about their inputs and outputs.
In the information flow type system, an arrow is assigned a flow type $\ell_1 \rightarrow \ell_2$ under a set of constraints, where ℓ_1 and ℓ_2 are security labels. The rules for `pure` and `(>>>)` are given in Figure 4. The `FlowArrow` type represents not only the underlying computation, but also the information flow typing—it is represented as a record

```
data FlowArrow l arr a b = FA
  { computation :: arr a b,
    flow        :: Flow l,
```

$$\frac{}{\vdash \texttt{pure}\, f : \ell \rightarrow \ell} \qquad \frac{C_1 \vdash f : \ell_1 \rightarrow \ell_2 \quad C_2 \vdash g : \ell_3 \rightarrow \ell_4}{C_1, C_2, \ell_2 \sqsubseteq \ell_3 \vdash f{>}{>}{>}g : \ell_1 \rightarrow \ell_4}$$

Fig. 4. Typing rules for `pure` and `>>>`.

$$s^{\mathbb{L}} ::= \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\mathbf{either}\; s^{\mathbb{L}}\; s^{\mathbb{L}}\;)^{\ell}$$

Fig. 5. Extended security types

```
    constraints :: [Constraint l]
  }
data Flow l = Trans l l | Flat
data Constraint l = LEQ l l
```

Here the `flow` component represents either $\ell_1 \to \ell_2$ (`Trans l1 l2`), or the "polymorphic" $\ell \to \ell$ for any ℓ (`Flat`), which is needed to give an accurate typing for `pure`. The `constraints` field just collects the constraints on the left of the turnstile. With this representation, it is easy to implement the typing rules in the arrow combinators. Security labels are introduced and checked by the arrow `tag l`, with flow `Trans l l`, which forces both its input and output to have the given security label.
Note that the information flow types are quite independent of the Haskell types! Moreover, they are not checked during Haskell type-checking. Rather, when a flow arrow is constructed during program execution, all the necessary constraints are collected dynamically—but they are checked before the underlying computation is run. Li and Zdancewic's library exports `FlowArrow` as an abstract type, and the only way to extract the underlying computation is via a certification function which solves the constraints first. If any constraint is not satisfied, then the underlying code is rejected.
Li and Zdancewic also considered declassification, which requires adding the user's security level as a context to the typing rules, and a new form of constraint—but we ignore the details here.

3 Refining security types

Li and Zdancewic's library uses single security labels as security types. As a consequence, values are classified secrets when they contain, partially or totally, some confidential information. For instance, if one component of a pair is secret, the whole pair becomes confidential. This design decision might be a potential restriction to build some applications in practice. With this in mind, we extend Li and Zdancewic's work to include security types with more than one security label. The presence of several security labels in security types allows to develop a more precise, and consequently permissive, analysis of the information flow inside of a program.

3.1 Security types

We assume a given security lattice $\mathbb{L}$ where security levels, denoted by ℓ, are ordered by a partial order $\leq$. Top and bottom elements are written $\top$ and $\bot$, respectively. Security types are given in Figure 5 and their subtyping relationship in Figure 6. Security type $(s^{\mathbb{L}}, s^{\mathbb{L}})$ provides security annotations for pair types. Security type $(\mathbf{either}\; s^{\mathbb{L}}\; s^{\mathbb{L}}\;)^{\ell}$ provides annotations for type `Either`. Security type ℓ decorates any other Haskell

$$\frac{\ell_1 \leq \ell_2}{\ell_1 \sqsubseteq \ell_2} \qquad \frac{s_1^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}} \quad s_2^{\mathbb{L}} \sqsubseteq s_4^{\mathbb{L}}}{(s_1^{\mathbb{L}}, s_2^{\mathbb{L}}) \sqsubseteq (s_3^{\mathbb{L}}, s_4^{\mathbb{L}})}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \quad s_1^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}} \quad s_2^{\mathbb{L}} \sqsubseteq s_4^{\mathbb{L}}}{(\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1} \sqsubseteq (\mathbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_2}}$$

Fig. 6. Subtyping relationship

type (e.g. `Int`, `Float`, `[a]`, etc.). Security types are represented in our library as follows:

```
data SecType l
         = SecLabel l
         | SecPair (SecType l) (SecType l)
         | SecEither (SecType l) (SecType l) l
```

where `l` implements a lattice of security levels.

3.2 Defining FlowArrowRef

The abstract data type `FlowArrowRef` defines our embedded language by implementing an *arrow* interface:

```
data FlowArrowRef l a b c = FARef
     { computation  :: a b c
     , flow         :: Flow (SecType l)
     , constraints  :: [Constraint (SecType l)] }
```

This definition is similar to the definition of `FlowArrow` except for using the type `(SecType l)` as type argument for `Flow` and `Constraint`. Constructor `Flat` needs to be removed from data type `Flow` as a consequence of dealing with security types with more than one security label. In `FlowArrow`, `Flat` is used to establish that pure computations have the same input and output security type. Unfortunately, `Flat` cannot be used in `FlowArrowRef`, otherwise secrets might be leaked. For instance, consider the program `pure ( (x,y) -> (y,x) )` that just flips components in a pair. Assume that `x`, annotated with security label `HIGH`, is a secret input and `y`, annotated with security label `LOW`, contains public information. If `(HIGH,LOW)` is the input and output security types for that program, the value of `x` will be immediately revealed!
Similarly to Li and Zdancewic's work, `FlowArrowRef` encodes a typing judgement to verify information-flow policies. Naturally, our encoding is more complex than that in `FlowArrow`. This complexity essentially arises from considering richer security types. The typing judgment has the form: $C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}$, where f is a purely-functional computation, C is a set of constrains that, when satisfied, guarantees information-flow policies, and $\tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}$ is a *flow type*, which denotes that f receives input values of type τ_1 with security type $s_1^{\mathbb{L}}$, and produces output values of type τ_2 with security type $s_2^{\mathbb{L}}$. Except for combinator `pure`, most of the typing rules

$$\frac{f \;::\; \tau_1 \;\rightarrow\; \tau_2}{\emptyset \vdash \texttt{pure}\, f \;:\; \tau_1 \mid s_1^{\mathbb{L}} \;\rightarrow\; \tau_2 \mid \textbf{only}\; join(s_l^{\mathbb{L}})}$$

Fig. 7. Typing rule for combinator `pure`

in Li and Zdancewic's work can be easily rewritten using this typing judgment, and therefore, we omit them here.

3.3 Security types and combinator `pure`

Different from Li and Zdancewick's work, it is not straightforward to determine security types for computations built with arrow combinators. Basically, the difficulty comes from deciding the output security type for combinator `pure`. This combinator can take any arbitrary Haskell function as its argument. Then, the structure of its output, and consequently its output security type, can be different in every application. For instance, output security types for `pure` computations that return numbers and pair of numbers consist of security labels and pair of security labels, respectively. Moreover, although the structure of the output security type could be determined, it is also difficult to establish the security labels appearing in it. To illustrate this point, consider the computation `pure ( \(x,y) -> (x+y, y) )`, where inputs x and y have security labels `LOW` and `HIGH`, respectively. It is clear that the output security type for this example is (`HIGH`, `HIGH`). However, in order to determine that, it is necessary to know how the input is used to build the output. This input-output dependency might be difficult to track when more complex functions are considered. With this in mind, we introduce a new security type to $s^{\mathbb{L}}$:

$$s^{\mathbb{L}} \;::=\; \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\textbf{either}\; s^{\mathbb{L}}\; s^{\mathbb{L}}\;)^{\ell} \mid \textbf{only}\; \ell$$

Security type **only** ℓ represents any security type that contains all their security labels as ℓ. Typing rule for `pure` is given in Figure 7. Observe the use of the Haskell typing judgment (written ::) in the hypothesis of the rule. Function $join(s_1^{\mathbb{L}})$ computes the join of all the security labels in $s_l^{\mathbb{L}}$. Essentially, the typing rule over-approximates the output security type by using the security labels found in the input security type. By only having one piece of secret information as input, results of `pure` computations are thus confidential regardless what they do or what kind of result they return. As a consequence, computations that follow combinator `pure` cannot operate on public data any more. As an example, consider the program `f >>> pure ( \(x,y) -> y + 1) >>> g`, where computation g operates on public data and computation `f` produces a pair where the first and second components are secret and public values, respectively. This simple program just adds one to the public output of `f` and provides that as the input of `g`. However, the program is rejected by the encoded type system in our library, even though no leaks are produced by this code. The reason for this is that program `g` receives confidential information from `pure` while it expects only public inputs. Since `pure` is responsible for allowing the use of any Haskell functions in the library, this restriction seems to be quite severe to implement concrete applications.

3.4 Combinator `lowerA`

Combinator `lowerA` is introduced to mitigate the restriction of not allowing computations on public data to take some input produced by `pure` combinators. Basically, `lowerA` takes a security label ℓ and an arrow computation p, and returns a computation p'. Computation p' behaves like p and has the same input type, output type, and input security type as p. However, its output security type might be different. The output security type is constructed based on the output type of p and it contains only security labels of value ℓ. In other words, `lowerA` downgrades the output of p to the security level ℓ. In principle, this combinator might be also used to leak secrets. An attacker can just apply `(lowerA LOW)` to every computation that involves secrets! To avoid this kind of attacks, `lowerA` filters out data with security level higher than ℓ.

Input filtering mechanism Filtration of data is done by replacing some pieces of information with `undefined` [1]. This idea is implemented by the member function `removeData` of the type-class `FilterData`. The signature of the type-class is the following:

```
class (Lattice l) => FilterData l t where
  removeData :: l -> t -> (SecType l) -> t
```

Method `removeData` receives a security level `l`, a value of type `t`, and a security type `(SecType l)`, and produces another value of type `t` where the information with security label higher than `l` is replaced by `undefined`. As an example, instantiations for integers and pairs are given in Figure 8. Observe how the use of type-classes allows to define different filtering policies for different kind of data. This is particularly useful when references are introduced in the language (see Section 4.6).
The introduction of undefined values might also introduce leaks due to termination. For instance, if filtered values are used inside of computations that branches on secrets, then the program might terminate (or not) depending on which branch is executed. However, these kind of leaks only reveal one bit of information about confidential data. In some scenarios, leaking one bit due to termination is acceptable and *termination-insensitive* security conditions are adopted for those cases. In fact, our library is particularly suitable to guarantee *termination-insensitive* security specifications.

Building output security types Besides introducing a filtering mechanism, `lowerA` constructs output security types where security labels are all the same. We define the following type-class:

```
class (Lattice l) => BuildSecType l t where
  buildSecType :: l -> t -> (SecType l)
```

Method `buildSecType` receives a security label `l` and a value of type `t`, and produces a security type for `t` where security labels are `l`. For instance, it produces security type `(l,l)` for pair of integers. Instantiations for pairs and integers are given in Figure 9.

[1] This is an undefined value in Haskell and it is member of every type.

```
instance (Lattice l)
         => FilterData l Int where
            removeData l x (SecLabel l') =
             if label_leq l' l then x
             else undefined
instance (Lattice l, FilterData l a,
          FilterData l b)
          => FilterData l (a,b) where
          removeData l (x, y) (SecPair lx ly) =
            (removeData l x lx, removeData l y ly)
```

Fig. 8. Instantiations for `FilterData`

```
instance (Lattice l) =>
         BuildSecType l Int where
         buildSecType l _ = (SecLabel l)

instance
 (Lattice l, BuildSecType l a, BuildSecType l b)
 => BuildSecType l (a,b) where
    buildSecType l _ =
        (SecPair (buildSecType l (undefined::a))
                 (buildSecType l (undefined::b)))
```

Fig. 9. Instantiations for `BuildSecType`

Observe that the value of the second argument of `buildSecType` is not needed, but its type. Type-classes provide a mechanism to access information about types in Haskell and take different actions, like building different security types, depending on them.
When `lowerA` receives a computation as an argument, it needs to know its output type in order to properly apply `buildSecType`. For that purpose, we introduce another type-class:

```
class (Lattice l, Arrow a)
  => TakeOutputType l a b c where
     deriveSecType :: l -> (a b c) -> (SecType l)
```

Method `deriveSecType` receives a security label `l`, an arrow computation `(a b c)`, and returns the corresponding security type `(SecType l)` for the output type `c`. The instantiation of this type-class is shown in Figure 10.
To put it briefly, combinator `lowerA` creates a new computation that behaves as the computation received as argument, but calling the described methods `removeData` and `buildSecType` in due course. The type signature for `lowerA` is given in Figure 11. Typing rule for `lowerA` is shown in Figure 12. Observe how the output security type is changed. Function ρ is defined in Figure 13 and implemented by the method `buildSecType`. As a simple example of the use of `lowerA`, we rewrite the example in Section 3.3 as follows: `f >>> lowerA LOW (pure (\(x,y) -> y + 1))`

```
instance
 (Lattice l, BuildSecType l c, Arrow a)
 => TakeOutputType l a b c where
 deriveSecType l ar =
    buildSecType l (undefined::c)
```

Fig. 10. Instantiation for `TakeOutputType`

```
lowerA :: ( Lattice l, Arrow a,
            FilterData l b, BuildSecType l c,
            TakeOutputType l (FlowArrowRef l a) b c )
          => l -> FlowArrowRef l a b c -> FlowArrowRef l a b c
```

Fig. 11. Type signature for `lowerA`

`>>> g`. Observe that the value received by program `g` is not confidential anymore, and consequently, the program passes the type-checking tests in our library. In this example, the filtering mechanism of `lowerA` does not introduce leaks due to termination. In general, the possibilities to exploit undefined values introduced by some computation like `lowerA LOW p` are related to the security of `p`. If `p` only produces `LOW` values, no leaks due to termination are introduced. Otherwise, if `p` presents, for instance, some flows from secret data to its output, a one-bit leak due to termination might happen as a price to pay for not being able to predict the input-output dependency of `p` and avoiding leaking the whole secret.

One alternative implementation to the input filter mechanism in `lowerA` ℓ p could have been to reject computation p if it takes some input with security label higher than ℓ. Unfortunately, this idea might not work properly when programs take input from external modules or components, which frequently provide data with different security levels to arrow computations. Consequently, the pattern `lowerA` ℓ (`pure` f) is particularly useful to get any values at security levels below ℓ regardless the security input type of `pure` f.

4 Adding references

Dealing with information-flow security in languages with reference manipulation is not a novelty. Unsurprisingly, Jif and FlowCaml include them as a language feature. Nevertheless, it is stated as an open question how Li and Zdancewic's library needs to be modified to consider side-effects. In particular, what arrows could be used to handle them and how their encoded type system needs to be modified. We have already started answering these question with the modification of `pure` and the introduction of `lowerA` in Section 3. We will complete answering Li and Zdancewic's questions by showing how to extend their library to introduce references. The developed techniques in this section can be considered for other kind of side-effects as well.

$$\frac{C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \to \ \tau_2 \mid s_2^{\mathbb{L}}}{C \vdash \mathtt{lowerA}\, \ell\, f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \to \ \tau_2 \mid \rho(\ell, \tau_2)}$$

Fig. 12. Typing rule for `lowerA`

$$\frac{}{\rho(int, \ell) \ \to \ \ell}$$

$$\frac{\rho(\tau, \ell) \ \to \ s_1^{\mathbb{L}}}{\rho(\tau\ ref, \ell) \ \to \ s_1^{\mathbb{L}}\, \mathbf{ref}^{\ell}}$$

$$\frac{\rho(\tau_1, \ell) \ \to \ s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \ \to \ s_2^{\mathbb{L}}}{\rho((\tau_1, \tau_2), \ell) \ \to \ (s_1^{\mathbb{L}}, s_2^{\mathbb{L}})}$$

$$\frac{\rho(\tau_1, \ell) \ \to \ s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \ \to \ s_2^{\mathbb{L}}}{\rho(either\ \tau_1\ \tau_2, \ell) \ \to \ (\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell}}$$

Fig. 13. Definition for Function ρ

4.1 Security types for references

The treatment of references is based on Pottier and Simonet's work [PS02]. They introduce security types for references containing two parts: a security type and a security label. The security type provides information about the data that is referred to, while the security label gives a security level to the reference itself as a value. Following the same approach, we extend our security types as follows:

$$s^{\mathbb{L}} \ ::= \ \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\mathbf{either}\ s^{\mathbb{L}}\ s^{\mathbb{L}}\ )^{\ell} \mid \mathbf{only}\ \ell \mid s^{\mathbb{L}}\ \mathbf{ref}^{\ell}$$

Observe that security types for references ($s^{\mathbb{L}}\ \mathbf{ref}^{\ell}$) are composed of two parts as mentioned before. The subtyping relationship is also extended as follows:

$$\frac{s_1^{\mathbb{L}} = s_2^{\mathbb{L}} \quad \ell_1 \sqsubseteq \ell_2}{s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell_1} \sqsubseteq s_2^{\mathbb{L}}\ \mathbf{ref}^{\ell_2}} \qquad (1)$$

In order to avoid aliasing problems[NNH99], this rule imposes an invariant in the subtyping relationship by requiring $s_1^{\mathbb{L}}$ to be the same as $s_2^{\mathbb{L}}$. Clearly, this invariant needs to be preserved by the arrow combinators in the library. However, `lowerA` could break that invariant! Remember that it changes every security label in the output security type of a given computation. As a consequence, we need to modify its implementation (see Section 4.2).

Data type `SecType` is extended as follows:

```
data SecType l
         = SecLabel l
         | SecPair (SecType l) (SecType l)
         | SecEither (SecType l) (SecType l) l
         | SecRef (SecType l) l
```

where `SecRef (SecType l) l` represents security types for references.

4.2 References and combinator `lowerA`

Combinator `lowerA` could break the subtyping invariant for references described in (1). As a result, aliasing problems, and therefore leakage of secrets, might be introduced. The root of this problem comes from the fact that `lowerA` only uses output types to determine output security types. To illustrate this problem, consider a program that has two public references, `r1` and `r2`, with security type `(SecRef (SecLabel LOW) LOW)`. Assume that both references refer to the same value. If `r1`, for instance, is fed into the computation `lowerA HIGH (pure id)`, the output produced, which is obviously `r1`, will have security type `(SecRef (SecLabel HIGH) HIGH)`. Observe that the security type for the content of the reference has changed. After doing that, leaks can occur by writing secrets using `r1` and reading them out by using `r2`. Naturally, `lowerA` could also examine input security types, but unfortunately this is not enough. Once again, the difficulty to track input-output dependencies of `pure` computations (see Section 3.3) makes it difficult to determine, for instance, which reference from the input correspond to which reference in the output. Consequently, it is also difficult to determine security types for references in the output based on the input security types. To overcome this problem, we use a mechanism that can transport security information about contents of references from the input to the output of an arrow computation. In this way, `lowerA` can read this information and place the corresponding security types references when needed, and thus keep the subtyping invariant. This mechanism relies on the use of singleton types, which are the topic of the next section.

4.3 Preserving subtyping invariants

On one side, combinator `lowerA` builds output security types based on the output type of computations. On the other hand, security types for the content of references must never be changed. So, why not encoding in the Haskell type system the security type for the content of references? Hence, `lowerA` can take the encoded information and precisely determines the corresponding security type for the content of each reference. Singleton types [Pie04] are adequate to represent specific values at the level of types. Essentially, they allow to have a match between values and types and vice versa. Our goal is, therefore, to encode values of type (`SecType l`) in more fine-grained Haskell types. For instance, the encoding for values of type `(SecType Label)` can be done as follows:

```
data SLow    = VLow
data SMedium = VMedium
data SHigh   = VHigh

data SSecLabel  lb         = VSecLabel  lb
data SSecPair   st1 st2    = VSecPair   st1 st2
data SSecEither st1 st2 lb= VSecEither st1 st2 lb
data SSecRef    st  lb     = VSecRef    st  lb
```

Observe how one type has been introduced for each constructor appearing in `Label` and `SecType`. With this encoding, we can now represent security types in the Haskell type system. As an example, security type `(SecRef (SecLabel HIGH) LOW)` can be encoded using the value `(VSecRef (VSecLabel VHigh) VLow)` of type `(SSecRef (SSecLabel SHigh) SLow)`.

As mentioned before, `lowerA` should use the encoded information to place the corresponding security types for content of references. In order to achieve that, we need a mapping from singleton types to values of type `(SecType l)`. The following code implements that:

```
class STLabel lb l where
  toLabel :: lb -> l

instance STLabel SLow Label where
  toLabel _ = LOW
instance STLabel SMedium Label where
  toLabel _ = MEDIUM
instance STLabel SHigh Label where
  toLabel _ = HIGH

class STSecType st l where
  toSecType :: st -> SecType l

instance STLabel lb l
   => STSecType (SSecLabel lb) l where
  toSecType _
    = SecLabel (toLabel (undefined::lb))
instance (STSecType st l, STLabel lb l)
  => STSecType (SSecRef st lb) l where
  toSecType _
    = SecRef (toSecType (undefined::st))
             (toLabel (undefined::lb))
instance (STSecType st1 l, STSecType st2 l)
  => STSecType (SSecPair st1 st2) l where
  toSecType _
    = SecPair (toSecType (undefined::st1))
              (toSecType (undefined::st2))
instance (STSecType st1 l,
          STSecType st2 l, STLabel lb l)
  => STSecType (SSecEither st1 st2 lb) l where
  toSecType _
    = SecEither (toSecType (undefined::st1))
                (toSecType (undefined::st2))
                (toLabel (undefined::lb))
```

Functions `toLabel` and `toSecType` return security labels and security types based on singleton types, respectively.

Having our encoding ready, we introduce references as values of the data type: `data Ref st a = Ref st (IORef a)`, where `(IORef a)` is the type for references in Haskell and `st` is a singleton type encoding the security type for its content. At this point, we are in conditions to extend the function `buildSecType`, used by `lowerA`, to build output security types:

```
instance (Lattice l, STSecType st l)
   => BuildSecType SecType l (SRef st a) where
  buildSecType l _
    = (SecRef (toSecType (undefined::st)) l)
```

Observe how `buildSecType` calls `toSecType` to build the security type for the content of the reference by passing an undefined value of singleton type `st`. The subtyping invariant is now preserved by `lowerA`. In fact, this technique can be used to preserve any subtyping invariant required in the library.

4.4 Reference manipulation

Li and Zdancewic's library uses the *underlying arrow* `(->)` to perform computations. However, we need to modify that in order to include side-effects produced by references. The following data type defines the *underlying arrow* used in our library: `data ArrowRef a b = a -> IO b`. *Underlying computations* can therefore take an argument of type `a` and return a value of type `(IO b)`, which probably produces some side-effects related to references.
Three primitives are provided to create, read, and write references: `createRefA`, `readRefA`, and `writeRefA`. Basically, these functions lift the traditional Haskell operations to manipulate references into `FlowArrowRef`, but performing some checking related to information-flow security (see Section 4.5). However, from a programmer's point of view, they look similar to any primitives that deal with references. For instance, `createRefA` has the following signature:

```
createRefA :: (Lattice l, STSecType st l, BuildSecType l a)
              => st -> l -> FlowArrowRef l ArrowRef a (Ref st a)
```

where singleton type `st` encodes the security type for the content of the reference, and `l` is the security level of the reference as a value. Observe that `ArrowRef` is used for the underlying computation. As an example, `(createRefA (VSecLabel VHigh) LOW)` returns a computation that creates a public reference to a secret value received as argument. This is the only primitive where programmers must use singleton types and where the library exploits the correspondence between values and types. Because of that, it could be possible to remove the argument `st` from `createRefA` to make its type signature simpler. However, by doing that, programmers would need to explicitly specify the type for every occurrences of `createRefA` with their corresponding `(STSecType st l)` and `(Ref st a)`.

4.5 Typing rules for reference primitives

Pottier and Simonet present a type-based information flow analysis for CoreML provided with references, exceptions and let-polymorphism [PS02]. Particularly, their type system is constraint-based and uses effects to deal with references. We restate some

$$\frac{}{e(\ell) \rightarrow \ell} \qquad \frac{e(s_1^{\mathbb{L}}) \rightarrow \ell_1 \quad e(s_2^{\mathbb{L}}) \rightarrow \ell_2}{e((s_1^{\mathbb{L}}, s_2^{\mathbb{L}})) \rightarrow \ell_1 \sqcup \ell_2}$$

$$\frac{}{e((\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell}) \rightarrow \ell} \qquad \frac{}{e(\mathbf{only}\ \ell) \rightarrow \ell}$$

$$\frac{}{e(s^{\mathbb{L}}\ \mathbf{ref}^{\ell}) \rightarrow \ell}$$

Fig. 14. Definition for Function e

$$(PURE)\frac{f\ :\ \tau_1 \rightarrow \tau_2}{\top, \emptyset \vdash \mathtt{pure}\ f\ :\ \tau_1 \mid s_1^{\mathbb{L}} \rightarrow \tau_2 \mid \mathbf{only}\ \ell}$$

$$(SEQ)\frac{pc_1, C_1, \vdash f_1\ :\ \tau_1 \mid s_1^{\mathbb{L}} \rightarrow \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2 \vdash f_2\ :\ \tau_2 \mid s_3^{\mathbb{L}} \rightarrow \tau_4 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup \{s_2^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}}\} \vdash f_1 \ggg f_2\ :\ \tau_1 \mid s_1^{\mathbb{L}} \rightarrow \tau_4 \mid s_4^{\mathbb{L}}}$$

$$(CHOICE)\frac{pc_1, C_1 \vdash f_1\ :\ \tau_1 \mid s_1^{\mathbb{L}} \rightarrow \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2, \vdash f_2\ :\ \tau_3 \mid s_3^{\mathbb{L}} \rightarrow \tau_2 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup C_3 \vdash f_1 \mid\mid\mid f_2\ :\ flow}$$

$$flow\ =\ either\ \tau_1\ \tau_3 \mid (\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \rightarrow \tau_2 \mid \uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell)$$

$$C_3 = \{(\mathbf{either} s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft (pc_1 \sqcap pc_2), (\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft e(\uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell))\}$$

Fig. 15. Typing rules for pure, sequential composition, and choice combinators

of their ideas in the framework of our library. More precisely, we adapt our encoded type-checker to include effects and consequently involve references.

We enhance the typing judgement introduced in Section 3.2 as follows: $pc\,, C \vdash f\ :\ \tau_1 \mid s_1^{\mathbb{L}} \rightarrow \tau_2 \mid s_2^{\mathbb{L}}$, where the new parameter, *pc*, is a lower bound on the security level of the memory cell that is written. In Figure 15, we show how typing rules for pure, sequential, and branching computations are rewritten using this new parameter. Typing rules for other combinators are adapted similarly. Rule `(PURE)` produces no side-effects and therefore it imposes no lower bounds in *pc*. Rule `(SEQ)` takes the meet of the lower bounds for side-effects as the new *pc*. Rule `(CHOICE)` essentially requires that the branching computation does not produce side-effects or results that are below the guard of the branch, which has type $either\ \tau_1\ \tau_3$. These requirements are enforced by $(\mathbf{either} s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft (pc_1 \sqcap pc_2)$ and $(\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft e(\uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell))$, respectively. As defined in Simonet and Pottier's work, constraint $s^{\mathbb{L}} \blacktriangleleft \ell$ imposes ℓ as an upper bound for every security label in $s^{\mathbb{L}}$. Function e determines the security level of a given value (see Figure 14). Operator $\uparrow$ lifts security labels that are below certain security level, but not violating subtyping invariants (see Figure 16).

Typing rules for references are introduced in Figure 17. Singleton type $\mathbf{s}^{\mathbb{L}}$ encodes the security type $s^{\mathbb{L}}$ and is generated by the value $(\mathbf{s}^{\mathbb{L}})_v$. Rule `(CREATE)` requires that the singleton type passed as argument matches the input security type. Otherwise,

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow\ \ell_1\, \ell_2\ \rightarrow\ \ell_2} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow\ \ell_1\, \ell_2\ \rightarrow\ \ell_1}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow\ (s^{\mathbb{L}}\ \mathbf{ref}^{\ell_1})\ \ell_2\ \rightarrow\ s^{\mathbb{L}}\ \mathbf{ref}^{\ell_2}} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow\ (s^{\mathbb{L}}\ \mathbf{ref}^{\ell_1})\ \ell_2\ \rightarrow\ s^{\mathbb{L}}\ \mathbf{ref}^{\ell_1}}$$

$$\frac{\uparrow\ s_1^{\mathbb{L}}\ \ell\ \rightarrow\ s_3^{\mathbb{L}} \quad \uparrow\ s_2^{\mathbb{L}}\ \ell\ \rightarrow\ s_4^{\mathbb{L}}}{\uparrow\ (s_1^{\mathbb{L}}, s_2^{\mathbb{L}})\ \ell\ \rightarrow\ (s_3^{\mathbb{L}}, s_4^{\mathbb{L}})} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad \uparrow\ s_1^{\mathbb{L}}\ \ell_2\ \rightarrow\ s_3^{\mathbb{L}} \quad \uparrow\ s_2^{\mathbb{L}}\ \ell_2\ \rightarrow\ s_4^{\mathbb{L}}}{\uparrow\ (\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_2}}$$

$$\frac{\ell_2 \sqsubset \ell_1 \quad \uparrow\ s_1^{\mathbb{L}}\ \ell_2\ \rightarrow\ s_3^{\mathbb{L}} \quad \uparrow\ s_2^{\mathbb{L}}\ \ell_2\ \rightarrow\ s_4^{\mathbb{L}}}{\uparrow\ (\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell_1}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s_3^{\mathbb{L}}\ s_4^{\mathbb{L}})^{\ell_1}}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow\ (\mathbf{only}\ \ell_1)\ \ell_2\ \rightarrow\ \mathbf{only}\ \ell_2} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow\ (\mathbf{only}\ \ell_1)\ \ell_2\ \rightarrow\ \mathbf{only}\ \ell_1}$$

Fig. 16. Definition for Function $\uparrow$

$$(CREATE)\frac{}{e(s_1^{\mathbb{L}}), \emptyset \vdash \texttt{createRefA}\ (\mathbf{s_1^{\mathbb{L}}})_v\ \ell\ :\ \tau \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau\ ref \mid s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell}}$$

$$(READ)\frac{}{\top, \emptyset \vdash \texttt{readRefA} : \tau\ ref \mid s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell}\ \rightarrow\ \tau \mid \uparrow (s_1^{\mathbb{L}}, \ell)}$$

$$(WRITE)\frac{}{e(s^{\mathbb{L}}), \{\ell \lhd s^{\mathbb{L}}\} \vdash \texttt{writeRefA} : (\tau\ ref, \tau) \mid (s^{\mathbb{L}}\ \mathbf{ref}^{\ell}, s^{\mathbb{L}})\ \rightarrow\ () \mid \bot}$$

Fig. 17. Typing rules for reference primitives

programmers could introduce inconsistencies in the type-checking process. The side-effect produced by creation of references is allocation of memory. Therefore, the pc is related with the security level of the content of the created reference ($e(s_1^{\mathbb{L}})$). Rule `(READ)` lifts security labels in the output security type considering the security level of the reference ($\uparrow\ (s_1^{\mathbb{L}}, \ell)$). Rule `(WRITE)` imposes the constraint $\ell \lhd s^{\mathbb{L}}$. Similarly to Simonet and Pottier's work, constraint $\ell \lhd s^{\mathbb{L}}$ requires $s^{\mathbb{L}}$ to have security level ℓ or greater, and is used to record a potential information flow.

We modify the implementation of the type-system in our library to include effects. Consequently, data type `FlowArrowRef` is extended with a new field called `pc` to represent lower bounds for side-effects as explained above. Data type `Constraint` is also extended to involve operators $\lhd$ and $\blacktriangleleft$. Moreover, we add unification mechanisms inside of arrow combinators to pass information about security types when needed. As a consequence, a few security annotations need to be provided by programmers. Li and Zdancewic's library does not need this feature since their security types are very simple. One of the interesting aspect of implementing unification inside of arrows is the generation of fresh names. Our library generates fresh names by applying renaming functions when arrow combinators are applied, but we omit the details here due to lack of space.

4.6 Filtering references

References introduce the possibility of having shared resources in programs. In Section 3.4, the filtering mechanism replaces some pieces of information with `undefined`. Nevertheless, it is not recommended to replace the content of some reference with `undefined` since it might be used by other parts (or threads) in the program. We still need to restrict the access to that content somehow. In order to do that, we introduce projection functions for each reference handled by the library. Projection functions are basically functions that return values less informative than their arguments. The concept of projection functions has been indirectly used in semantic models for information-flow security [Hun91, SS99]. The instance for references of the method `removeData` creates projection functions that, when applied to the contents of their associated references, return values where some information higher than some security level is replaced by `undefined`. However, the content of the reference itself is not modified. Observe that the filtering principle applied by projection functions and `removeData` is the same. Combinator `readRefA` is also modified to return the content of the reference by firstly passing it through its corresponding projection function. Due to lack of space, we omit the implementation of these ideas here.

5 Information-flow in a concurrent setting

Concurrency introduces new covert channels, or unintended ways, to leak secret information to an attacker. As a consequence, the traditional techniques to enforce information flow policies in sequential programs are not sufficient for multithreaded languages [SV98]. One particularly dangerous covert channel is called *internal timing*. It allows to leak information when secrets affect the timing behavior of a thread, which via the scheduler, affects the order in which public computations occur. Consider the following two imperative programs running in two different threads:

$$\begin{array}{l} t_1 : (\texttt{if h} > 0\ \texttt{then skip}(120)\ \texttt{else skip}(1));\ \texttt{l} := 1 \\ t_2 : \texttt{skip}(60);\ \texttt{l} := 0 \end{array} \qquad (2)$$

Variables h and l store secret and public information, respectively. Assume $\texttt{skip}(n)$ executes n consecutive $skip$ commands. Notice that both t_1 and t_2 are secure in isolation under the notion of *noninterference* [SM03]. However, by running them in parallel, it is possible to leak information about h. To illustrate that, we assume an scheduler with time slice of 80 steps that always starts by running t_1. On one hand, if $\texttt{h} > 0$, t_1 will run for 80 steps, and while being running $\texttt{skip}(120)$, t_2 is scheduled and run until completion. Then, the control is given again to t_1, which completes its execution. The final value of `l` is 1. On the other hand, if $\texttt{h} \leq 0$, t_1 finishes first its execution. After that, t_2 is scheduled and run until completion. In this case, the final value of `l` is 0. An attacker can, therefore, deduce if $\texttt{h} > 0$ (or not) by observing the final value of `l`. Different from the *external timing* covert channel, the attacker does not need to observe the actual execution time of a program in order to deduce some secret information. Moreover, internal timing leaks can also be magnified via loops, where each iteration of the loop can leak one bit of the secret. Hence, entire secret values can be leaked.

There are several existing approaches to tackling internal timing flows. Several works by Volpano and Smith [SV98, VS99, Smi01, Smi03] propose a special primitive called `protect`. By definition, `protect`(c) takes one atomic step in the semantics with the effect of executing c to the end. Internal timing leaks are removed if every computation that branches on secrets is wrapped by `protect`$()$ commands. However, implementing `protect` imposes a major challenge [SS00, Sab01, RS06a] (except for cooperative schedulers [RS06b]). These proposals rely on the modification of the run-time environment or the assumption of randomized schedulers, which are rarely found in practice. Russo et al. [RHNS06] propose a transformation to close internal timing channels that does not require the modification of the run-time environment. The transformation works for programs that run under a wide class of round-robin schedulers and only rejects those ones that have symptoms of illegal flows inherent from sequential settings. Boudol and Castellani [BC01, BC02] propose type systems for languages that do not rely on the `protect` primitive. However, they reject programs with assignments to low variables after some computation that branches on secrets. Internal timing problem can also be solved by considering external timing. Definitions related to external timing involve stronger attackers. As expected, an stronger attacker model imposes more restriction on programs. For instance, loops branching on secrets are disallowed. There are several works on that direction [Aga00, SS00, Sab01, SM02, KM06]. Zdancewic and Myers [ZM03] prevent internal timing leaks by disallowing races on public data. However, their approach rejects innocent secure programs like $l := 0 \parallel l := 1$ where l is a public variable. Recently, Huisman et al. [HWS06] improved Zdancewic and Myers' work by using logic-based characterizations and well known model checking techniques. Several proposals have been explored in process-calculus settings [HVY00, FG01, Rya01, HY02, Pot02], but without considering the impact of scheduling.

The referred works above have neglected to consider implementing case studies where the proposed enforcement mechanisms are applied. This work presents, to the best of our knowledge, the first concrete implementation of a case study that consider information -flow policies in presence of concurrency.

6 Closing internal timing channels

We incorporate a run-time mechanism to close internal timing covert channels in our library. We base our approach in a combination of ideas taken from the literature. On one hand, Russo and Sabelfeld [RS06b] show how to implement `protect`$()$ for cooperative schedulers. Essentially, their work states that threads must not yield control inside of computations that branch on secrets. Russo et al. [RHNS06], on the other hand, express that a class of round-robin schedulers does not suffer from leaks due to dynamic thread creation. As a consequence, creation of threads can be allowed at any point in programs. By mixing these two ideas, we modify the *underlying* arrow combinators in order to implement a cooperative round-robin scheduler and to guarantee

that computations branching on secrets do not yield control when running. In this way, internal timing leaks are removed from programs and a flexible treatment for dynamic thread creation is also obtained. In fact, the introduced modifications are completely independent to the encoded type system described in Section 3 and 4.
Cooperative schedulers are based on yielding control when programs indicate that. On the other hand, programs are written using arrow combinators, which can be seen as a kind of *building blocks*. In our library, *simple* arrow combinators yield control after finishing their execution if they are not part of computations that branch on secrets. Example of such combinators are `pure`, `createRef`, `readRef`, and `writeRef`. Computations branching on secrets do not yield control regardless how many building blocks compose them. As result, *simple* arrow combinators and computations that branch on secrets are atomic computational units involved in interleavings. The round-robin scheduler is obtained by yielding control in a particular way.
Concurrency is introduced in our implementation by importing the Haskell module `Control.Concurrent` [SPJF96, The] . This module provides dynamic thread creation and pre-emptive concurrency. Since threads can be scheduled anytime, some synchronization is needed to restrict their execution as round-robin. Software transactional memory(STM) [HHMJ05] provides easy-to-reason and simple primitives to do that. We could have chosen more standard primitives like semaphores or `MVar` [SPJF96]. However, the obtained code would have been more complicated.
We start introducing information concerning scheduling upon the *underlying* arrow `ArrowRef`:

```
data RRobin a = RRobin
   { data  :: a, iD :: ThreadId,
     queue :: TVar [ThreadId], blocks :: Int }

data ArrowRef a b
     = AR ((RRobin a) -> IO (RRobin b))
```

Data type `(RRobin a)` stores information related to scheduling in the input and output values of arrows. Field `data` stores the input data for the arrow. Field `iD` stores the thread identification number where the arrow computation is executed. Field `queue` stores a round-robin list of threads identifiers and its access is protected by a mutex (`TVar [ThreadId]`). The list is updated when creation or termination of threads occur. Field `blocks` indicates if the thread executing the arrow computation must wait for its turn to run and then, when finishing, yields the control to another thread. This field plays an essential rôle to guarantee atomic execution of computations that branch on secrets.
We introduce two new combinators in the underlying arrow: `waitForYield` and `yieldControl`. Essentially, these combinators are responsible for implementing a round-robin scheduler. Combinator `waitForYield` blocks until the content of the head of the round-robin queue (`TVar [ThreadId]`) is the same as the thread identification (`iD`) running this combinator. Combinator `yieldControl` removes the head of the round-robin queue and put it as the last element. Both combinators have no computational effects if the field `blocks` is different from zero. The implementation of these combinators is shown in Figure 18. Function `atomically` guarantees mutual

```
waitTurn :: RRobin a -> IO ()
waitTurn sch = if (blocks sch) > 0 then return ()
               else atomically (
                     do q <- readTVar (queue sch)
                        if head q /= (iD sch)
                        then retry
                        else return ())

waitForYield :: ArrowRef a a
waitForYield = AR (\sch -> do waitTurn sch
                              return sch)

nextTurn :: RRobin a -> IO ()
nextTurn sch
  = if (blocks sch) > 0 then return ()
    else atomically (
           do q <- readTVar (queue sch)
              writeTVar (queue sch)
                        ((tail q)++[head q])
              return () )

yieldControl :: ArrowRef a a
yieldControl = AR (\sch -> do nextTurn sch
                              return sch)
```

Fig. 18. Primitives for yielding control

exclusion access to the round-robin queue. Function `retry` blocks the thread until `queue` changes its value. When this happens, it resumes its execution from the first command wrapped by `atomically`. It is important to remark that combinators in the underlying arrow are not accessible for users of the library.

Simple arrow combinators include now `waitForYield` and `yieldControl` before and after finishing their computations, respectively. Nevertheless, combinators related with branches are threaded differently. Computations that branch on secrets must not yield control until finishing their execution. Branching combinators, like (|||), can be applied to arrow computations that involve `yieldControl` in their bodies. As a consequence, when the guard of the branch involves some secrets, these combinators must no yield control to other threads. We introduce two more combinators to the *underlying arrow*: `beginAtomic` and `endAtomic`. When placed like `beginAtomic >>> f >>> endAtomic`, they leave without any effect the combinators `waitForYield` and `yieldControl` appearing in `f`. Therefore, program `f` executes until completion without yielding control to other threads. We then modify the implementation of combinators related with branchings in order to include `beginAtomic` and `endAtomic` when the condition of the branch depends on secrets. We show the Implementation details of `beginAtomic` and `endAtomic` in Figure 19. Observe that `beginAtomic` and `endAtomic` count how many computations branching on secret are nested. Com-

```
beginAtomic :: ArrowRef a a
beginAtomic
  = waitForYield >>>
    AR (\sch -> return sch {blocks =
                             ((blocks sch)+1)} )

endAtomic :: ArrowRef a a
endAtomic
  = AR (\sch -> return sch {blocks =
                             ((blocks sch)-1)})
    >>> yieldControl
```

Fig. 19. Primitives for atomicity

$$\frac{pc, C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}}{pc, C \vdash \texttt{forkRef}\ f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ () \mid \bot}$$

Fig. 20. Typing rule for `forkRef`

binators `waitForYield`, `yieldControl`, `beginAtomic` and `endAtomic` need to be pairwise to properly work.
Dynamic thread creation is introduced by the new arrow combinator `forkRef`. It takes a computation as argument and spawns it in a new thread with an exception handler. If the new thread raises an exception, the handler forces all the program to finish, reducing the bandwidth of leakings due to no termination. The typing rule for `forkRef` is shown in Figure 20. Observe that the returned value of f is discarded since f will be run in another thread.

7 Case study: online shopping

In order to evaluate the flexibility of the arrow combinators and techniques proposed in Sections 3, 4, and 6, we implemented a case study of an online shopping server. Basically, the server processes transactions related to buying products. It receives information from the network and spawn different threads to perform purchases for each client. For simplicity, we assume that there is only one product to buy and that the only information provided by clients are their names, billing addresses, and credit card numbers composed of 16 digits. We also assume that there are security levels `HIGH` and `LOW` for secret and public information, respectively. Our library guarantees, in this example, that the confidentiality of credit card numbers is preserved.
The server program consists of three components: `protectData`, `purchase`, and `showPurchase`. Component `protectData` receives information from clients and determines that credit card numbers are the only secrets in the system. The implementation of `protectData` is just a few lines that apply combinator `tag` to its input. We consider this component as part of the trusted computing based. Component

Secret: 100011100001101111001001101111110000001111111111111111

Run	Leaked credit card number	Seconds
1	101011111001111111111011101111110000001111111111111111	27
2	110011100001101111011101101111010000001111111111111111	27
3	101011100001101111101001101111110000001111111111111111	28
4	100011100101101111001001101111110000001111111111111011	28
5	100011100001101111001001101111110100001111111111111111	29

Inferred Secret: 100011100001101111001001101111110000001111111111111111

Fig. 21. Results produced by the malicious code

`purchase` simulates buying products. Moreover, it copies the client credit card number and the rest of his/her information into two different databases, respectively. We simulate the access to these databases with references to different lists of data. Component `showPurchase` retrieves information from the database with public information and shows it on the screen (a public channel).

The online shopping server can be modified to execute malicious code that exploits the internal timing covert channel. An attack similar to (2) can be implemented if no countermeasures are taken. However, such an attack only reveals one bit of the secret. In order for the attacker to obtain complete credit card numbers, it is necessary to magnify the attack by introducing a loop. Each iteration of the loop leaks one bit of the secret. The implementation of this attack reveals a credit card number in about two minutes [2]. Notoriously, it was quite straightforward to leak the sixteen digits of a credit card number even though we have no information about the run-time environment. This shows how feasible and dangerous are internal timing leaks in practice.

Our malicious code concatenates credit card numbers after the billing addresses of clients. Thus, credit card numbers can be displayed on the screen by just invoking `showPurchase`. To illustrate that, we consider a client with the credit card number `9999999999999999`. We run the attack several times obtaining different leaked credit card numbers (see Figure 21). These numbers differ in at most three bits from the binary representation of the secret. This imprecision comes from the lack of knowledge about the run-time environment, in particular, the lack of knowledge about scheduler policies. Scheduler policies are important for an internal timing attack to succeed. Nevertheless, by repeatedly running the attack and taking the most frequent boolean values in each position, it is possible to obtain the value of the secret with very high confidence. Observe that the secret and the inferred secret are the same in Figure 21.

We repeatedly run the malicious code mentioned above but with the countermeasures described in Section 6. In this opportunity, the leaked credit card number was always 0. In other words, the attack did not succeed. There is an obvious overhead introduced by restricting the scheduler in the run-time environment to behave like a round-robin

[2] Every experiment was run on a laptop Pentium M 1.5 GHz and 512 MB RAM.

one. However, this is acceptable since only small parts of a system need to manipulate secrets and therefore be written using our library.

8 Conclusions

We have presented an extension to Li and Zdancewic's library to consider secure programs with reference manipulation and concurrency. On one hand, introducing references requires to handle more richer security types than those in Li and Zdancewic's work. As consequence, a more precise analysis for information-flow security is needed. In order to obtain that, we combine several ideas from the literature in our implementation: singleton types, type-classes in Haskell, and projection functions. On the other hand, supporting concurrency requires to deal with internal-timing attacks. The extension includes a mechanism to close internal-timing covert channels and provides a flexible treatment for dynamic thread creation. Therefore, it is not necessary to modify the run-time environment to obtain secure programs. These achievements are result of taking several ideas from the literature: round-robin cooperative schedulers and software transactional memories. Similarly to Li and Zdancewic's work, the technical development in this paper is informal, although we have implemented it in Haskell. The type system encoded in `FlowArrowRef` can be mainly justified by following standard techniques to prove non-interference properties [VSI96, PS02]. A case study has been also implemented to evaluate the techniques proposed in this work. It reveals that internal-timing leaks are feasible and dangerous in practice and how our library properly repairs them. To the best of our knowledge, this is the first tool that supports information-flow security and concurrency, and the first case study implemented that involves concurrent programs and information-flow policies. The implementation of the library and the case study is publicly available in [TR].

Acknowledgments We wish to thank to the anonymous reviewers and our colleagues in the ProSec group at Chalmers 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

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

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

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

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

[HHMJ05] T. Harris, M. Herlihy, S. Marlow, and S. P. Jones. Composable memory transactions. In *Proc. ACM Symp. on Principles and Practice of Parallel Programming, to appear*, June 2005.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[Hun91] Sebastian Hunt. Pers generalise projections for strictness analysis (extended abstract). In *Proc. 1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, 1991. Springer-Verlag.

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

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

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

[KM06] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *FAST'05*, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

[LZ06] P. Li and S. Zdancewic. Encoding information flow in haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

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

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Pie04] Benjamin C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004.

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

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[RHNS06] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. Annual Asian Computing Science Conference*, LNCS, December 2006.

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

[RS06b] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. PSI'06*, LNCS. Springer-Verlag, June 2006.

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

[Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[Sim03] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

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

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

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

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

[SPJF96] Andrew Gordon Simon Peyton Jones and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[SS99] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming*, volume 1576 of *LNCS*, pages 40–58. Springer-Verlag, March 1999.

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

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

[The] The GHC Team. The glasgow haskell compiler. Software release. `http://haskell.org/ghc/`.

[TR] Ta Chung Tsai and Alejandro Russo. A library for secure multithreaded information flow in haskell. Software release. Available at `http://www.cs.chalmers.se/~russo/tsai.htm`.

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

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

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

CHAPTER

7

A Light-Weight Library for Information-Flow Security in Haskell

Extended version of
A Library for Light-weight Information-Flow Security in Haskell,
In Proceedings of the ACM SIGPLAN 2008 Haskell Symposium,
Victoria, Brithish Columbia, Canada, 25th of September, 2008.

A Light-Weight Library for Information-Flow Security in Haskell

Alejandro Russo, Koen Claessen, and John Hughes

Chalmers University of Technology, Gothenburg, Sweden
{russo,koen,rjmh}@chalmers.se

Abstract. Protecting confidentiality of data has become increasingly important for computing systems. *Information-flow* techniques have been developed over the years to achieve that purpose, leading to special-purpose languages that guarantee information-flow security in programs. However, rather than producing a new language from scratch, information-flow security can also be provided as a library. This has been done previously in Haskell using the arrow framework. In this paper, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve the same goals. We present a *monadic library* to provide information-flow security for Haskell programs. The library introduces mechanisms to protect confidentiality of data for pure computations, that we then easily, and modularly, extend to include dealing with side-effects. We also present combinators to dynamically enforce different declassification policies when release of information is required in a controlled manner. It is possible to enforce policies related to *what*, by *whom*, and *when* information is released or a combination of them. The well-known concept of monads together with the light-weight characteristic of our approach makes the library suitable to build applications where confidentiality of data is an issue.

1 Introduction

Protecting confidentiality of data has become increasingly important for computing systems. Often, software is so complex that it is hard to see if a program can be abused by a malicious person to gain access to private data. This is important when developing software oneself, and becomes increasingly more important if one is forced to trust other people's code.
Information-flow techniques have been developed over the years to achieve this kind of protection. For example, as a result, two main stream compilers, Jif (based on Java) and Flowcaml (based on Ocaml) have been developed to guarantee information-flow security in programs.
However, it is a very heavy-weight solution to introduce a new programming language for dealing with information-flow. In this work, we explore the possibility of expressing restrictions on information-flow as a library rather than a new language.
We end up with a light-weight monadic approach to the problem of expressing and ensuring information-flow in Haskell. Code that exhibits information flows that are disallowed will be ill-typed and rejected by the type checker. Our approach is general enough to deal with practical concepts such as secure reading and writing to files (which can be

generalized to capture any information exchange with the outside world) and declassification (a pragmatic way of allowing controlled information leakage [SS05]).
Our library might be used in scenarios where we want to incorporate in our programs some code written by outsiders (untrusted programmers) to access our private information. Such code can be also allowed to interact with the outside world (for example by accessing the web). We would like to have a guarantee that the program will not send our private data to an attacker. A slightly different, but related, scenario is where we ourselves write the possibly unsafe code, but we want to have the help of the type checker to find possible security mistakes.
Li and Zdancewic [LZ06] have previously shown how to provide information-flow security also as a library, but their implementation is based on *arrows* [Hug00], which naturally requires programmers to be familiar with arrows when writing security-related code. In this work, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve very similar goals.

1.1 Motivating example

Consider a machine running Linux with the default installation of the Shadow Suite [Jac96] responsible to store and manage users' passwords. In this machine, the file `/etc/passwd` contains information regarding users such as user and group ID's, which are used by many system programs. This file must remain world readable. Otherwise, simple commands as `ls -l` stop working. Passwords are properly set in the file `/etc/shadow`, which can only be read and written by root. From now on, we refer to the passwords stored in this file as *shadow passwords*. Programs that verify passwords need to be run as root. From the security point of view, this requirement implies that very careful programming practices must be followed when creating such programs. For instance, if a program running as root has a shell escape, it is not desirable that such shell escape runs with root privileges. The process to verify a password usually consists of taking the input provided by the user, applying some cryptographic algorithms to it, and comparing the result of that with the user's information stored in `/etc/shadow`. Observe that an attacker can encrypt a dictionary of common passwords offline and then, given some file `/etc/shadow`, try to guess users' passwords by checking matches. This attack is known as an *offline dictionary attack* and is one of the most common methods for gaining or expanding unauthorized access to systems [NS05]. In order to obtain the content of `/etc/shadow`, the attacker needs to obtain root privileges, which is not impossible to achieve [Loc08]. Given these facts, we can conclude that there are mainly two security problems with shadow passwords: programs require having root privileges to verify passwords and offline dictionary attacks. We start dealing with these problems by firstly limiting the access to the password file. With this in mind, we assume that information stored in `/etc/shadow` is only accessible through an API. The following Haskell code shows an example of such API.

```
data Spwd = Spwd { uid :: UID, cypher :: Cypher }

getSpwdName :: Name -> IO (Maybe Spwd)
putSpwd     :: Spwd -> IO ()
```

Data type `Spwd` stores users' identification number (`uid::UID`) and users' password (cypher :: Cypher). For a simple presentation, we assume that passwords are stored as plain text and not cyphers. Function `getSpwdName` receives a user name and returns his (her) password if such user exists. Function `putSpwd` takes a register of type `Spwd` and adds it to the shadow password file. This API is now the only way to have access to shadow passwords. We can still be more restrictive and require that such API is only called under root privileges, which is usually the case for Unix-like systems. Unfortunately, this restriction does not help much since attackers could obtain unauthorized root access and then steal the passwords. However, by applying information-flow techniques to the API and programs that use it, it is possible to guarantee that passwords are not revealed while making possible to verify them. In other words, offline dictionary attacks are avoided as well as some requirements as having root privileges to verify passwords. In Section 3.3, we show a secure version of this API.

1.2 Contributions

We present a *light-weight library* for *information-flow security* in Haskell. The library is *monadic*, which we argue is easier to use than arrows, which were used in previous attempts. The library has a pure part, but also deals with *side-effects*, such as the secure reading and writing of files. The library also provides novel and powerful means to specify *declassification policies*.

1.3 Assumptions

In the rest of the paper, we assume that the programming language we work with is a controlled version of Haskell, where code is divided up into *trusted* code, written by someone we trust, and *untrusted* code, written by the attacker. There are no restrictions on the trusted code. However, the untrusted code has certain restrictions; certain modules are not available to the untrusted programmer. For example, all modules providing IO functions, including exceptions (and of course `unsafePerformIO`) are not allowed. Our library will reintroduce part of that functionality to the untrusted programmer in a controlled, and therefore, secure way.

2 Non-interference for pure computations

Non-interference is a well-known *security policy* that preserves confidentiality of data [Coh78, GM82]. It states that public outcomes of programs do not depend on their confidential inputs.

In imperative languages, information leaks arise from the presence of *explicit* and *implicit* flows inside of programs [DD77]. Explicit flows are produced when secret data is placed explicitly into public locations by an assignment. Implicit flows, on the other hand, use control constructs in the language in order to reveal information. In a pure functional language, however, this distinction becomes less meaningful, since there are no assignments nor control constructs. For example, a conditional (if-then-else) is just a function as any other function in the language. In a pure language, all information-flow is explicit; information only flows from function arguments to function results.

To illustrate information leaks in pure languages, we proceed assuming that a programmer, potentially malicious, needs to write a function

```
f :: (Char,Int) -> (Char,Int)
```

where characters and integers are considered respectively secret and public data. We assume that attackers can only control public inputs and observe public results when running programs, and can thus only observe the second component of the pair returned by function `f`. For simplicity, we also assume that type `Char` represents ASCII characters.
If a programmer writes the code

```
f (c, i) = (chr (ord c + i), i+3)
```

then the function is non-interferent and preserves the confidentiality of `c`; the public output of `f` is independent of the value of `c` [1]. If a programmer instead writes

```
f (c, i) = (c, ord c)
```

then information about `c` is revealed, and the program is not non-interferent! Attackers might try to write less noticeable information leaks however. For instance, the code

```
f (c, i) = (c, if ord c > 31 then 0 else 1)
```

leaks information about the printability of the character `c` and therefore should be disallowed as well.
In this section, we show how monads can be used to avoid leaks and enforce the non-interference property for pure computations.

2.1 The Sec monad

In order to make security information-flow specific, we are going to make a distinction at the type level between *protected* data and *public* data. Protected data only lives inside a special *monad* [Wad92]. This security monad makes sure that only the parts of the code that have the right to do so are able to look at protected data.
In larger programs, it becomes necessary to talk about several *security levels* or *areas*. In this case, values are not merely protected or public, but they can be protected by a certain security level `s`.
Take a look at Fig. 1, which shows the API of an abstract type, `Sec`, which is a functor and a monad. There are two functions provided on the type `Sec`; `sec` is used to protect a value, and `open` is used to look at a protected value. However, to look at a protected value of type `Sec s a`, one needs to have a value of type `s`. Restricting access to values of different such types `s` by means of the module system allows fine control over which parts of the program can look at what data. (For this to work, `open` needs to be strict in its second argument.)
For example, if we define a security area `H` in the following way:

```
data H = H
```

[1] Function `chr` returns an exception when the received argument does not represent an ASCII code. By observing occurrences of exceptions or computations that diverge, an attacker can deduce some information about secrets. However, we only consider programs that terminate successfully.

```
newtype Sec s a

instance Functor (Sec s)
instance Monad (Sec s)

sec :: a -> Sec s a
open :: Sec s a -> s -> a
```

Fig. 1. The Sec monad

```
module Lattice where

data L = L
data H = H

class Less sl sh where
 less :: sh -> sl -> ()

instance Less L L where
 less _ _ = ()

instance Less L H where
 less _ _ = ()

instance Less H H where
 less _ _ = ()
```

Fig. 2. Implementation of a two-point lattice

then we can model the type of the function `f` given in the beginning of this section as follows:

```
f :: (Sec H Char, Int) -> (Sec H Char, Int)
```

The first, secure, example of `f` can be programmed as follows:

```
f (sc,i) = ((\c -> chr (ord c + i)) ‘fmap‘ sc,i+3)
```

However, the other two definitions can not be programmed without making use of `H` or breaking the type checker.
So, for a part of the program that has no means to create non-bottom values of a type `s`, direct access to protected values of type `Sec s a` is impossible. However, computations involving protected data are possible as long as the data stays protected. This can be formalized by stating that type `Sec` guarantees a *non-interference* property. For any type `A`, and values `a1, a2 :: A`, a function

```
f :: Sec H A -> Bool
```

will produce the same result for arguments `a1` and `a2`. 08 See Appendix 7 for more details.
We will later show the implementation of the type `Sec` and its associated functions.

2.2 Security lattice

Valid information flows inside of programs are determined by a *lattice on security levels* [Den76]. Security levels are associated to data in order to establish its degree of confidentiality. The ordering relation in the lattice, written $\sqsubseteq$, represents allowed flows.

For instance, $l_1 \sqsubseteq l_2$ indicates that information at security level l_1 can flow into entities of security level l_2.

For simplicity, in this paper, we will only use a two-point lattice with security levels H and L where L $\sqsubseteq$ H and H $\not\sqsubseteq$ L. Security levels H and L denote secret (*high*) and public (*low*) information, respectively. The implementation of the lattice is shown in Figure 2. Type class Less encodes the relation $\sqsubseteq$ and security levels are represented as singleton types [Pie04]. The role of less is explained in Section 4. Public information is characterized by the security level L. Constructor L is then publicly available so that data at security level L can be observed by anyone, which also includes attackers.

As explained earlier, attackers must have no access to the constructor H. In Section 4, we describe how to achieve such restriction.

Finally, to capture the fact that valid information flows occur from *lower* (L) to higher (H) security levels, we introduce the function

```
up :: Less sl sh => Sec sl a -> Sec sh a
```

The function up can be used to turn any protected value into a protected value at a higher security level. The implementation of up will be shown later.

3 Non-interference and side-effects

The techniques described in Section 2 do not perform computations with side-effects. The reason for that is that side-effects involving confidential data cannot be executed when they are created inside of the monad Sec s.

Even if we allowed a restricted and secure form of file reading and writing in the IO-monad, that would still not be enough. For example, if we, read information from file A, and depending on the value of a secret, want to write either to a file B or file C, we would obtain a computation of type IO (Sec H (IO ())). It is easy to see that these types quickly become unmanagable, and, more importantly, unusable.

In this section, we show how we can augment our security API to be able to deal with *controlled* side-effects while still maintaining non-interference properties.

In this paper, we concentrate how to provide an API that allows reading and writing protected data from and to files. For this to work properly, files need to contain a security level, so that only data from the right security level can be written to a file. We assume that the attacker has no way of observing what side-effects were performed, other than through our API. (The attacker, so to say, sits within the Haskell program and has no way of getting out[2].)

The ideas for reading and writing files can be extended to deal with many other controlled IO operations, such as creating, reading and writing secure references, communicating over secure channels, etc. We will however not deal with the details of such operations in this paper.

[2] A situation where the attacker is in league with a hacker who has gotten access to our system, and can for example read log files, is beyond our control and the guarantees of our library.

3.1 Secure files

We model all interactions with the outside world by operations for reading and writing files [Tan01]. For that reason, we decide to include secure file operations in our library. We start by assigning security levels to files in order to indicate the confidentiality of their contents. More precisely, we introduce the abstract data type `File s`. Values of type `File s` represent names of files whose contents have security level `s`. These files are provided by the trusted programmer. We assume that attackers have no access to the internal representation of `File s`. In Section 4, we show how to guarantee such assumption.
A first try for providing secure file operations is to provide the following two functions:

```
readSecIO  :: File s -> IO (Sec s String)
writeSecIO :: File s -> Sec s String -> IO ()
```

These functions do not destroy non-interference, because they do not open up for extra information-flow between security levels. The data read from a file with security level `s` is itself protected with security level `s`, and any data of security level `s` can be written to a file of security level `s`.
However, the above functions are not enough to preserve confidentiality of data. Take a look at the following program:

```
writeToAFile :: Sec H String -> Sec H (IO ())
writeToAFile secs =
  (\s -> if length s < 10
           then writeSecIO file1 s
           else writeSecIO file2 s) ‘fmap‘ secs
```

Here, `file1, file2 :: File H` is assumed to be defined elsewhere.
The behavior of the above function is indeed dependent on the protected data in its argument, as indicated by the result type. However, only the *side-effects* of the computation are dependent on the data, not the *result value*. Why is this important? Because we assume that the attacker has no way of observing from within the program what these side-effects are! (Unless the attacker can observe the results of the side-effects, namely the change of file contents in either `file1` or `file2`, but that information can only be obtained by someone with the appropriate security clearance anyway.) This assumption is valid for the scenarios described in Section 1.
In other words, since side-effects cannot be observed from within a program, we are going to allow the leakage of side-effects. Our assumption is only true if we restrict the IO actions that the attacker can perform.

3.2 The SecIO monad

To this end, we introduce a new monad, called `SecIO`. This monad is a variant of the regular IO monad that keeps track of the security level of all data that was used inside it.
Take a look at Fig. 3, which shows the API for an abstract type `SecIO`, which is a functor and a monad. Values of type `SecIO s a` represent computations that can

```
newtype SecIO s a

instance Functor (SecIO s)
instance Monad (SecIO s)

value :: Sec s a -> SecIO s a

readSecIO  :: File s' -> SecIO s (Sec s' String)
writeSecIO :: File s -> String -> SecIO s ()

plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
run  :: SecIO s a -> IO (Sec s a)
```

Fig. 3. The SecIO monad

securely read from any file, securely write to files of security level `s` (or higher), and look at data protected at level `s` (or lower).

The function `value` can be used to look at a protected value at the current security level. The function `readSecIO` reads protected data from files at any security level, protecting the result as such. The function `writeSecIO` writes data to files of the current security level.

The function `plug` is used to import computations with side-effects at a high level into computations with side-effects at a low level of security. Observe that only the side-effects are "leaked", not the result, which is still appropriately protected by the high security level. This function is particularly suitable to write programs that contain loops that depend on public information and perform, based on secret and public data, side-effects on secret files in each iteration.

These functions together with the return and bind operations for `SecIO s` constitute the basic interface for programmers.

Based on that, more convenient and handy functions can then be defined. For instance,

```
s_read :: Less s' s => File s' -> SecIO s String
s_read file =  do ss <- readSecIO file
                  value (up ss)

s_write :: Less s' s =>
           File s -> String -> SecIO s' (Sec s ())
s_write file str = plug (writeSecIO file str)
```

Observe that `s_read` and `s_write` have simpler types while practically providing the same functionality as `readSecIO` and `writeSecIO`, respectively.

In the next section, we show how to implement the core part of our library: the monads `Sec s` and `SecIO s`. We continue this section with an example that shows how these APIs can be used.

3.3 Developing a secure shadow passwords API

As an example of how to apply information-flow mechanisms, we describe how to adapt the API described in the introduction to guarantee that neither API's callers or the API itself reveal shadow passwords. Specifically, passwords cannot be copied into public files at all. Hence, offline dictionary attacks are avoided as well as the requirement of having root privileges to verify passwords. As mentioned in the introduction, we assume that the contents of `/etc/shadow` is only accessible through the API. For simplicity, we assume that this file is stored in the local file system, which naturally breaks the assumption we have just mentioned (user root has access to all the files in the system). However, it is not difficult to imagine an API that establishes, for example, a connection to some sort of password server in order to get information regarding shadow passwords.
We firstly start adapting our library to include the two-point lattice mentioned in Section 2. We decide to associate security level `H`, which represents secret information, to data regarding shadow passwords. Then, we indicate that file `/etc/shadow` stores secret data by writing the following lines

```
shadowPwds :: File H
shadowPwds  = MkFile "/etc/shadow"
```

We proceed to modify the API to indicate what is the secret data handled by it. More precisely, we redefine the API as follows:

```
getSpwdName :: Name -> IO (Maybe (Sec H Spwd))
putSpwd     :: Sec H Spwd -> IO ()
```

where values of type `Spwd` are now "marked" as secrets [3]. The API's functions are then adapted, without too much effort, to meet their new types. In order to manipulate data inside of the monad `Sec H`, API's callers need to import the library in their code. Since `/etc/shadow` is the only file with type `File H` in our implementation, this is the only place where secrets can be stored after executing calls to the API. By marking values of type `Spwd` as secrets, we restrict how information flows inside of the API and API's callers while making possible to operate with them. In Section 5, we show how to implement a login program using the adapted API.

4 Implementation of monads Sec and SecIO

In this section, we provide a possible implementation of the APIs presented in the previous two sections.
In Fig. 4 we show a possible implementation of `Sec`. `Sec` is implemented as an identity monad, allowing access to its implementation through various functions in the obvious way. The presence of `less` in the definition of function `up` includes `Less` in its typing constrains. Function `unSecType` is used for typing purposes and has no computational meaning. Note the addition of the function `reveal`, which can reveal any protected

[3] Values of type `Maybe` are not included inside of `Sec H` since the existence of passwords is linked to the existence of users in the system, which is considered public information.

```
module Sec where

-- Sec
newtype Sec s a = MkSec a

instance Monad (Sec s) where
  return x = sec x

  MkSec a >>= MkSec k =
    MkSec (let MkSec b = k a in b)

sec :: a -> Sec s a
sec x = MkSec x

open :: Sec s a -> s -> a
open (MkSec a) s = s ‘seq‘ a

up :: Less s s’ => Sec s a -> Sec s’ a
up sec_s@(MkSec a) = less s s’ ‘seq‘ sec_s’
                     where (sec_s’) = MkSec a
                           s        = unSecType sec_s
                           s’       = unSecType sec_s’

-- For type-checking purposes (not exported).
unSecType :: Sec s a -> s
unSecType _ = undefined

-- only for trusted code!
reveal :: Sec s a -> a
reveal (MkSec a) = a
```

Fig. 4. Implementation of Sec monad

value. This function is not going to be available to the untrusted code, but the trusted code might sometimes need it. In particular, the implementation of `SecIO` needs it in order to allow the leakage of side-effects.

In Fig. 5 we show a possible implementation of `SecIO`. It is implemented as an IO computation that produces a safe result. As an invariant, the IO part of a value of type `SecIO s a` should only contain unobservable (by the attacker) side-effects, such as the reading from and writing to files.

There are a few things to note about the implementation. Firstly, the function `reveal` is used in the implementation of monadic bind, in order to leak the *side-effects* from the protected IO computation. Remember that we assume that the performance of side-effects (reading and writing files) cannot be observed by the attacker. Some leakage of side-effects is unavoidable in any implementation of the functionality of `SecIO`. Secondly, the definition of the type `File` does not make use of its argument `s`. This is also unavoidable, because it is only by a promise from the trusted programmer that certain

```
module SecIO where
import Lattice
import Sec

-- SecIO
newtype SecIO s a = MkSecIO (IO (Sec s a))

instance Monad (SecIO s) where
  return x = MkSecIO (return (return x))

  MkSecIO m >>= k =
    MkSecIO (do sa <- m
                let MkSecIO m' = k (reveal sa)
                m')

-- SecIO functions
value :: Sec s a -> SecIO s a
value sa = MkSecIO (return sa)

run :: SecIO s a -> IO (Sec s a)
run (MkSecIO m) = m

plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
plug ss_sh@(MkSecIO m)
     = less sl sh `seq` ss_sl
        where
          (ss_sl) = MkSecIO (do sha <- m
                                return (sec sha))
          sl      = unSecIOType ss_sl
          sh      = unSecIOType ss_sh

-- For type-checking purposes (not exported).
unSecIOType :: SecIO s a -> s
unSecIOType _ = undefined

-- File IO
data File s = MkFile FilePath

readSecIO :: File s' -> SecIO s (Sec s' String)
readSecIO (MkFile file) =
  MkSecIO ((sec . sec) `fmap` readFile file)

writeSecIO :: File s' -> String -> SecIO s ()
writeSecIO (MkFile file) s =
  MkSecIO (sec `fmap` writeFile file s)
```

Fig. 5. Implementation of SecIO monad

```
module SecLibTypes ( L (..), H,  Less () ) where
import Lattice


module SecLib
 (   Sec, open, sec, up
   , SecIO, value, plug, run,
   , File, readSecIO, writeSecIO, s_read, s_write
 )
where

import Sec
import SecIO
```

Fig. 6. Modules to be imported by untrusted code

files belong to certain security levels. Thirdly, function `plug`, similarly to function `up`, includes `less` and an auxiliary function (`unSecIOType`) to properly generate type constraints.
The modules `Sec`, `SecIO`, and `Lattice` can only be used by trusted programmers. The untrusted programmers only get access to modules `SecLibTypes` and `SecLib`, shown in Fig. 6. They import the three previous modules, but only export the trusted functions. Observe that the type `L` and its constructor `L` are exported, but for `H`, only the type is exported and not its constructor. Method `less` is also not exported. Therefore, functions `up` and `plug` are only called with the instances of `Less` defined in `Lattice.hs`.
In order to check that a module is safe with respect to information-flow, the only thing we have to check is that it does not import trusted modules, in particular:

- `Sec` and `SecIO`
- any module providing exception handling, for example `Control.Monad.Exception`,
- any module providing unsafe extensions, for example `System.IO.Unsafe`

5 Declassification

Non-interference is a security policy that specifies the absence of information flows from secret to public data. However, real-word applications release some information as part of their intended behavior. Non-interference does not provide means to distinguish between intended releases of information and those ones produced by malicious code, programming errors, or vulnerability attacks. Consequently, it is needed to relax the notion of non-interference to consider *declassification* policies or intended ways to leak information. In this section, we introduce run-time mechanisms to enforce some declassification policies found in the literature.

Declassification policies have been recently classified in different dimensions[SS05]. Each dimension represents aspects of declassification. Aspects correspond to *what*, *when*, *where*, and by *whom* data is released. In general, type-systems to enforce different declassification policies include different features, e.g rewriting rules, type and effects, and external analysis [ML00, SM04, CM04]. Encoding these features directly into the Haskell type system would considerably increase the complexity of our library. For the sake of simplicity and modularity, we preserve the part of the library that guarantees non-interference while orthogonally introducing run-time mechanisms for declassification. More precisely, declassification policies are encoded as programs which perform run-time checks at the moment of downgrading information. In this way, declassification policies can be as flexible and general as programs! Additionally, we provide functions that automatically generate declassification policies based on some criteria. We call such programs *declassification combinators*. We provide combinators for the dimensions *what*, *when*, and *who* (*where* can be thought as a particular case of *when*). As a result, programmers can combine dimensions by combining applications of these combinators.

5.1 Escape hatches

In our library, declassification is performed through some special functions. By borrowing terminology introduced in [SM04], we call these functions "escape hatches" and we represent them as follows.

```
type Hatch s s' a b = Sec s a -> IO (Maybe (Sec s' b))
```

Escape hatches are functions that take some data at security level `s`, perform some computations with it, and then probably return a result depending if downgrading of information to security level `s'` is allowed or not. Arbitrary escape hatches can be included in the library depending on the declassification policies needed for the built applications. In fact, escape hatches are just functions. Types `IO` and `Maybe` are present in the definition of `Hatch s s' a b` in order to represent run-time checks and the fact that declassification may not be possible on some circumstances. By placing `Maybe` outside of monad `Sec s'`, the fact that declassification is possible or not is public information and programs can thus take different actions in each case. Consequently, it is important to remark that declassification policies should not depend on secret values in order to avoid unintended leaks (we give examples of such policies later). Otherwise, it would be possible to reveal information about secrets by inspecting the returned constructor (`Just` or `Nothing`) when applying escape hatches.
As mentioned in the beginning of the section, we include some *declassification combinators* that are responsible for generating escape hatches. The simplest combinator creates escape hatches that always succeed when downgrading information. Specifically, we define the following combinator.

```
hatch :: Less s' s => (a -> b) -> Hatch s s' a b
hatch f = \sa -> return(Just(return(f (reveal sa))))
```

Basically, `hatch` takes a function and returns an escape hatch that applies such function to a value of security level `s` and returns the result of that at security level `s'` where `s'` $\sqsubseteq$ `s`. Observe how the function `reveal` is used for declassification.

The idea is that the function `hatch` is used by trusted code in order to introduce a controlled amount of leaking to the attacker. Note that it is possibly dangerous for the trusted code to export a *polymorphic* escape hatch to the attacker! A polymorphic function can often be used to leak an unlimited amount of information, by for example applying it to lists of data. In general, escape hatches that are exported should be monomorphic.

5.2 The What dimension

In general, type systems that enforce declassification policies related to "what" information is released are somehow conservatives [SM04, AS07, MR07]. The main reason for that is the difficulty to statically predict how the data to be declassified is manipulated or changed by programs. Inspired by quantitative information-theoretical works [CHM02], we focus on "how much" information can be leak instead of determining exactly "what" is leaked. In this light, we introduce the following declassification combinator.

```
ntimes :: Int -> Hatch s s' a b -> IO (Hatch s s' a b)
ntimes n f
 = do ref <- newIORef n
      return (\sa -> do k <- readIORef ref
                        if k <= 0
                           then do return Nothing
                           else do writeIORef ref (k-1)
                                   f sa )
```

Essentially, `ntimes` takes a number `n` and an escape hatch `h`, and returns a new escape hatch that produces the same result as `h` but that can only be applied at most `n` times. To achieve that, the combinator creates the reference `ref` to the number of times (`n`) that the escape hatch (`h`) can be applied. Every application of the escape hatch then checks if the maximum number of allowed applications has been reached by observing the condition `k <= 0`. Additionally, every application of the escape hatch also reduce the number of possible future applications by executing `writeIORef ref (k-1)`. The generated escape hatch returns `Nothing` if the policy is violated as a manner to avoid leaking more information than intended. Inspecting if the result of applying an escape hatch is `Nothing` or not can be considered as a covert channels by itself when happening inside of computations related to confidential data. Fortunately, escape hatches applied inside of computations depending on secrets are never executed. For instance, if we try to apply an escape hatch inside of some secret computation, it will have the type `Sec H (IO (Maybe (Sec L b)))` for some type `b`. Declassification is performed inside of the `IO` monad and it is not possible to extract `IO` computations from the monad `Sec H` unless than another escape hatches is declared to release `IO` computations. Therefore, escape hatches must be introduced to release pure values rather than side-effecting computations, which seems to be the case for most applications.
Note that the function `ntimes` is safe to be exported to the attacker, since it only restricts the use of existing hatches.

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import SpwdData

declassification
 = ntimes 3 (hatch (\(spwd,c) -> cypher spwd == c))
   :: IO (Hatch H L (Spwd, String) Bool)


module Main ( main ) where
import Policies
import Login

main = do match <- declassification
          login match


module Login ( login ) where
import SecLibTypes ; import SecLib
import SpwdData ; import Spwd
import Maybe

check :: (?match :: Hatch H L (Spwd, Cypher) Bool)
         => Sec H Spwd -> String -> Int -> String
            -> IO ()
check spwd pwd n u =
  do acc <- ?match ((\s -> (s, pwd)) ‘fmap‘ spwd)
     if (public (fromJust acc))
               then putStrLn "Launching shell..."
               else do putStrLn "Invalid login!"
                       auth (n-1) u spwd

auth 0 _ spwd = return ()
auth n u spwd = do putStr "Password:"
                   pwd <- getLine
                   check spwd pwd n u

login match
  = do let ?match = match
       putStrLn "Welcome!"
       putStr "login:"
       u <- getLine
       src <- getSpwdName u
       case src of
            Nothing    -> putStrLn "Invalid user!"
            Just spwd  -> auth 3 u spwd
```

Fig. 7. Secure login program

As an example of how ntimes can be used, we write a login program that uses the secure shadow password API described in Section 3.3. It is not possible to write such program without having means for declassification. The login program must release some information about users' passwords: if access is granted, then the attacker knows that his input matches the password, otherwise he knows that it does not. We present the program in Figure 7. Module Policies introduces declassification policies for our login program and states that a shadow password can be compared by equality at most three times. This module is trusted and must not be imported by untrusted code. Otherwise, attackers can create an unrestricted number of escape hatches in order to leak secrets! Module SecLibTypes, described in Section 4, is extended to include type definitions related to declassification as, for instance, Hatch s s' a b. Module Declassification introduces declassification combinators (e.g. ntimes). These modules are part of our trusted base. Module Declassification must not be imported by untrusted code for the same reasons given for module Policies. Modules SpwdData and Spwd respectively include the data type declaration of Spwd and the API described in Section 3.3. Module Main extracts declassification policies defined in Policies and pass them to the login function. In general, this module determines what functions are called from untrusted code in order to run the program. In this case, it determines that login must be called to perform the login procedure. Since the module imports module Policies, it also belongs to the trusted base. The most interesting module is Login. This module does not belong to our trusted base and therefore it may contain code written by possibly malicious programmers. Because declassification policies can be applied at any part of the untrusted code, we place them into implicit parameters [LLMS00]. Implicit parameters can be thought as some kind of global variables and they are declared by writing variable names starting with the symbol ?. Module Login contains three functions: check, auth, and login. Function check takes the password spwd :: Sec H Spwd stored in the system for the user u :: String and checks, by applying the escape hatch placed in ?match, if the user's input pwd :: String matches the password stored in the field cypher of spwd. Assuming that is possible to perform the declassification described by ?match, variable acc stores if the access is granted or not. We assume that untrusted code has access to the function public :: Sec L a -> a to extract the public values from monad Sec L. In the example, public is applied to values returned by ?match. If the access is denied, check might give another chance to the user by calling the function auth. Function auth is responsible to ask the user's password and validates it at most n times. Function login asks for the user name and checks that the user is registered in the system by calling the function getSpwdName from the secure shadow password API.

Since program in Figure 7 type-checks, it respects the declassification policies defined in module Policies, i.e. the password can be compared for equality only three times. To illustrate that, we place our selves in the role of the attacker and modify function check to call auth n u spwd instead. As a result, it would be now possible to try as many passwords as the user wants and thus increasing the amount of information leak by unit of time. Observe that this situation is particularly dangerous when passwords have short length as PIN numbers in ATMs. Nevertheless, if we try to run the modified code, we get the message `*** Exception: Maybe.fromJust: Nothing` after the user tries more than three times to check if the access can be granted or not.

5.3 The When dimension

```
module Bid ( bid ) where

obtainBid :: FilePath -> IO Int
obtainBid file = do s <- readFile file
                    return (read s :: Int)

bid  = do putStrLn "Bid system!"
          putStrLn "-----------"
          putStrLn ""
          putStrLn "Obtaining the bids..."
          a <- obtainBid "bidA"
          -- writeFile "bidB" (show (a+1))
          b <- obtainBid "bidB"
          putStrLn (if a > b then "A wins!"
                              else "B wins!")
```

Fig. 8. Insecure bidding system

As a motivating example for handling this dimension, we can consider the scenario described in [CM04] of a sealed auction where each bidder submits a single secret bid in a sealed envelope. Once all bids are submitted, the envelopes are opened and the bids are compared. The highest bidder wins. One security property that is important for this program is that no bidder knows any of the other bids until all the bids have been submitted. Program in Figure 8 simulates this process for two bidders: `A` and `B`. We represent envelopes as files. Function `obtainBid` opens an envelope and extracts the bid. The rest of the program is self-explanatory. It is possible to incorrectly implement the auction protocol by mistake or intentionally. For instance, if we uncommented the line in Figure 8, the program uses the bid from user `A` to make user `B` the winner. However, no information about `A`'s bid must be available until `B` submits his (her) own bid.
The library introduces the when dimension by associating events in the system that indicates at which time release of information may occur. For instance, “releasing a software key may occur after the payment has been confirmed”. Inspired by [BS06], we implement boolean flags called *flow locks* [4] that, when open, allow downgrading of information.

[4] The notion presented here about flow locks is not exactly the same that is introduced in Broberg and Sands's paper. For instance, their work can statically check if a program respects the declassification policies determined by the flow locks. Moreover, the state of the locks is not

Flow locks are introduced by the following combinator.

```
when :: Hatch s s' a b ->
        IO (Hatch s s' a b, Open, Close)
when f = do ref <- newIORef False
            return (\sa -> do b <- readIORef ref
                              if b then f sa
                                   else return Nothing
                    , writeIORef ref True
                    , writeIORef ref False)
```

Basically, `when` takes an escape hatch `h` and returns a new escape hatch that produces the same result as `h` but that has associated a flow lock to it. The combinator creates the reference `ref` to an initially close flow lock represented as `False`. The returned escape hatch can only be applied when the associated flow lock is open (i.e. the corresponding boolean flag is set to `True`). Observe that, by inspecting the value of `b`, every application of the escape hatch checks that the flow lock is open before declassifying information. The combinator also returns computations to open and close the lock, which respectively have type `Open` and `Close`. These computations must be only used by trusted code. Otherwise, the attacker can execute them at any time in the untrusted code and thus ignoring the events that indicate when declassification may occur. `Open` and `Close` are just synonymous type declarations for `IO ()`.
We can then implement a secure bidding system. We firstly define our security lattice composed by the security levels `A`, `B`, and `L`, where $\mathtt{L} \sqsubseteq \mathtt{A}$ and $\mathtt{L} \sqsubseteq \mathtt{B}$. Security levels `A` and `B` are respectively associated to information coming from users `A` and `B`, while `L` denotes public information. We implement these security levels as singleton types with constructors `A :: A`, `B :: B`, and `L::L`. The described security lattice is very simple and therefore we omit details about its implementation. The secure bidding system is shown in Figure 9. At first glance, it might seem that this implementation is much more complex than the insecure one. However, the module `Bid`, the core of the bidding system, has approximately the same size as before. The rest of the modules are related to properly setting up the security level of different resources in the program as well as the corresponding declassification policies. Module `Files` declares the security level `A` and `B` for the files that store the bids of users `A` and `B`, respectively. Module `Policies` defines the escape hatches `ha` and `hb` to release information that belongs to users `A` and `B`, respectively. Computations `openA` and `closeA` (`openB` and `closeB`) open and close the flow lock associated to `hA`(`hB`), respectively. As mentioned before, the opening and closing of locks are produced by trusted code. In this case, the opening of locks happens when bids are read from files. We then place function `obtainBid` in the trusted module `Main`. We also adapt such function to read files at security level `s` and return their contents, but opening the flow lock received as argument. Function `main` obtains the escape hatches from `declassification` and defines trusted function responsible for opening flow locks. Function `obtainBidA` (`obtainBidB`) reads the bid of user `A` (`B`) and opens the lock for releasing the bid of user `B` (`A`). Differently from

related with the state of programs at all. We differ from these two points due to the dynamic nature of our approach. However, the intuitive idea of allowing downgrading of information when locks are open is preserved in our implementation.

```
module Files ( bidAF, bidBF ) where
import Sec (secret, File (File) )
import Lattice

bidAF :: File A
bidAF = MkFile "bidA"

bidBF :: File B
bidBF = MkFile "bidB"


module Policies ( declassification ) where
import SecLibTypes ; import Declassification

declassification
 = do (pA :: Hatch A L Int Int, openA, closeA)
         <- when (hatch id)
      (pB :: Hatch B L Int Int, openB, closeB)
         <- when (hatch id)
      return (pA, openA, closeA, pB, openB, closeB)


module Main ( main ) where
import Policies ; import Files ; import SecLib
import Bid

obtainBid :: File s -> Open -> IO (Sec s Int)
obtainBid file open
  = do sec <- run (do r <- s_read file
                      return (read r :: Int))
       open
       return sec

main
  = do (hA, openA, closeA,
        hB, openB, closeB) <- declassification
       let obtainBidA = obtainBid bidAF openB
           obtainBidB = obtainBid bidBF openA
       bid hA obtainBidA hB obtainBidB


module Bid ( bid ) where
import SecLibTypes ; import SecLib

bid hA obtainBidA hB obtainBidB
 = do putStrLn "Bid system!"
      putStrLn "-----------"
      putStrLn ""
      putStrLn "Obtaining the bids..."
      bidA   <- obtainBidA
      -- Just cheat <- hA bidA
      bidB   <- obtainBidB
      Just seca <- hA bidA
      Just secb <- hB bidB
      putStrLn(if (public seca) > (public secb)
                  then "A wins!"
                  else "B wins!")
```

Fig. 9. Secure bidding system

the insecure version in Figure 8, function `bid` receives as arguments escape hatches and functions to obtain bids. Module `Bid` is written by the attacker or possibly malicious programmer. In this module, function `bid` obtains the bids to later compare them. In order to compare bids, they need to be extracted from values of type `Sec A Int` and `Sec B Int` through the escape hatches `ha` and `hb`, respectively. It is then not possible to determine which bid is the highest before obtaining all for them. For instance, if we uncommented the line in function `bid`, we obtain a program that tries to release the bid from user `A` before getting the bid for user `B`, which is clearly a non-desirable behavior for the auction system. However, if we run the program, we get the message `*** Exception: Maybe.fromJust: Nothing` since the flow lock associated to release `A`'s bid is not open. In order to open it, we firstly need to get `B`'s bid!
To illustrate why flow locks may need to be closed, we take the example on step further by thinking of a bidding system that allows the users to bid more than once. In this case, function `bid` is called several times and flow locks related to `hA` and `hB` must be closed between each call. Otherwise, all the flow locks are open at the second call of `bid`, which allows bids to be released at any time. It is not difficult to imagine this implementation by considering that function `main` calls computations `closeA` and `closeB` before each call of `bid`.
For simplicity, we considered an auction system with only two users. However, it is possible to use flow locks when more users are present in the auction. Indeed, we can create escape hatches that are associated to as many flow locks as users. In order to do that, we can compose `when` with itself as many times as users we have in the system. In this way, the escape hatch obtained in the end is associated to as many flow locks as users. Then, when a user submits its bid, his corresponding flow lock is open.
Attackers can still write programs that wrongly implement the auction system. For instance, we can write a program that makes user `A` the winner all the time by just replacing the `if-then-else` in Figure 9 by `putStrLn "The user A wins!"`. However, user `A` is going to be the winner because the program is not implemented correctly, but not because the program "cheated" by inspecting `B`'s bid. Correctness of programs are stronger properties than those ones captured by declassification policies.

5.4 The Who dimension

In the *Decentralized Label Model* (DLM) [ML97, ML98, ML00] data is marked with a set of principals who owns the information. While executing a program, the code is authorized to act on behalf of some set of principals known as *authority*. Then, declassification makes a copy of the released data and marked it with the same principals as before the downgrading but excluding those ones appearing in the authority of the code. We do not consider situations where some principals can act on behalf of others.
Similarly to [LZ06], we adapt the idea of DLM to work on a security lattice. Authorities are assigned with a security level l in the lattice and they are able to declassify data at that security level. To achieve that, we introduce a declassification combinator that checks the authority of the code before applying an escape hatch. As indicated in [BS06], DLM can be expressed using flow locks. Fortunately, our implementation is also suitable for that. More precisely, we have the following declassification combinator.

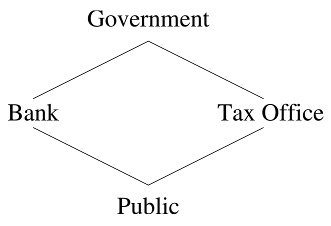

Fig. 10. Security lattice

```
data Authority s = Authority Open Close

who :: Hatch s s' a b -> IO (Hatch s s' a b, Authority s)
who f = do (whof, open, close) <- when f
           return (whof, Authority open close)

certify :: s -> Authority s -> IO a -> IO a
certify s (Authority open close) io =
  s `seq` (do open ; a <- io ; close ; return a)
```

Combinator `who` takes an escape hatch an returns another escape hatch that is associated with a flow lock. The main idea here is that the flow lock is open when the code runs under the same authority as the security level appearing as the argument of the escape hatch. The mechanisms to open and close the flow lock are placed inside of the data type `Authority s`. The constructor of this data type is not accessible for attackers. Otherwise, they can avoid the certification process to determine that some piece of code runs under some authority. Such certification process is carried out by the function `certify`. This function takes an element of security type `s`, an `Authority s`, and a computation `IO a`. In Section 4, we explain that constructors that belongs to security levels above the security level of the attacker are not exported. For instance, in the two-point lattice considered so far, attackers can only observe data at security level `L`, and thus constructor `H :: H` is not exported to untrusted modules. This assumption needs to be relaxed in order to consider this dimension for declassification. To certify that some code has authority `s`, we require that such code, possibly malicious, has only access to the constructors for security level `s` and the security level denoting public information. In this way, it is reflected that code running under authority `s` can freely declassify data from security level `s` as expected in DLM. Function `certify` checks that it receives a valid constructor for the security type `s` by applying `seq` to it, and then respectively opens and closes a flow lock before and after running the `IO` computation received as argument. Observe that this function can be freely used by attackers since it requires to provide the right constructor for some security level `s` and only authorities at that level must have it. Therefore, assignments of authorities to pieces of code must be clearly part of the trusted code.

As a motivating example for this dimension, we start consider the security lattice in Figure 10. We have the security levels: Government, Bank, Tax Office, and Public to represent information related to citizens that is used for such entities. Unless that in-

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import Data

declassification
 = do (hB :: (Hatch B L Account Status),
       authBank) <- who (hatch status)
      (hT :: (Hatch T L Citizen Address),
       authTax)  <- who (hatch address)
      (hG :: (Hatch G T Immigrant Citizen),
       authG) <- who (hatch inmigrants)
      (hG' :: (Hatch G B Study Result),
       authG') <- who (hatch studies)
      return ((hB, authBank), (hT, authTax),
              (hG, authG), (hG', authG'))

module Bank ( bank ) where
import SecLibTypes ; import SecLib
import Data

bank :: B -> (Hatch B L Account Status,
              Authority B) -> IO ()
bank = ...

module TaxOffice ( taxoffice ) where

import SecLibTypes ; import SecLib
import Data

taxoffice
 :: T -> (Hatch T L Citizen Address, Authority T)
    -> IO ()
taxoffice = ...

module Government ( government ) where
import SecLibTypes ; import SecLib
import Data

government
 :: G -> (Hatch G T Immigrant  Citizen,
    Authority G) -> (Hatch G B Study Result,
    Authority G) -> IO ()
government = ...


module Main ( main ) where
import Policies ; import Lattice
import Bank ; import TaxOffice ; import Government

main
  = do (whohB,  whohT, whohG, whohG')
         <- declassification
       bank B whohB
       taxoffice T whohT
       government G whohG whohG'
       return ()
```

Fig. 11. Skeleton for an application

formation is made public, banks cannot have access to information stored in the tax office and vice versa. Government, on the other hand, can have full access to the information stored at banks and the tax office, which can be debatable for any real government. However, we made such assumption to simplify the example and rather illustrate how functions `who` and `certify` can be used. We implement the security levels Government, Bank, Tax Office, and Public with the singleton types `G`, `B`, `T`, and `L`, respectively. The described security lattice is very simple and therefore we omit details about its implementation. We assume that the declassification polices are the followings: banks can declassify the status of their accounts (i.e. if an account is open or close), the tax office can release the address of the citizens, and the government can provide information about new immigrants to the tax office as well as revealing results of financial studies related to the economy of the country to the banks. Observe that, for instance, it is possible for the government to declassify some information to a bank, and then the bank divulges that information to the public by opening or closing some accounts. In order to avoid that, a more complex security lattice needs to be encoded. However, for simplicity, we tighten to the lattice in Figure 10. In Figure 11, we give the skeleton of an application that uses these security levels and the mentioned declassification policies. Module `Policies` declares declassification policies constructed by combinator `who`. Accounts, status of accounts, citizens, addresses, immigrants, financial studies, and outcomes of financial studies are represented by data types `Account`, `Status`, `Citizen`, `Address`, `Immigrant`, `Study`, and `Result`, respectively. Functions `status`, `address`, `immigrant`, and `study` have types `Account -> Status`, `Citizen -> Address`, `Immigrant -> Citizen`, and `Study -> Result`, respectively. These functions together with declarations of data types related to the application are placed in the module `Data`. Function `declassification` implements the declassification policies described before. Modules `Bank`, `TaxOffice`, and `Government` are untrusted and they might include malicious code. Functions `bank`, `taxoffice`, and `government` receive the escape hatches together with values of type `Authority s` for some corresponding instances of `s`. Observe that `bank`, `taxoffice`, and `government` expects to receive the constructor for security types `B`, `T`, and `G`, respectively. In other words, the authority for `bank`, `taxoffice`, and `government` is set to `B`, `T`, and `G`, respectively. Consequently, it is then possible for those functions to apply `cerfity` with escape hatches that release information at their authority level. Module `Main` sets the authority for each of the given functions while providing the corresponding escape hatches. Observe how constructors `B :: B`, `T :: T`, and `G :: G` are given to functions `bank`, `taxoffice`, and `government`, respectively. Malicious code placed in one function only compromises confidential information related to its authority's security level. For instance, if function `bank` contains malicious code, then confidential information related to the bank may be at risk. However, if `government` is compromised, all the information in the system may be affected. Function `government` should be carefully designed, or perhaps other restrictions regarding the application of the escape hatch must be imposed in this function (see next subsection). This phenomenon also occurs in `DLM` when a process running with the authority of all the principals in the system contains malicious code.

5.5 Combining dimensions

For some application, declassification policies are not so simple as those ones captured by the dimensions of what, when, and who. For those scenarios, the user of the library has basically two options. One one hand, the user can program his own policy, which provides enough flexibility. However, such flexibility could be dangerous when declassification policies are not implemented carefully. For instance, an escape hatch must not decide if declassification is possible by inspecting confidential data. Otherwise, attackers learn information about secrets when applying escape hatches by inspecting if the returned values are `Nothing` or not. On the other hand, users can specify more interesting declassification policies by combining applications of `ntimes`, `when`, and `who` together. For instance, we extend the *what*-policy from the example given in Section 5.2 to consider more dimensions as follows.

```
comb = do h <- ntimes 3
               (hatch (\(spwd,c) -> cypher spwd == c))
          (h', open, close) <- when h
          (h'', auth) <- who h'
          return (h'':: Hatch H L (Spwd, String) Bool,
                  open, close, auth)
```

Observe how `comb` defines an escape hatch that releases information if it is applied in a piece of code with authority `H` when some events that execute `open` happened and information has not been previously released more than three times. Other combinations are also possible. To the best of our knowledge, this is the first implementation of mechanisms to enforce more than one dimension for declassification.

6 Related work

Much previous related work addresses non-interference and functional languages consider reduced programming languages [HR98, VSI96, VS97] or requires designing compilers from scratch [PS02, Sim03]. Rather than implementing compilers, Li and Zdancewic [LZ06] show how to provide information flow security as a library for a real programming language. They provide an implementation for Haskell based on arrows combinators[Hug00], which naturally requires programmers to be familiar with arrows when writing security-related code. Their library still imposes restrictions on what kind of programs can be written. In particular, their approach does not generalize naturally in the presence of side-effects or information composed of data with different security levels. To incorporate these features, the library requires major changes as well as the introduction of new combinators [TRH07].

In this paper, we show that a less general notion, namely monads, is enough to provide information-flow security as a library. We propose a light-weight library ($\sim$ 400 LOC) able to handle side-effecting computations and that requires programmers to be familiar with monads rather than arrows. Moreover, by just placing data into corresponding `Sec s` monads, our library is also able to handle data composed of elements with different security levels. However, there exists one restriction in our approach w.r.t. to the arrow approach. Since our security levels are represented by types, all of them have to be known statically at compile-time[5], whereas in the arrow approach, they can be constructed at run-time.

Abadi et. al. developed the *dependency core calculus* (DCC) [ABHR99] based on a hierarchy of monads to guarantee non-interference. Similarly, `Sec` constructs a hierarchy of monads when applied to security levels `s`. However, DCC uses non-standard typing rules for its *bind* operations while our library just provides instances of the type class `Monad`. Tse and Zdancewic translate DCC to System F and show that non-interference can be stated using the *parametricity* theorem for F [TZ04]. They also provide an implementation in Haskell for a two-point lattice. Their implementation encodes each security level as an abstract data type constructed from functions and binding operations to compose computations with permitted flows. The same kind of ideas relies behind `Sec s`, `open`, and `close` (see Section 4). Their implementation requires, at most, $O(n^2)$ definitions for binders for n-points lattices. Since they consider the same non-standard features for binders as in DCC, they provide as many definitions for binders as different type of values produced after composing secure computations. Moreover, their implementation needs to be compiled with the GHC's flag `-fallow-undecidable-instances`. On one hand, our library requires, at most, $O(n^2)$ instantiations on the type class `Less` for n-points lattices, but it does not provide more than one definition for binders nor requires allowing undecidable instances in GHC [6]. DCC and Tse and Zdancewic's approaches do not consider computations with side-effects. Moreover, Tse and Zdancewic leaves as an open question how to encode more expressive policies, such as declassification, directly in the type system of Haskell.

Harrison and Hook show how to implement an abstract operating system called *separation kernel* [HH05]. Programs running under this multi-threading operating system are non-interferent. To achieve that, the authors rely on properties related to monad transformers as well as state and resumption monads. Basically, each thread is represented as an state monad that have access to the locations related to the thread's security level while state monad transformers act as parallel composition. Interleaving and communication between threads is carried out by plugging a resumption monads on top of the parallel composition of all the threads in the system. Non-interference is then enforced by the scheduler implementation, which only allow signaling threads at the same, or higher, security level as the thread that issued the signal. Different from that, our library enforces non-interference by typing. The authors also use monads differently than we do since their goals are constructing secure kernels rather than providing information-flow security as a library. For instance, we do not use state monads, state transformers, or resumption monads since we do not model threads. As a result, our library is simpler and more suitable to write sequential programs in Haskell. It is stated as a future work how to extend our library to include concurrency.

Crary et. al. design a monadic calculus for non-interference for programs with mutable state[CKP03]. Their language distinguishes between *term* and *expressions*, where terms

[5] We are investigating the use of polymorphic recursion to alleviate this – this remains future work however.

[6] All the code shown in the paper works with the Glasgow Haskell Compiler (GHC) with the flag `-fglasgow-exts`

are pure and expressions are (possibly) effectful computations. The calculus mainly tracks flow of information by inspecting the security levels of effects produced by expressions. Expressions can be included at the term level as an element of the monadic type $\bigcirc_{(r,w)}A$, which denotes a suspended computation where the security level r is an upper bound on the security levels of the store locations that the suspended computation reads, while w is a lower bound on the security level of the store locations to which it writes. Authors introduce the notion of *informativeness* in order to relax some typing rules so that reading and writing into secret store locations can be included in large computations related to public data. A type A is informative at security level r or above if its values can be used or observed by computations that may read data from security level r or above. In our library, the type `SecIO s a` makes the value of type `a` only informative at security level `s`. In principle, the value of type `a` cannot be used anywhere but inside the monad `SecIO s`. Considering a two-point lattice, we introduce the function `plug :: Less L H => SecIO H a -> SecIO L (Sec H a)` to allow reading and writing secret files into computations related to public data. Observe that the function preserves the informativeness of `a` by placing it inside of the monad `Sec H`.
Recently, several approaches have been proposed to dynamically enforce non-interference [GBJS06, SST07, NSCT07]. In order to be sound, these approaches still need to perform some static analysis prior to or at run-time. Authors argue, in one way or another, that their methods are more precise than just applying an static analysis to the whole program. For instance, if there is an insecure piece of dead code in a program, most of the static analysis techniques will reject that program while some of their approaches will not. The reason for that relies in the fact that dead code is generally not executed and therefore not analyzed by dynamic enforcement mechanisms. Our library also combines static and dynamic techniques but in a different way. Non-interference is statically enforced through type-checking while run-time mechanisms are introduced for declassification. By dynamically enforcing declassification policies, we are able to modularly extend the part of the library that enforce non-interference to add downgrading of information and being able to enforce several dimensions for declassification in a flexible and simple manner. To the best of our knowledge, this is the first implementation of declassification policies that are enforced at run-time and the first implementation that allows combining dimensions for declassifications.

7 Conclusions

We have presented a light-weight library for information-flow security in Haskell. Based on specially designed monads, the library guarantees that well-typed programs are non-interferent; i.e. secret data is not leaked into public channels. When intended release of information is required, the library also provides novel means to specify declassification policies, which comes from the fact that policies are dynamically enforced and it is possible to construct complex policies from simple ones in a compositional manner. Taking ideas from the literature, we show examples of declassification policies related to what, when, and by whom information is released. The implementation of the library and the examples described in this paper are publicly available in [RCH08a]. The well-known concept of monads together with the light-weight and flexible characteristic of our approach makes the library suitable to build Haskell applications where confidentiality of data is an issue.

Acknowledgements We wish to thank to Aslan Askarov, Ulf Norell, Andrei Sabelfeld, David Sands, and Josef Svenningsson for useful comments and discussions about this work. 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

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

[AS07] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

[BS06] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In Peter Sestoft, editor, *Proc. European Symp. on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

[CHM02] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[CKP03] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state, 2003.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

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

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

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[GBJS06] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer-Verlag, December 2006.

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

[HH05] W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 16–30, Washington, DC, USA, 2005. IEEE Computer Society.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

[Hug00] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[Jac96] M. H. Jackson. Linux shadow password howto. Available at `http://tldp.org/HOWTO/Shadow-Password-HOWTO.html`, 1996.

[LLMS00] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, New York, NY, USA, 2000. ACM.

[Loc08] Local Root Exploit. Linux kernel 2.6 local root exploit. Available at `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=465246`, February 2008.

[LZ06] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[LZ07] P. Li and S. Zdancewic. Arrows for secure information flow. Available at `http://www.seas.upenn.edu/∼lipeng/homepage/lz06tcs.pdf`, 2007.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MR07] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In Rocco De Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 141–156. Springer, 2007.

[NS05] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.

[NSCT07] S. K. Nair, P. N.D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, September 2007.

[Pie04] B. C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004.

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[RCH08a] A. Russo, K. Claessen, and J. Hughes. Lightweight information-flow security in Haskell. Software release. Available at `http://www.cs.chalmers.se/∼russo/seclib.htm`, 2008.

[RCH08b] A. Russo, K. Claessen, and J. Hughes. Lightweight information-flow security in Haskell. Technical Report. Chalmers University of Technology. To appear., September 2008.

[Sim03] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE Computer Society, 2005.

[SST07] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 203–217, 2007.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

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

[TZ04] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 115–125, New York, NY, USA, 2004. ACM.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

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

[Wad92] P. Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, August 1992.

$$
\begin{aligned}
e \ &::= \texttt{tt} \mid \texttt{ff} \mid \texttt{case}\ e\ e\ e \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i\ e \mid \texttt{fix}\ e \\
&\quad\mid \texttt{return}_\ell\ e \mid e\ \star_\ell\ e \mid \texttt{up}_{\ell,\ell'}\ e \mid \texttt{public}_\ell\ e \mid \texttt{S}_\ell\ e \mid \bullet \\
v \ &::= \texttt{tt} \mid \texttt{ff} \mid x \mid \lambda x.e \mid (v,v) \mid \texttt{fix}\ e \mid \texttt{S}_\ell\ e \\
\tau \ &::= \texttt{Bool} \mid \tau \to \tau \mid (\tau,\tau) \mid \texttt{Sec}_\ell\ \tau
\end{aligned}
$$

Fig. 12. Terms, values, and types

$$
\begin{aligned}
E \ &::= [] \mid \texttt{case}\ E\ e\ e \mid E\ e \mid (E,e) \mid (e,E) \mid \pi_i\ E \\
&\quad\mid \texttt{return}_\ell\ E \mid E\ \star_\ell\ e \mid \texttt{up}_{\ell,\ell'}\ E \mid \texttt{public}_\ell\ E
\end{aligned}
$$

$$
\begin{array}{ll}
E[\texttt{case}\ \texttt{tt}\ e_1\ e_2] \longrightarrow E[e_1] & E[\texttt{return}_\ell\ e] \longrightarrow E[\texttt{S}_\ell\ e] \\
E[\texttt{case}\ \texttt{ff}\ e_1\ e_2] \longrightarrow E[e_2] & E[(\texttt{S}_\ell\ e_1)\ \star_\ell\ e_2] \longrightarrow E[([e_1/x]e_2)\ \star_\ell\ \lambda x.\texttt{return}_\ell\ x] \\
E[(\lambda x.e_1)\ e_2] \longrightarrow E[[e_2/x]e_1] & E[(\texttt{S}_\ell\ e)\ \star_\ell\ \lambda x.\texttt{return}_\ell\ x] \longrightarrow (\texttt{S}_\ell\ e) \\
E[\pi_i\ (e_1,e_2)] \longrightarrow E[e_i] & E[\texttt{up}_{\ell,\ell'}\ (\texttt{S}_\ell\ e)] \longrightarrow E[\texttt{S}_{\ell'}\ e] \\
E[\texttt{fix}\ e] \longrightarrow E[e\ (\texttt{fix}\ e)] & E[\texttt{public}_{\ell'}\ (\texttt{S}_\ell\ e)] \longrightarrow E[e]
\end{array}
$$

Fig. 13. Operational semantics

Soundness

This section formalizes the non-interference guarantee obtained by our approach. Due to lack of space, we only include details regarding the core, purely functional, part of the library. Further details can be found in [RCH08b].

To illustrate our approach, we take a simple call-by-need λ-calculus extended with boolean values, pairs, recursion, and monadic operations, as shown in Figure 12. Syntactic categories e, v, and τ represent terms, values, and types, respectively. Monadic terms, values, and types are decorated with security levels (ℓ) in order to make type-checking easier. Three special syntax nodes are added to the language: $\texttt{public}_\ell$, $\texttt{S}_\ell\ e$, and $\bullet$, . Node $\texttt{S}_\ell\ e$ denotes the run-time representation of a monadic value and it does not appear in programs. Node $\texttt{public}_\ell$ denotes an operation that allows to extract results of monadic computations. Node $\bullet$ represents term erasure, which is explained later. The operational semantics of the language is formalized in Figure 13 and it is self explanatory.

Typing rules related to non-monadic terms are standard and therefore we omit them. Erased terms $\bullet$ are associated to any type ($\Gamma \vdash \bullet : \tau$). Typing rules for monadic operations are given in Figure 14. The first three typing rules are self explanoty. However, typing rule for primitive $\texttt{public}_\ell$ deserves some explanation. Predicate $\mathrm{Attacker}$ defines the attacker model. Given a security level ℓ, predicate $\mathit{Attacker}(\ell)$ holds iff ℓ is the highest security level where attackers can observe data, which we consider as unique. For instance, the attacker described in Section 3.3 can only observe public data,

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{return}_\ell\, e : \mathtt{Sec}_\ell\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Sec}_\ell\, \tau \qquad \Gamma \vdash e_2 : \tau \to \mathtt{Sec}_\ell\, \tau'}{\Gamma \vdash e_1 \star_\ell e_2 : \mathtt{Sec}_\ell\, \tau'}$$

$$\frac{\Gamma \vdash e : \mathtt{Sec}_\ell\, \tau}{\Gamma \vdash \mathtt{up}_{\ell,\ell'}\, e : \mathtt{Sec}_{\ell'}\, \tau}\ (\ell \sqsubseteq \ell')$$

$$\frac{\Gamma \vdash e : \mathtt{Sec}_{\ell'}\, \tau \qquad \mathit{Attacker}(\ell)}{\Gamma \vdash \mathtt{public}_\ell\, e : \tau}\ (\ell' \sqsubseteq \ell)$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{S}_\ell\, e : \mathtt{Sec}_\ell\, \tau}$$

Fig. 14. Typing rules for monadic operations and values

i.e. $\mathit{Attacker}(\mathtt{L})$ holds for the two-point lattice. If the attacker can observe data at security level ℓ, then the attacker can extract such data from a monadic computation of type $\mathtt{Sec}_{\ell'}\ \tau$, where $\ell' \sqsubseteq \ell$. Later, we will show that well-typed programs preserve confidentiality of data at higher security levels than ℓ.

Similar to [LZ07], we formalize the non-interference property by using the technique of term erasing. Intuitively, data at security levels where the attacker cannot observe information can be safely rewritten to the syntax node $\bullet$. Function ε_ℓ is responsible to perform such rewritten for data at security level not lower than ℓ, as shown in Figure 15. This function is also defined for contexts E as an homomorphism (e.g. $\varepsilon_\ell([]) = []$, $\varepsilon_\ell(\mathtt{case}\ E\ e_1\ e_2) = \mathtt{case}\ \varepsilon_\ell(E)\ \varepsilon_\ell(e_1)\ \varepsilon_\ell(e_2)$, etc). We define a relation $(\Longrightarrow)$ for erased terms as follows.

$$\frac{e_1 \longrightarrow e_2}{e_1 \Longrightarrow \varepsilon_\ell(e_2)}$$

Erased terms evaluate in the same way as terms, except that, after one evaluation step, the resulting term is erased again. The relation guarantees that "confidential" data is erased as soon as it is created.

To formalize a non-interference-like result, we establish a simulation between terms $(\longrightarrow)$ and erased terms $(\Longrightarrow)$. The following lemmas establish the simulation for one and multiple steps. We write $\longrightarrow^*$ and $\Longrightarrow^*$ for the reflexive and transitive clousure of $\longrightarrow$ and $\Longrightarrow$, respectively.

Lemma 1 (Single step-simulation). *If $\Gamma \vdash e_1 : \tau$, $\mathit{Attacker}(\ell)$ holds, and $e_1 \longrightarrow e_2$, then $\Gamma \vdash e_2 : \tau$ and $\varepsilon_\ell(e_1) \Longrightarrow^* \varepsilon_\ell(e_2)$.*

Lemma 2 (Multiple step-simulation). *If $\Gamma \vdash e_1 : \tau$, $\mathit{Attacker}(\ell)$ holds, and $e_1 \longrightarrow^* e_2$, then $\Gamma \vdash e_2 : \tau$ and $\varepsilon_\ell(e_1) \Longrightarrow^* \varepsilon_\ell(e_2)$.*

Once established the simulation between $(\longrightarrow)$ and $(\Longrightarrow)$, we can proceed to formalize the security guarantee. The following theorem gives a non-interference-like property.

$$
\begin{array}{ll}
\varepsilon_\ell(\mathtt{tt}) & = \mathtt{tt} \\
\varepsilon_\ell(\mathtt{ff}) & = \mathtt{ff} \\
\varepsilon_\ell(\mathtt{case}\; e_1\; e_2\; e_3) & = \mathtt{case}\; \varepsilon_\ell(e_1)\; \varepsilon_\ell(e_2)\; \varepsilon_\ell(e_3) \\
\varepsilon_\ell(x) & = x \\
\varepsilon_\ell(\lambda x.e) & = \lambda x.\varepsilon_\ell(e) \\
\varepsilon_\ell(e_1\; e_2) & = \varepsilon_\ell(e_1)\; \varepsilon_\ell(e_2) \\
\varepsilon_\ell((e_1, e_2)) & = (\varepsilon_\ell(e_1), \varepsilon_\ell(e_2)) \\
\varepsilon_\ell(\pi_i\; e) & = \pi_i\; \varepsilon_\ell(e) \\
\varepsilon_\ell(\mathtt{fix}\; e) & = \mathtt{fix}\; \varepsilon_\ell(e) \\
\varepsilon_\ell(\mathtt{return}_{\ell'}\; e) & = \begin{cases} \mathtt{return}_{\ell'}\; (\varepsilon_\ell(e)) & , \ell' \sqsubseteq \ell \\ \bullet & , \text{otherwise} \end{cases} \\
\varepsilon_\ell(e_1 \star_{\ell'} e_2) & = \begin{cases} (\varepsilon_\ell(e_1)) \star_{\ell'} (\varepsilon_\ell(e_2)) & , \ell' \sqsubseteq \ell \\ \bullet & , \text{otherwise} \end{cases} \\
\varepsilon_\ell(\mathtt{up}_{\ell',\ell''}\; e) & = \begin{cases} \mathtt{up}_{\ell',\ell''}\; (\varepsilon_\ell(e)) & , \ell'' \sqsubseteq \ell \\ \bullet & , \text{otherwise} \end{cases} \\
\varepsilon_\ell(\mathtt{public}_{\ell'}\; e) & = \begin{cases} \mathtt{public}_{\ell'}\; (\varepsilon_\ell(e)) & , \ell' \sqsubseteq \ell \\ \bullet & , \text{otherwise} \end{cases} \\
\varepsilon_\ell(\mathtt{S}_{\ell'}\; e) & = \begin{cases} \mathtt{S}_{\ell'}\; (\varepsilon_\ell(e)) & , \ell' \sqsubseteq \ell \\ \bullet & , \text{otherwise} \end{cases} \\
\varepsilon_\ell(\bullet) & = \bullet
\end{array}
$$

Fig. 15. Definition of the erasure function

Theorem 1. *Given security levels ℓ, ℓ' and function e such that it is not the case $\ell' \sqsubseteq \ell$, $Attacker(\ell)$ holds, e does not include $\bullet$ or constructor $\mathtt{S}_{\ell''}$ for any ℓ'', and $\Gamma \vdash e : \mathtt{Sec}_{\ell'}\; \tau \to \mathtt{Bool}$, then*

$$
\begin{aligned}
\forall e_1 e_2 . (\Gamma \vdash e_i : \tau)_{i=1,2} \;\wedge\; & e\; (\mathtt{return}_{\ell'}\; e_1) \longrightarrow^* v_1 \;\wedge \\
& e\; (\mathtt{return}_{\ell'}\; e_2) \longrightarrow^* v_2 \Rightarrow v_1 = v_2
\end{aligned}
$$

The theorem states that given two pieces of data that are "secret", e.g. not observable by the attacker, the result of the computation must not be affected by any of them. Therefore, no leaks are produced. The proof easily follows from Lemma 2 and the fact that $\longrightarrow$ and $\Longrightarrow$ are deterministic.