Thesis for the Degree of Licentiate of Engineering

Controlling Timing Channels in Multithreaded Programs

Alejandro Russo

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2007

Controlling Timing Channels in Multithreaded Programs
Alejandro Russo

Technical Report no. 38L
ISSN 1652–876X
Department of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, 2007

Abstract

"The solution lies in secrecy," said Medra. "But so does the problem."
III.Tern, Tales from Earthsea, Ursula K. Le Guin

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. Volpano and Smith 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.

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. While this approach allows the implementation of a generalized version of `protect`, it 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 `protect` or interactions with 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.

To evaluate some of the ideas described above, the thesis presents an *implementation* in Haskell of a library that provides information-flow security for multithreaded code. The 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 guarantees information-flow security in multithreaded programs and the first implementation of a case study that involves concurrency and information-flow policies.

Contents

CHAPTER

1

Introduction

Computer systems are nowadays connected by networks. For instance, computers and mobile phones are often connected to Internet and cellular networks, respectively. These systems frequently send, receive, and store confidential information in order to perform some tasks requested by users. It is common to provide credit card numbers when performing online shopping or to store private multimedia contents in mobile phones. Besides the personal use of such systems, there is a tendency to rely more and more on computers in order to perform legal procedures, e.g. calculation of salaries, taxes, or pensions rates. Those legal procedures are commonly carried out by governments or subcontractors. Thus, preserving the confidentiality of data about citizens becomes an important requirement in order to guarantee privacy rights in any democratic government.

Sharing or accessing information over networks provides clear benefits to users, and Internet is a clear example of that. Unfortunately, connecting computers to networks also exposes them to attacks. One clear example of that is the malicious code placed on the web. Users download software from Internet without any guarantee that their confidential data are not sent over the network while running those programs. Since mobile phones can be seen as computers with reduced computational power, they are also victim of attackers. As before, users download or exchange, via Bluetooth, ring tones, games, or software. Likewise, there are still no guarantees that the confidentiality of data is preserved by 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 by software developers, 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.

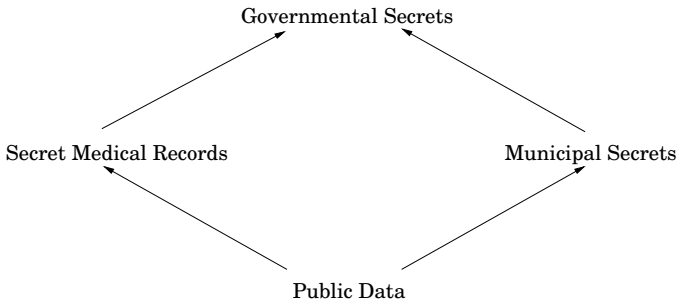

Fig. 1. Security lattice

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 in granted.
Information-flow policies can be formalized by attaching security labels to computational entities and data in the system, and defining how the information can flow between different security levels. For instance, it possible to define the following information-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_2 can flow into entities of security level l_1. 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 possitions 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].

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]. The information-flow analysis is typically performed by static program analysis. As a consequence, it is possible to guarantee *end-to-end* securities, as confidentiality, by just analyzing the whole code of a given system.

Confidentiality policies could be precisely characterized by using program semantics. Moreover, they can be provably enforced by traditional mechanisms as type systems. *Noninterference* is a well known end-to-end property of programs that expresses the freeness of flows from more secret security levels to less secret ones. In other words, a variation in the confidential input of a program does not produce any variation of its public outputs. The attacker model defines what the attacker can observe about the execution of programs. For the noninterference 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 respectively consists on secret and public variables initialized with some values. If a program terminates, it does 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 $[\![C]\!] : S \rightarrow S \cup \{\bot\}$ 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 noninterference 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. For that reason, it is classified 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 noninterference 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 values. 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* noninterference properties.

Types of Flows

Language-based information-flow techniques deal with mechanisms used by programming languages to convey information. These mechanisms include assignments and branching instructions. Confidentiality of data can be preserved if programs are free of illegal *explicit and implict* 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, can use control constructs in the language to leak information. As an example, the program

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

leaks if $h > 0$ or not by using the construct $\texttt{if} - \texttt{then} - \texttt{else}$. Even though there is no direct assignment of secret values to public variables, the final value of l depends on the secret value h.

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 kind of mechanism are referred as *covert channels*[Lam73]. For example, the execution time of a program, memory consumption, and concurrency features can be used to leak confidential information. This thesis proposes techniques to deal with *covert channels* introduced by some concurrent features. More precisely, 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. This covert channel is called *internal timing covert channel* [VS99] and is the main focus of this presentation.

Thesis overview

The thesis takes a language-based approach to information-flow enforcement for concurrent programs. In this section, we briefly outline the contents of the four chapters.

Securing Interaction between Threads and the Scheduler Existing approaches to specifying and enforcing 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, `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 `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 noninterference-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. The proposed techniques relies on the modification of the scheduler in the run-time environment.
This chapter is an extended version of the paper accepted to the 19th IEEE Computer Security Foundations Workshop, Venice, Italy, July 5-7, 2006.

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.

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.

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 guarantee 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.

References

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

[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.

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

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

[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.

[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.

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

CHAPTER

2

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.

Securing Interaction between Threads and the Scheduler

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

- *Permissiveness* The presence of multithreading enables new attacks which are not possible for sequential programs. The challenge is to reject these attacks without compromising the permissiveness of the model. In other words, information flow models should accept as many intuitively secure and useful programs as possible.
- *Scheduler-independence* The security of a given program should not critically depend on a particular scheduler [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.
- *Standard 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 non-standard primitives that accommodate security, they should be clearly and securely implementable.
- *Language expressiveness* A key to a practical security model is an expressive underlying language. In particular, the language should be able to treat dynamic thread creation, as well as provide possibilities for synchronization.
- *Practical enforcement* Another practical key is a tractable security enforcement mechanism. Particularly attractive is compile-time automatic *compositional* analysis. Such an analysis should nevertheless be *permissive*, striving to trade as little expressiveness and efficiency for security as possible.

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

2 Motivation and background

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

2.1 Leaks via scheduler

Assume a partition of variables into high (secret) and low (public). Suppose h and l are a high and a low variable, respectively. Intuitively, information flow in a program is secure (or satisfies *noninterference* [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:

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

These threads are clearly secure in isolation because 1 is always the outcome for l in d_1, and 0 is always the outcome for l in d_2. However, when d_1 and d_2 are executed in parallel, the security of the threadpool is no longer guaranteed. In fact, the program will leak whether the initial value of h was positive into l under many reasonable schedulers.
We observe that program $c_1 \parallel c_2$ can be straightforwardly secured by synchronization. Assuming the underlying language features locks, we can rewrite the program as

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

The lock primitives ensure that the undesired interleaving of c_1 and c_2 is prevented.
Unfortunately, synchronization primitives offer no general solution. The source of the leak in program $d_1 \parallel d_2$ is *internal timing* [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 live thread can be scheduled with the same probability. Volpano and Smith introduce a special primitive in order to help protecting against internal timing leaks. This primitive is called `protect`, and it can be applied to any command that contains no loops. A protected command $\texttt{protect}(c)$ is executed atomically, *by definition* of its semantics. Such a primitive can be used to secure program $d_1 \parallel d_2$ as:

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

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

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

Unfortunately, such an implementation is insecure. The somewhat subtle reason is that when the execution is inside of the `if` statement, the other threads do not become *instantly locked*. Thread d_2 can still be scheduled, which could result in blocking and updating the wait list for the lock with d_2.
For simplicity, assume that $\texttt{sleep}(\texttt{n})$ 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 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 sen-

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

Fig. 1. Command syntax

sitive 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 [Aga00, SS00] might arbitrarily change the efficiency of the resulting code (and possibly result in a diverging program). In the present work, we assume a weaker attacker and aim for a more permissive security condition and analysis.
External timing-sensitive security has been extended to languages with semaphores primitives [Sab01] and message passing [SM02].

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 non-compositional because the parallel composition with a thread assigning to low variables is not generally secure.
Most 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.

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.

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

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

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

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

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

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

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

Fig. 2. Semantics for commands

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 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 $\left\{\bullet\!\rightsquigarrow, \rightsquigarrow\!\bullet, \circ_{\vec{d}}, \bullet_{\vec{d}}\right\}$, where $\vec{d}$ is a set of threads. The sequential composition rule propagates events to the top level. We describe the meaning of the events in conjunction with the rules that involve the events.

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

The hiding command `hide` moves the current thread from the low to the high part of the threadpool. This is expressed in the semantics by event $\rightsquigarrow\!\bullet$ which communicates to the scheduler to treat the thread as high. The unhiding command `unhide` has the dual effect: it communicates to the scheduler by event $\bullet\!\rightsquigarrow$ that the thread should be treated as low. We define independent commands `hide` and `unhide` instead of forcing them to wrap code blocks syntactically (cf. `protect`). We expect this choice to be useful when adding exceptions to the language. For example, an `unhide` in an exception handler may refer to several `hide` primitives under a `try` statement.

Commands $\texttt{fork}(c, \vec{d})$ and $\texttt{hfork}(c, \vec{d})$ dynamically spawn a collection of threads $\vec{d}$ while the current thread runs command c. The difference between the two primitives is in the generated event. Command `fork` signals about the creation of low threads with

event $\circ_{\vec{d}}$ (where $\circ$ is read "low") while `hfork` indicates that new threads should be treated as high by event $\bullet_{\vec{d}}$ (where $\bullet$ is read "high").

3.2 Semantics for schedulers

Figure 3 depicts the semantic rules that describe the behavior of the scheduler. a scheduler is, generally, a program σ that forms a *scheduler configuration* $\langle\sigma, \nu\rangle$ together with a memory ν. We assume that the scheduler memory is disjoint from the program memory. The scheduler memory contains variable q that regulates for how many steps a thread can be scheduled. Live threads are tracked by variable t that consists of low and high parts. the low part is named by $t_\circ$, while the high part is composed of two subpools named $t_\bullet$ and t_e. Threads in $t_\bullet$ are always high, but threads in t_e were low in the past, are high at present, and might eventually be low in the future. Threads are moved back and forth from $t_\circ$ to t_e by executing the hiding and unhiding commands. Variable r represents the running thread. Variable s regulates whether low threads may be scheduled. When s is $\circ$, both low and high threads may be scheduled. However, when s is $\bullet$, only high threads may be scheduled, preventing low threads from observing internal timing information about high threads. In addition, the scheduler might have some internal variables.

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

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

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

Events $r\rightsquigarrow\bullet$ and $\bullet\rightsquigarrow r$ are triggered by the `hide` and `unhide` commands, respectively. The scheduler handles event $r\rightsquigarrow\bullet$ by moving the current thread from the low to the

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

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

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

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

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

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

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

Fig. 3. Semantics for schedulers

high part of the threadpool and setting s' to $\bullet$. Upon event $\bullet\rightsquigarrow r$, the scheduler moves the thread back to the low part of the threadpool, setting s' to $\circ$.
Events $r\rightsquigarrow\bullet\times$ and $\bullet\rightsquigarrow r\times$ are triggered by `hide` and `unhide`, respectively, when they are the last commands to be executed by a thread.

3.3 Semantics for threadpools

The interaction between threads and the scheduler takes place at the top level, the level of *threadpool configurations*. These configurations have the form $\langle \vec{c}, m, \sigma, \nu\rangle \xrightarrow{\alpha} \langle \vec{c'}, m', \sigma', \nu'\rangle$ where α ranges over the same set of events as in the semantics for schedulers.
The semantics for threadpool configurations is displayed in Figure 4. The dynamic thread creation rule is triggered when the running thread c_r generates a thread creation event $\alpha_{\vec{d}}$ where α is either $\bullet$ or $\circ$. This event is synchronized with scheduler event $\alpha^r_{\vec{d}}$ that requests the scheduler to handle the new threads depending on whether α is high or low.
If c_r does not spawn new threads or terminate, then its command rule is synchronized with scheduler event $r\rightsquigarrow$. If c_r terminates in a transition without labels, then scheduler

$$\frac{\langle c_r, m\rangle \xrightarrow{\alpha\,\vec{d}} \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \xrightarrow{\alpha\,\vec{d}}{}^{r} \langle \sigma', \nu'\rangle \qquad \alpha \in \{\bullet, \circ\}}{\langle c_1 \ldots c_n, m, \sigma, \nu\rangle \xrightarrow{\alpha\,\vec{d}}{}^{r} \langle c_1 \ldots c_{r-1} c_r' \vec{d} c_{r+1} \ldots c_n, m', \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \rightharpoonup \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \xrightarrow{r\rightsquigarrow} \langle \sigma', \nu'\rangle}{\langle c_1 \ldots c_n, m, \sigma, \nu\rangle \xrightarrow{r\rightsquigarrow} \langle c_1 \ldots c_{r-1} c_r' c_{r+1} \ldots c_n, m', \sigma', \nu'\rangle}$$

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

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

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

$$\frac{\langle \sigma, \nu\rangle \xrightarrow{\uparrow_\alpha r'} \langle \sigma', \nu'\rangle \qquad \alpha \in \{\circ, \bullet\}, r' \in \{1, \ldots, n\}}{\langle c_1 \ldots c_n, m, \sigma, \nu\rangle \xrightarrow{\uparrow_\alpha r'} \langle c_1 \ldots c_n, m, \sigma', \nu'\rangle}$$

$$\frac{\langle c_r, m\rangle \xrightarrow{\alpha} \langle c_r', m'\rangle \qquad \langle \sigma, \nu\rangle \xrightarrow{\alpha} \langle \sigma', \nu'\rangle \qquad \alpha \in \{r \rightsquigarrow \bullet, \bullet \rightsquigarrow r\}}{\langle c_1 \ldots c_n, m, \sigma, \nu\rangle \xrightarrow{\alpha} \langle c_1 \ldots c_{r-1} c_r' c_{r+1} \ldots c_n, m', \sigma', \nu'\rangle}$$

Fig. 4. Semantics for threadpools

event $r \rightsquigarrow \times$ is required for synchronization in order to update the threadpool information in the scheduler memory. If c_r terminates with $\rightsquigarrow\bullet$ (resp., $\bullet\rightsquigarrow$) then synchronization with $r \rightsquigarrow \bullet\times$ (resp., $\bullet\rightsquigarrow r\times$) is required to record both termination and hiding (resp., unhiding).

Scheduler event $\uparrow_\alpha r'$ triggers a selection of a new thread r' without affecting the commands in the threadpool or their memory. Finally, entering and exiting the high part of the threadpool is performed by synchronizing the current thread and the scheduler on events $r\rightsquigarrow\bullet$ and $\bullet\rightsquigarrow r$.

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

3.4 On multi-level extensions

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

In the presence of a general security lattice, the threadpool is partitioned into as many parts as the number of security levels. Commands $\texttt{hide}_\ell$, $\texttt{unhide}_\ell$, 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 need to ensure that no threads from ℓ'-threadpools, for all ℓ' such that $\ell \not\sqsubseteq \ell'$ may execute until the hidden thread reaches $\texttt{unhide}_\ell$. Naturally, command $\texttt{fork}_\ell$ creates threads in ℓ-threadpool.

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

4 Security specification

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

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

Scheduler-related events have different distinguishability levels. Events $\circ^r_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, $\uparrow_\circ r'$, $r\rightsquigarrow\bullet$, $\bullet\rightsquigarrow r$, $r\rightsquigarrow\bullet\times$, and $\bullet\rightsquigarrow r\times$ (where r and r' are low threads) operate on low threads and are therefore low events. On the other hand, events $\bullet^r_{\vec{d}}$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, and $\uparrow_\bullet r'$ (where r and r' are high threads) are high.

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

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

- *if $\langle\sigma_i, \nu_i\rangle \xrightarrow{\alpha} \langle\sigma'_i, \nu'_i\rangle$ 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 $\langle\sigma'_i, \nu'_i\rangle\ R\ \langle\sigma'_{3-i}, \nu'_{3-i}\rangle$.*

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` $(\texttt{True})$ `do` $\{$
$q := M; run(r);$
`while` $(q > 0)$ `do` $\{$
`receive`
$\circ^{r}_{\vec{d}}:$ $\quad t_\circ := \texttt{append}(t_\circ, N(\vec{d}));$
$\bullet^{r}_{\vec{d}}:$ $\quad t_\bullet := \texttt{append}(t_\bullet, N(\vec{d}));$
$r \rightsquigarrow:$ $\quad$ `skip`;
$r \rightsquigarrow \times:$ $\quad t_\circ := \texttt{remove}(r, t_\circ); t_\bullet := \texttt{remove}(r, t_\bullet);$
$q := 0;$
$r \rightsquigarrow \bullet:$ $\quad t_\circ := \texttt{remove}(r, t_\circ); t_\bullet := \texttt{remove}(r, t_\bullet);$
$t_\bullet := \texttt{append}(t_\bullet, [r]); s := 1;$
$\bullet \rightsquigarrow r:$ $\quad t_\circ := \texttt{append}(t_\circ, [r]);$
$t_\bullet := \texttt{remove}(r, t_\bullet); s := 0; q := 0;$
$r \rightsquigarrow \bullet \times:$ $\quad t_\circ := \texttt{remove}(r, t_\circ); t_\bullet := \texttt{remove}(r, t_\bullet);$
$s := 1; q := 0;$
$\bullet \rightsquigarrow r \times:$ $\quad t_\circ := \texttt{remove}(r, t_\circ); t_\bullet := \texttt{remove}(r, t_\bullet);$
$s := 0; q := 0;$
`end receive`;
$q := q - 1$
$\};$
$turn := (turn + 1)\%2;$
`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. 5. 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 5 displays an example of a scheduler in pseudocode. This is a round-robin scheduler that keeps track of two lists of threads: low and high ones. The scheduler interchangeably chooses between threads from these two lists, when possible. It waits for events generated by the running thread (expressed by primitive `receive`). Functions `head`, `tail`, `remove`, and `append` have the standard semantics for list operations. Operation $N(\vec{d})$, variables $t_\circ$, $t_\bullet$, s, r, and q have the same purpose as described in Section 3.2. Constant M is a positive natural number. Variable $turn$ encodes the interchangeable choices between low and high threads. Function $run(r)$ launches the execution of thread r. It is not difficult to show that this schedulers complies to the assumptions from Section 3.2, and that it is noninterferent.

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

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

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

A form of scheduler independence is built in the definition by the universal quantification over all noninterferent schedulers. Although the universally quantified condition may appear difficult to guarantee, we will show that the security type system from Section 5 ensures that any typable program is secure. Note that this security definition is *termination-insensitive* [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 systems

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

5.1 Typing rules

Figure 6 displays the typing rules for expressions and commands. Suppose Γ is a *typing environment* which includes security type information for variables (whether they are low or high) and two variables, *pc* and *hc*, ranging over security types (low or $high$). By convention, we write Γ_v for Γ restricted to all variables *but* v.
Expression typing judgments have the form $\Gamma \vdash e : \tau$ where τ is low only if all variables in e (denoted *FV*(e)) are low. If there exists a high variable that occurs in e then τ must be $high$. Expression types make no use of type variables *pc* and *hc*.
Command typing judgments have the form $\Gamma \vdash c : \tau$. As a starting point, let us see how the rules track sequential-style information flow. The assignment rule ensures that information cannot leak *explicitly* by assigning an expression that contains high variables into a low variable. Further, *implicit* flows are prevented by the program counter mechanism [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.

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

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

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

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

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

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

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

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

Fig. 6. 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 track information flow specific to the multithreaded setting. The main job of these variables is to record whether the current thread is in the high part of the threadpool ($hc = high$) or is in the low part ($hc = low$). Command type τ reflects the level of the hiding context after the command execution.

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

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

By removing the typing rules for `hide`, `unhide`, `hfork`, and the security type variables *hc* and τ from Figure 6, we obtain a standard type system for securing information flow in sequential programs (cf. [VSI96]). This illustrates that our type provides a general technique for modular extension of systems that track information flow in a sequential setting.

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

5.2 Soundness

We enlist some helpful lemmas for proving the soundness of the type system. The first lemma states that high control commands must be typed with high *hc*.

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

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

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* $\Gamma_{\mathit{hc},\mathit{pc}}, \mathit{pc} \mapsto \tau_c, \mathit{hc} \mapsto high \vdash c : low$ *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* $\Gamma_{\mathit{hc},\mathit{pc}}, \mathit{pc} \mapsto \tau_c, \mathit{hc} \mapsto high \vdash c' : low$*.*

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* $\Gamma_{\mathit{hc}}, \mathit{hc} \mapsto low \vdash c : low$*, 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 a `hide` instruction is not executed.

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

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. *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:*

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

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

Corollary 1 (Soundness). *If* Γ_{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 are as follows:

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

The semantics for commands also need 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 need to react to such an event by reseting counter q' to 0:

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

We need to ensure that the only possibility to schedule another thread is by generating event $\not\rightsquigarrow$. Hence, we add premise $q' = \infty$ to the semantics rules for schedulers that handle events $\uparrow_\bullet r'$ and $\uparrow_\circ r'$. Additionally, the last rule in Figure 4 now allows α to range over $\{r \rightsquigarrow \bullet, \bullet \rightsquigarrow r, r \not\rightsquigarrow\}$, which propagates yielding events $\not\rightsquigarrow$ from threads to the scheduler. Similar to scheduler events $r \rightsquigarrow \bullet \times$ and $\bullet \rightsquigarrow r \times$, a new transition is added to the threadpool semantics to include the case when `yield` is executed as the last command by a thread.
At the type-system level, yielding control while inside a high control command, as well as inside `hide`/`unhide` pairs, is potentially dangerous. These situations are avoided by a type rule for `yield` that restricts *pc* and *hc* to low:

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

A theorem that implies soundness for the modified type system can be proved similarly to Theorem 1.
Recently, we have suggested a mechanism to enforcing security under cooperative scheduling [RS06]. 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 [RS06] allows no dynamic thread creation in high context. This is improved by the approach sketched in this section, because it retains the flexibility that is offered by `hfork`.

7 Ticket purchase example

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

Consider the code fragment in Figure 7. This fragment is a part of a program that handles a ticket purchase. Variables have subscripts indicating their security levels (l for low and h for high). Suppose f_l contains public data for the flight being booked (including the class and seat details), p_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 stores if the procedure to print a ticket has been successfully carried out.

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

This program creates threads in a high context because the guard of the `if` in the main thread depends on m_h. Furthermore, the main thread contains an assignment to a low variable (d_l) after the instructions that branches on secrets. Nevertheless, a minor modification of the program (which can, generally, be easily automated) by replacing `if` $(m_h + n_l > 50000)$ `then fork`$(s_h := GOLD, updateStatus)$ with

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

results in a typable (and therefore secure) program.

8 Implementation issues

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

$\ldots$
$n_l := computeMilesFor(f_l);$
$m_h := miles(p_l);$
$s_h := statusOf(p_l);$
$o_h := s_h;$
`if` $(m_h + n_l > 50000)$
  `then fork`$(s_h := GOLD, updateStatus);$
$ok_l := printTicket(p_l, f_l, d_l);$
$\ldots$
$updateStatus$:
`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. 7. Ticket purchase code

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. We have modified this library to allow the implementation of the primitives `hide` and `unhide` as well as a noninterferent scheduler based on the round-robin policy from Section 4. The scheduler consists of one list of threads for each security level, in this case, low and high. The scheduler interchangeably chooses between elements of those lists depending on the value of s (i.e., low and high threads when $s = \circ$, and only high ones otherwise).

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

9 Conclusion

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

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

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

Language expressiveness As discussed earlier, a flexible treatment of dynamic thread creation is a part of our model. Input/output and synchronization are also desirable features. We expect a natural extension of our model with input/output primitives on channels labeled with security levels, as well as synchronization primitives (such as semaphores) that operate on different security levels. For the two-point security lattice, we imagine the following extension of the type system. Low channels would allow low threads to input to low variables and to output low expressions. Low semaphores s would permit low threads to execute both $\mathtt{P}(s)$ and $\mathtt{V}(s)$ operations. High channels would allow high threads to input/output data and allow low threads to output data. High semaphores would allow high threads s to perform both $\mathtt{P}(s)$ and $\mathtt{V}(s)$ operations and allow low threads to perform $\mathtt{V}(s)$. Formalizing this intuition is subject to our future work.

Practical enforcement We have demonstrated that security can be enforced for both cooperative and preemptive schedulers using a compositional type system. The type system accommodates permissive programming. We have illustrated by an example in Section 7 that the permissiveness of dynamic thread creation is not majorly restricted by the type system. The type system does not involve padding to eliminate timing leaks at the cost of efficiency. Our future work plans include adapting the type system to unstructured languages (such as languages with exceptions and bytecode) and facilitating tool support for it.

Acknowledgments

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

References

[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.

[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.

[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.

[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.

[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. 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.

[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.

[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 $c = \texttt{if then}\ e\ \texttt{else}\ c_1 c_2$ or $c = \texttt{while do}\ ec$, and $\Gamma \vdash e : high$, then $\Gamma(\mathsf{hc}) = high$.

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

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

Proof. By simple induction on the typing derivation. □

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

Proof. By induction on the type derivation of c.

`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 neither produce any labeled event.

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

$$\Gamma_{hc}, hc \mapsto high \vdash c_1 : \tau' \quad (1)$$

$$\Gamma_{hc}, hc \mapsto \tau' \vdash c_2 : \tau \quad (2)$$

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

$$\langle c_1, m\rangle \xrightarrow{\alpha} \langle \texttt{stop}, m'\rangle \quad (3)$$

$$\langle c_1, m\rangle \xrightarrow{\alpha} \langle c_1', m'\rangle \quad (4)$$

Both cases are probed 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 probed 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 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

$$\Gamma_{hc,pc}, pc \mapsto \tau_e \sqcup \tau_c \sqcup high, hc \mapsto high \vdash c_t : high \tag{5}$$
$$\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c_2 : low \tag{6}$$

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

$$\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c_t : high \tag{7}$$

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 $\langle c_1; c_2, m\rangle \overset{\bullet^{\vec{d}}}{\rightharpoonup} \langle c; c_2, m\rangle$. By inspecting the typing derivation of $c_1; c_2$, we obtain that

$$\Gamma_{hc,pc}, pc \mapsto high, hc \mapsto high \vdash c : high \tag{8}$$
$$\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c_2 : low \tag{9}$$

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

$$\Gamma_{hc,pc}, pc \mapsto \tau_c, hc \mapsto high \vdash c : high \tag{10}$$

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

□

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

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

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

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

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

Proof. By induction on the size of command c.

$|c| = 1$**)** 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 proof 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

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

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 $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$, $(\Gamma[\mathit{hc} \mapsto \mathit{low}] \vdash c_i : \mathit{low})_{i \in L(\vec{c_1}, \nu_1)}$, 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 $R_9(\sigma_1, \nu_1, \sigma_2, \nu_2)$, then:

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

Proof. By case analysis on command/scheduler steps. 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.

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 $\{r, r'\} \subseteq t_{\circ_1}$ and $r_e \in t_{e_1}$).

$\alpha_1 = \circ^r_{\vec{d}}$ **)** By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, and that $c_r = \texttt{fork}(c, \vec{d})$ or $c_r = \texttt{fork}(c, \vec{d}); c^*$ for some commands c and c^*. We are only going to show the proof for the case when $c_r = \texttt{fork}(c, \vec{d}); c^*$ since the proof for $c_r = \texttt{fork}(c, \vec{d})$ proceeds in a similar way.

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

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

$\alpha_1 = r\rightsquigarrow\times$ **)** By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r \in L(\vec{c_1}, \nu_1)$, $\langle c_r, m_1\rangle \rightharpoonup \langle \texttt{stop}, m_1'\rangle$, and that $\langle \sigma_1, \nu_1\rangle \overset{r\rightsquigarrow\times}{\rightarrow} \langle \sigma_1', \nu_1'\rangle$. Because $\langle \sigma_1, \nu_1\rangle \sim_L \langle \sigma_2, \nu_2\rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2\rangle \overset{r\rightsquigarrow\times}{\rightarrow} \langle \sigma_2', \nu_2'\rangle$. 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 $\langle c_r, m_2\rangle \rightharpoonup \langle \texttt{stop}, m_2\rangle$. We can therefore conclude that the transition $\langle \vec{c_2}, m_2, \sigma_2, \nu_2\rangle \overset{r\rightsquigarrow\times}{\rightharpoonup} \langle \vec{c_2}', m_2', \sigma_2', \nu_2'\rangle$ 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 $\langle\sigma_1, \nu_1\rangle \overset{\uparrow_\circ r'}{\rightarrow} \langle\sigma_1', \nu_1'\rangle$. Because $\langle\sigma_1, \nu_1\rangle \sim_L \langle\sigma_2, \nu_2\rangle$ and α_1 is low, we also have that $\langle\sigma_2, \nu_2\rangle \overset{\uparrow_\circ r'}{\rightarrow} \langle\sigma_2', \nu_2'\rangle$. We can therefore conclude that the transition $\langle\vec{c_2}, m_2, \sigma_2, \nu_2\rangle \overset{\uparrow_\circ r'}{\rightarrow} \langle\vec{c_2}', m_2', \sigma_2', \nu_2'\rangle$ holds.

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

$\alpha_1 = r \rightsquigarrow \bullet$**)** By inspecting the semantics for threadpools, the scheduler, and commands, we have that $c_r = \texttt{hide}; c^*$ for some command c^*, $\langle c_r, m_1\rangle \overset{\rightsquigarrow\bullet}{\rightarrow} \langle c_r', m_1\rangle$, and that $\langle\sigma_1, \nu_1\rangle \overset{r\rightsquigarrow\bullet}{\rightarrow} \langle\sigma_1', \nu_1'\rangle$. We also know that $\langle c_r, m_2\rangle \overset{\rightsquigarrow\bullet}{\rightarrow} \langle c_r', m_2\rangle$ since $L(\vec{c_1}) = L(\vec{c}_2)$. Moreover, we know that $\langle\sigma_1, \nu_1\rangle \sim_L \langle\sigma_2, \nu_2\rangle$ and α_1 is low, we also have that $\langle\sigma_2, \nu_2\rangle \overset{r\rightsquigarrow\bullet}{\rightarrow} \langle\sigma_2', \nu_2'\rangle$. We can thus conclude that the transition $\langle\vec{c_2}, m_2, \sigma_2, \nu_2\rangle \overset{r\rightsquigarrow\bullet}{\rightarrow} \langle\vec{c_2}', m_2', \sigma_2', \nu_2'\rangle$ holds.

We know that $EL(\vec{c_1}) = \emptyset$ because a low thread was scheduled to produce the event $r \rightsquigarrow \bullet$. Then, $EL(\vec{c_2}) = \emptyset$ since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds. By applying Lemma 8 to c_r, we know that $c^* = c'; \texttt{unhide}$, $c^* = \texttt{unhide}$, $c^* = c'; \texttt{unhide}; p^*$, or $c^* = \texttt{unhide}; p^*$, where c' has no `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 $\sim_L$.

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

Because $\langle \sigma_1, \nu_1 \rangle \sim_L \langle \sigma_2, \nu_2 \rangle$ and α_1 is low, we also have that $\langle \sigma_2, \nu_2 \rangle \overset{\bullet\rightsquigarrow r}{\rightarrow} \langle \sigma_2', \nu_2' \rangle$. 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 belongs to the threadpool $\vec{c_2}$ as well. Let us call it c_r^2. Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds and it is not possible for a thread to make progress by a high computation, we have that $c_r^2 = \texttt{unhide}; p$. As a consequence of that, it holds that $\langle c_r^2, m_2 \rangle \overset{\bullet\rightsquigarrow}{\rightarrow} \langle p, m_2 \rangle$. Thus, transition $\langle \vec{c_2}, m_2, \sigma_2, \nu_2 \rangle \overset{\bullet\rightsquigarrow r}{\rightarrow} \langle \vec{c_2}', m_2', \sigma_2', \nu_2' \rangle$ 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 and c_r^2 become the thread p. By inspecting the semantics for the scheduler, we have that $EL(\vec{c_1}', \nu_1') = EL(\vec{c_2}', \nu_2') = \emptyset$. Then, by taking $p' = \texttt{skip}$, it trivially holds that $R_5(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2', \texttt{skip})$. $R_6(\vec{c_1}', \nu_1')$ holds since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ and $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds ; and by inspecting the type derivation of c_r. $R_7(\vec{c_1}', \nu_1', \vec{c_2}', \nu_2')$ 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 $\Gamma[hc \mapsto low] \vdash c_r : low$ 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.

High events $\bullet_{\vec{d}}^r$, $r\rightsquigarrow$, $r\rightsquigarrow\times$, and $\uparrow_\bullet r'$ (where $\{r, r'\} \subseteq t_{\bullet_1} \cup t_{e_1}$).

$\alpha_1 = \bullet_{\vec{d}}^r$ **)** 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'; `unhide`, $c_r = \texttt{hfork}(c, \vec{d})$; `unhide`, $c_r = \texttt{hfork}(c, \vec{d})$; c'; `unhide`; p, or $c_r = \texttt{hfork}(c, \vec{d})$; `unhide`; p, where c' has no `unhide` commands. Then, $R_5(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2, p')$ holds by taking $p' = p$. $R_8(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ holds by Lemma 5. Finally, $R_7(\vec{c_1'}, \nu_1', \vec{c_2}, \nu_2)$ holds since $R_7(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because high threads have made no progress.

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

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

$c_r \in EL(\vec{c_1}, \nu_1)$**)** $R_1(m_1', m_2)$ holds by applying Lemma 5. $R_2(\vec{c_1}', \nu_1')$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}', \nu_1')$, and $R_3(\vec{c_2}, \nu_2)$, hold since $R_2(\vec{c_1}, \nu_1)$, $R_2(\vec{c_2}, \nu_2)$, $R_3(\vec{c_1}, \nu_1)$, and $R_3(\vec{c_2}, \nu_2)$ hold, and by inspecting the semantics for the scheduler. $R_4(\vec{c_1}', \nu_1', \vec{c_2}, \nu_2)$ holds since $R_4(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2)$ holds and because the transition α_1 does not affect the low threads. For a similar reason, $R_6(\vec{c_1}', \nu_1')$ also holds. Since $R_5(\vec{c_1}, \nu_1, \vec{c_2}, \nu_2, p)$ holds, we know that $c_r = c'$; `unhide` , $c_r =$ `unhide`, $c_r = c'$; `unhide`; p, or $c_r =$ `unhide`; p for some command c' without `unhide` instructions. However, $c_r \neq$ `unhide`; p and $c_r \neq$ `unhide` since $\alpha_1 = r\rightsquigarrow$. The proof proceeds similarly when $c_r = c'$; `unhide` or $c_r = c'$; `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 $c_r' = c''$; `unhide`; p where $\langle c', m_1\rangle \rightharpoonup \langle c'', m_1'\rangle$ and $c'' \neq$ `stop` or $c_r' =$ `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 $\sim_L$.

□

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

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

CHAPTER

3

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

$$\frac{\langle c_i, m\rangle \xrightarrow{\alpha} \langle c_i', m'\rangle \qquad \alpha \in \{\epsilon, \vec{d}\} \qquad \sigma = i}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma, \langle c_1 \ldots c_{i-1} c_i' \alpha c_{i+1} \ldots c_n\rangle, m'\rangle}$$

$$\frac{\langle c_i, m\rangle \xrightarrow{\alpha} \langle \texttt{stop}, m'\rangle \qquad \sigma = i}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma, \langle c_1 \ldots c_{i-1} c_{i+1} \ldots c_n\rangle, m'\rangle}$$

$$\frac{\langle c_i, m\rangle \xrightarrow{\not\sim} \langle c_i', m\rangle \qquad \sigma = i \qquad \sigma' = (i \bmod n) + 1 \qquad c_i' \neq \texttt{stop}}{\langle \sigma, \langle c_1 \ldots c_n\rangle, m\rangle \rightarrow \langle \sigma', \langle c_1 \ldots c_{i-1} c_i' c_{i+1} \ldots c_n\rangle, m\rangle}$$

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 $\not\sim$. The latter label is used in the transition rule for `yield`:

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

Labels are then propagated through sequential composition to the threadpool-semantics level. Dynamic thread creation is performed by command `fork`:

$$\langle \texttt{fork}(c, \vec{d}), m\rangle \xrightarrow{\vec{d}} \langle c, m\rangle$$

This has the effect of continuing with thread c while spawning a sequence of fresh threads $\vec{d}$. *Threadpool configurations* have form $\langle\!\langle \sigma, \langle c_1 \ldots c_n \rangle, m \rangle\!\rangle$ where σ is the scheduler's running thread number, $\langle c_1 \ldots c_n \rangle$ 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 $cfg \rightarrow^0 cfg$, for any configuration cfg, and $cfg \rightarrow^v cfg'$, for $v > 0$, if there is a configuration cfg'' such that $cfg \rightarrow cfg''$ and $cfg'' \rightarrow^{v-1} cfg'$. Then, $cfg \rightarrow^* cfg'$ if $cfg \rightarrow^v cfg'$ for some $v \geq 0$. Threadpool configuration cfg *terminates* in memory m (written $cfg \Downarrow m$) if $cfg \rightarrow^* \langle\!\langle \sigma, \langle\rangle, m \rangle\!\rangle$ for some σ. In particular, $cfg \Downarrow^v m$ is written when $cfg \rightarrow^v \langle\!\langle \sigma, \langle\rangle, m \rangle\!\rangle$. If $\langle\rangle$ is not finitely reachable from cfg, then cfg *diverges* (written $cfg \Uparrow$). 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* $\Gamma : IDs \rightarrow \{high, low\}$ 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., $\forall x \in IDs. \Gamma(x) = low \Longrightarrow m_1(x) = m_2(x)$.
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*

$$\forall m_1, m_2. m_1 =_L m_2 \;\;\&\;\; \langle\!\langle 1, \langle c \rangle, m_1 \rangle\!\rangle \Downarrow m_1' \;\;\&\;\; \langle\!\langle 1, \langle c \rangle, m_2 \rangle\!\rangle \Downarrow m_2' \Longrightarrow m_1' =_L m_2'$$

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*

$$\forall m_1, m_2. m_1 =_L m_2 \Longrightarrow$$
$$\langle\!\langle 1, \langle c \rangle, m_1 \rangle\!\rangle \Downarrow m_1' \;\&\; \langle\!\langle 1, \langle c \rangle, m_2 \rangle\!\rangle \Downarrow m_2' \;\&\; m_1' =_L m_2' \vee \langle\!\langle 1, \langle c \rangle, m_1 \rangle\!\rangle \Uparrow \& \langle\!\langle 1, \langle c \rangle, m_2 \rangle\!\rangle \Uparrow$$

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

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

$$\text{(HCTX)}\frac{\text{No } \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash c : \mathit{high}}$$

$$\frac{}{\Gamma \vdash \texttt{skip} \hookrightarrow \texttt{skip}; \texttt{yield}} \qquad \frac{}{\Gamma \vdash \texttt{yield} \hookrightarrow \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \sqsubseteq \Gamma(v)}{\Gamma \vdash v := e \hookrightarrow v := e; \texttt{yield}} \qquad \frac{\Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$\frac{\Gamma \vdash e : \mathit{low} \quad \Gamma \vdash c_1 \hookrightarrow c_1' \quad \Gamma \vdash c_2 \hookrightarrow c_2'}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ (\texttt{yield}; c_1')\ \texttt{else}\ (\texttt{yield}; c_2')}$$

$$\text{(H-IF)}\frac{\Gamma \vdash e : \mathit{high} \quad \Gamma \vdash c_1 : \mathit{high} \quad \Gamma \vdash c_2 : \mathit{high}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2); \texttt{yield}}$$

$$\frac{\Gamma \vdash e : \mathit{low} \quad \Gamma \vdash c \hookrightarrow c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ (\texttt{yield}; c')); \texttt{yield}}$$

$$\text{(H-W)}\frac{\Gamma \vdash e : \mathit{high} \quad \Gamma \vdash c : \mathit{high}}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow (\texttt{while}\ e\ \texttt{do}\ c); \texttt{yield}}$$

$$\frac{\Gamma \vdash c \hookrightarrow c' \quad \Gamma \vdash d_1 \hookrightarrow d_1' \quad \ldots \quad \Gamma \vdash d_n \hookrightarrow d_n'}{\Gamma \vdash \texttt{fork}(c, d_1 \ldots d_n) \hookrightarrow \texttt{fork}(c', d_1' \ldots d_n')}$$

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:

$$\text{(HCTX')}\frac{\text{No } \texttt{while}, \texttt{yield}, \texttt{fork} \text{ or assignment to } l \text{ in } c}{\Gamma \vdash_{\text{TS}} c : high}$$

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 $\vec{c}_\sigma = \langle c_1, c_2, \ldots, c_{\sigma-1}, c_{\sigma+1}, \ldots, c_m\rangle$.

$c_\sigma = \texttt{if e then c}_1 \texttt{ else c}_2$**)** When $\Gamma \vdash e : low$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH 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 $b_1 = \texttt{True}$ and $b_2 = \texttt{False}$. We know that $c_\sigma' = (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2); \texttt{yield}$ by applying the transformation to c_σ. By inspecting the semantics for threadpools, we know that

$$\langle \sigma, c_\sigma' \oplus \vec{c}\,'_\sigma, m_1\rangle \rightharpoonup \langle \sigma, (c_1; \texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1\rangle$$
$$\langle \sigma, c_\sigma' \oplus \vec{c}\,'_\sigma, m_2\rangle \rightharpoonup \langle \sigma, (c_2; \texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2\rangle$$

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

$$\langle \sigma, (c_1; \texttt{yield}) \oplus \vec{c}\,'_\sigma, m_1\rangle \rightharpoonup^* \langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1''\rangle \quad (1)$$
$$\langle \sigma, (c_2; \texttt{yield}) \oplus \vec{c}\,'_\sigma, m_2\rangle \rightharpoonup^* \langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2''\rangle \quad (2)$$

where $m_1 =_L m_1''$ and $m_2 =_L m_2''$. Additionally, we know by 1 and 2 that

$$\langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_1''\rangle \rightharpoonup \langle \sigma', \vec{c}\,'_\sigma, m_1''\rangle \quad (3)$$
$$\langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}\,'_\sigma, m_2''\rangle \rightharpoonup \langle \sigma', \vec{c}\,'_\sigma, m_2''\rangle \quad (4)$$

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.

$\rightarrow_{1,0}^{1}$**)** The only possibilities are that $c_\sigma = \texttt{yield}$. The lemma trivially holds in this case.

$\rightarrow_{1,0}^{v+1}, v \geq 1$**)**

$c_\sigma = \texttt{if e then c}_1\ \texttt{else c}_2$**)** When $\Gamma \vdash e : \mathit{low}$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH afterwards. The interesting case is when $\Gamma \vdash e : \mathit{high}$ 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

$$\langle \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_1\rangle \rightharpoonup_{0,0} \langle \sigma, (c_1; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_1\rangle \quad (6)$$
$$\langle \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_2\rangle \rightharpoonup_{0,0} \langle \sigma, (c_2; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_2\rangle \quad (7)$$

By H, we know that $\Gamma \vdash_{\mathrm{TS}} c_i : \mathit{high}$. Thus, we can apply Lemma 1 to obtain that

$$\langle \sigma, (c_1; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_1\rangle \rightharpoonup^*_{0,0} \langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m^*_1\rangle \quad (8)$$
$$\langle \sigma, (c_2; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_2\rangle \rightharpoonup^*_{0,0} \langle \sigma, (\texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m^*_2\rangle \quad (9)$$

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

$$\langle \sigma', \vec{c}'_\sigma, m^*_1\rangle \rightharpoonup^k_{1,0} \langle \sigma'', \vec{c}'', m'_1\rangle \quad (10)$$
$$\langle \sigma', \vec{c}'_\sigma, m^*_2\rangle \rightharpoonup^k_{1,0} \langle \sigma'', \vec{c}'', m'_2\rangle \quad (11)$$

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 $y + f > 0$, $\Gamma \vdash_{\mathrm{TS}} c_i \hookrightarrow c'_i$ where $c_i \in \vec{c}$ and $c'_i \in \vec{c}'$, $m_1 =_L m_2$, and $\langle \sigma, \langle \vec{c}'\rangle, m_1\rangle \rightharpoonup^*_{y,f} \langle \sigma', \langle \vec{c}''\rangle, m'_1\rangle$, then there exists m'_2 such that $\langle \sigma, \langle \vec{c}'\rangle, m_2\rangle \rightharpoonup^*_{y,f} \langle \sigma', \langle \vec{c}''\rangle, m'_2\rangle$, 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 $\Gamma \vdash e : \mathit{low}$, the proof proceeds by applying the semantic for threadpools to reduce the `if` construct, and by applying IH afterwards. The interesting case is when $\Gamma \vdash e : \mathit{high}$ 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

$$(\langle \sigma, c'_\sigma \oplus \vec{c}'_\sigma, m_i\rangle \rightharpoonup \langle \sigma, (c_i; \texttt{yield})_\sigma \oplus \vec{c}'_\sigma, m_i\rangle)_{i=1,2}$$

By H, we know that $(\Gamma \vdash_{\mathrm{TS}} c_i : high)_{i=1,2}$. 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 $(m_i =_L m_i^*)_{i=1,2}$.

$c_\sigma = \texttt{while e do c}$**)** Loops with secrets on guards are not allowed by the transformation. Thus, the only possible case is when $\Gamma \vdash e : low$. 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

4

Closing Internal Timing Channels by Transformation

In Proceedings of the 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 ≥ k
  then skip; skip else skip;
l := 1
```

Fig. 1. Internal timing leak with `fork`

```
p := 0;
while n ≥ 0 do
  k := 2^(n−1);
  fork(skip; skip; l := 0);
  if h ≥ 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 $\texttt{fork}((\lambda\vec{x}.c)\ @\ \vec{e})$, 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: $\texttt{fork}(c)$. 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-

$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{fork}((\lambda\vec{x}.c)\,@\,\vec{e})$$
$$\mid \texttt{stop} \mid sem := \texttt{newSem}(n) \mid \texttt{wait}(sem) \mid \texttt{signal}(sem)$$

Fig. 3. Command syntax (with x and sem ranging over variables, and n over integer literals)

$$\frac{(\vec{e}, m) \downarrow \vec{v}}{\langle \texttt{fork}((\lambda\vec{x}.d)\,@\,\vec{e}), m\rangle h \xrightarrow{\lambda\vec{x}.d,\vec{v}} \langle \texttt{stop}, m\rangle h} \qquad \frac{(sem, m) \downarrow r \qquad h(r).cnt = 0}{\langle \texttt{wait}(sem), m\rangle h \xrightarrow{\otimes r} \langle \texttt{stop}, m\rangle h}$$

$$\frac{(sem, m) \downarrow r \qquad h(r).cnt > 0 \qquad h' = h[r.cnt := r.cnt - 1]}{\langle \texttt{wait}(sem), m\rangle h \rightharpoonup \langle \texttt{stop}, m\rangle h'}$$

$$\frac{(sem, m) \downarrow r}{\langle \texttt{signal}(sem), m\rangle h \xrightarrow{\odot r} \langle \texttt{stop}, m\rangle h}$$

$$\frac{i = max(dom(h)) + 1 \qquad h' = h \cup \{i \mapsto (\text{cnt} = n, \text{que} = \langle\rangle)\}}{\langle s := \texttt{newSem}(n), m\rangle h \rightharpoonup \langle \texttt{stop}, m[s := i]\rangle h'}$$

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 $\langle\!\langle\langle(c_0, m_0) \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1})\rangle, g, h, j\rangle\!\rangle$, 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, $j \in 0 \ldots n-1$ 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 $0 \ldots n-1$ 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.

$$\begin{aligned} i' &= i, && \text{if } n' < n \text{ and } i < n-1 \\ &= 0, && \text{if } n' < n \text{ and } i = n-1 \\ &= (i+1) \bmod n', && \text{otherwise} \end{aligned} \tag{1}$$

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 $1 \ldots k$ 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 ith semaphore variable (under some enumeration) is mapped to i ($i \in dom(h_k)$).*

Define $(c, g) \Downarrow g'$ if and only if $\langle\!\langle (c, m), g_k, h_k\rangle\!\rangle 0 \to^* \langle\!\langle , g', h'\rangle\!\rangle j$, for some h' and j, where $\to^*$ is the reflexive and transitive closure of the transition relation $\to$, and m is the empty function (since the initial thread c has no local variables).

Note that the definitions of $\to^*$ and $\Downarrow$ depend on the choice of scheduler, but this is elided in the notation.

$$\frac{\langle c_i, m_i \cup g\rangle h \rightharpoonup \langle c_i', m_i' \cup g'\rangle h' \qquad (i,n,n,j) \in SC}{\langle \ldots (c_i, m_i) \ldots, g, h\rangle i \rightarrow \langle \ldots (c_i', m_i') \ldots, g', h'\rangle j}$$

$$\frac{c_i = \texttt{stop} \qquad (i,n,n-1,j) \in SC}{\langle \ldots (c_i, m_i) \ldots, g, h\rangle i \rightarrow \langle \ldots (c_{i-1}, m_{i-1})(c_{i+1}, m_{i+1}) \ldots, g, h\rangle j}$$

$$\frac{\langle c_i, m_i \cup g\rangle h \stackrel{\lambda \vec{x}.d, \vec{v}}{\rightharpoonup} \langle c_i', m_i' \cup g'\rangle h' \qquad m = \{\vec{x} \mapsto \vec{v}\} \qquad (i,n,n+1,j) \in SC}{\langle \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1}), g, h\rangle i \rightarrow \langle \ldots (c_i', m_i') \ldots (c_{n-1}, m_{n-1})(d, m), g', h'\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g\rangle h \stackrel{\otimes_{\mathrm{r}}}{\rightharpoonup} \langle c_i', m_i' \cup g'\rangle h' \\ h'' = h'[r.\mathsf{que} := (r.\mathsf{que} :: (c_i', m_i'))] \qquad (i,n,n-1,j) \in SC\end{array}}{\langle \ldots (c_i, m_i) \ldots, g, h\rangle i \rightarrow \langle \ldots (c_{i-1}, m_{i-1})(c_{i+1}, m_{i+1}) \ldots, g', h''\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g\rangle h \stackrel{\odot_{\mathrm{r}}}{\rightharpoonup} \langle c_i', m_i' \cup g'\rangle h' \\ h'(r).\mathsf{que} = (c,m) :: ws \qquad h'' = h'[r.\mathsf{que} := ws] \qquad (i,n,n+1,j) \in SC\end{array}}{\langle \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1}), g, h\rangle i \rightarrow \langle \ldots (c_i', m_i') \ldots (c_{n-1}, m_{n-1})(c, m), g', h''\rangle j}$$

$$\frac{\begin{array}{c}\langle c_i, m_i \cup g\rangle h \stackrel{\odot_{\mathrm{r}}}{\rightharpoonup} \langle c_i', m_i' \cup g'\rangle h' \\ h'(r).\mathsf{que} = \langle\rangle \qquad h'' = h'[r.\mathsf{cnt} := r.\mathsf{cnt} + 1] \qquad (i,n,n,j) \in SC\end{array}}{\langle \ldots (c_i, m_i) \ldots, g, h\rangle i \rightarrow \langle \ldots (c_i', m_i') \ldots, g', h''\rangle j}$$

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):

$$\texttt{if}\ (h \geq k)\ \texttt{then skip; skip else skip} \parallel l := 0 \parallel l := 1$$

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:

$$\frac{\Gamma; w, s, a, b, m \vdash c \hookrightarrow c' \qquad w, s \textit{ fresh}}{\Gamma \vdash c \hookrightarrow_t m := \texttt{newSem}(1); a := \texttt{newSem}(1); w := \texttt{newSem}(1); \vec{h_l} := \vec{l}; c'} \tag{2}$$

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 $\Gamma \vdash c \nhookrightarrow c'$, 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:

$$(\texttt{if } h_1 \texttt{ then skip; skip else skip}); l := 1 \parallel d \tag{3}$$

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

$$\frac{}{\Gamma; w, s, a, b, m \vdash \texttt{skip} \hookrightarrow \texttt{skip}} \qquad \frac{(\Gamma; w, s, a, b, m \vdash c_i \hookrightarrow c_i')_{i=1,2}}{\Gamma; w, s, a, b, m \vdash c_1; c_2 \hookrightarrow c_1'; c_2'}$$

$$\text{(H-ASG)}\ \frac{\Gamma \vdash e \nrightarrow e' \qquad \Gamma(x) = \mathit{high}}{\begin{array}{l}\Gamma; w, s, a, b, m \vdash x := e \hookrightarrow\ s := \texttt{newSem}(0);\\ \qquad\qquad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); x := e'; \texttt{signal}(\hat{s}))\ @\ ws);\\ \qquad\qquad w := s\end{array}}$$

$$\text{(L-ASG)}\ \frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma(x) = \mathit{low} \qquad \Gamma \vdash e \nrightarrow e'}{\begin{array}{l}\Gamma; w, s, a, b \vdash x := e \hookrightarrow\ s := \texttt{newSem}(0);\\ \qquad\qquad \texttt{wait}(m);\ x := e;\ b := \texttt{newSem}(0);\\ \qquad\qquad \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{wait}(\hat{w}); \texttt{wait}(\hat{a}); h_x := e';\\ \qquad\qquad\qquad \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s}))@wsab);\\ \qquad\qquad a := b; \texttt{signal}(m);\\ \qquad\qquad w := s\end{array}}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma; w, s, a, b, m \vdash c \hookrightarrow c'}{\Gamma; w, s, a, b, m \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow \texttt{while}\ e\ \texttt{do}\ c'}$$

$$\frac{\Gamma \vdash e : \mathit{low} \qquad (\Gamma; w, s, a, b, m \vdash c_i \hookrightarrow c_i')_{i=1,2}}{\Gamma; w, s, a, b, m \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}$$

$$\text{(H-IF)}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash e \nrightarrow e'\\ (\Gamma \vdash c_i \nrightarrow c_i')_{i=1,2} \qquad c_t = \texttt{if}\ e'\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'\end{array}}{\begin{array}{l}\Gamma; w, s, a, b, m \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\\ \qquad\hookrightarrow\ s := \texttt{newSem}(0);\\ \qquad\qquad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); c_t; \texttt{signal}(\hat{s}))\ @\ ws);\\ \qquad\qquad w := s\end{array}}$$

$$\text{(H-W)}\ \frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash e \nrightarrow e' \qquad \Gamma \vdash c \nrightarrow c' \qquad c_t = \texttt{while}\ e'\ \texttt{do}\ c'}{\begin{array}{l}\Gamma; w, s, a, b, m \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow\ s := \texttt{newSem}(0);\\ \qquad\qquad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); c_t; \texttt{signal}(\hat{s}))\ @\ ws);\\ \qquad\qquad w := s\end{array}}$$

$$\frac{\begin{array}{c}\Gamma; w', s', a, b, m \vdash d \hookrightarrow d' \qquad\qquad w', s'\ \mathit{fresh}\\ c_t = \texttt{fork}((\lambda \hat{w}\hat{s}\hat{w}'.\texttt{wait}(\hat{w}); \texttt{signal}(\hat{w}); \texttt{signal}(\hat{s}); \texttt{signal}(\hat{w}'))\ @\ \hat{w}\hat{s}w')\end{array}}{\begin{array}{l}\Gamma; w, s, a, b, m \vdash \texttt{fork}(d) \hookrightarrow\ s := \texttt{newSem}(0);\\ \qquad\qquad \texttt{fork}((\lambda \hat{w}\hat{s}.w' := \texttt{newSem}(0); c_t; d')\ @\ ws);\\ \qquad\qquad w := s\end{array}}$$

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

$$\begin{array}{l}
w := \texttt{newSem}(1); \quad //\text{initialization from top-level rule (2)} \\
s := \texttt{newSem}(0); \\
\texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); (\texttt{if}\ h_1\ \texttt{then}\ h_2 := 2 * h_2 + h_l; \texttt{skip} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \texttt{skip}); \texttt{signal}(\hat{s})) \\
\qquad @ ws) \\
w := s \\
l := 1; s := \texttt{newSem}(0); \\
\texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w}); h_l := 1; \texttt{signal}(\hat{s})) @ ws) \\
w := s
\end{array} \tag{5}$$

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

$$\begin{array}{ll} c_2' : & c_1'; s := \texttt{newSem}(0); \\ & \texttt{fork}((\lambda \hat{w} \hat{s}.\texttt{wait}(\hat{w}); h_2 := h_2 + 1; \texttt{signal}(\hat{s})) \\ & \qquad @ws; \\ & w := s; \\ & \texttt{wait}(m); l := 3; b := \texttt{newSem}(0); \\ & \texttt{fork}((\lambda \hat{w} \hat{s} \hat{a} \hat{b}.\texttt{wait}(\hat{w}); \texttt{wait}(\hat{a}); h_l := 3; \\ & \qquad \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s}))@wsab); \\ & a := b; \texttt{signal}(m); \\ & \parallel d_1' \end{array} \tag{8}$$

$$
\begin{array}{l}
c_1' : w := \texttt{newSem}(1);\\
\quad s := \texttt{newSem}(0);\\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{wait}(\hat{w});\\
\qquad\qquad \texttt{if } h_1 \texttt{ then } h_2 := 2 * h_2 + h_l;\\
\qquad\qquad\qquad\qquad \texttt{skip};\\
\qquad\qquad\qquad \texttt{else skip};\\
\qquad\qquad\quad \texttt{signal}(\hat{s}))@ws);\\
\quad w := s\\
\quad s := \texttt{newSem}(0);\\
\quad \texttt{wait}(m); l := 1; b := \texttt{newSem}(0);\\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{wait}(\hat{w}); \texttt{wait}(\hat{a}); h_l := 1;\\
\qquad\quad \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s}))@wsab);\\
\quad a := b; \texttt{signal}(m);\\
\quad w := s
\end{array}
\qquad
\begin{array}{l}
d_1' : w_d := \texttt{newSem}(1);\\
\quad \texttt{skip}; \texttt{skip};\\
\quad s_d := \texttt{newSem}(0);\\
\quad \texttt{wait}(m); l := 0; b := \texttt{newSem}(0);\\
\quad \texttt{fork}((\lambda \hat{w_d}\hat{s_d}\hat{a}\hat{b}.\texttt{wait}(\hat{w_d}); \texttt{wait}(\hat{a});\\
\qquad\quad h_l := 0; \texttt{signal}(\hat{b}); \texttt{signal}(\hat{s_d}))\\
\qquad\quad @w_d s_d ab);\\
\quad a := b; \texttt{signal}(m);\\
\quad w_d := s_d
\end{array}
$$

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 ($\texttt{wait}(\underline{\hat{w}})$) passes it to another thread ($\texttt{signal}(\underline{\hat{w}})$).
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 $nextHotel()$ returns the next available hotel in the area (for simplicity, we assume there is always at least one). Function $getHotelLocation()$ provides the location of a given hotel, and function $distance()$ computes the distance between two locations. Function $moreHotels?()$ returns true if there are more hotels for $nextHotel()$ to retrieve. Function $moreTypeRooms?()$ returns true if there are more room types for $nextTypeRoom()$. Function $showTypeRoom()$ 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 $\gamma = \{\!|(c_0, m_0) \ldots, g, h|\!\}j$. We also define $\gamma.pool = \langle(c_0, m_0)\ldots\rangle$, $\gamma.globals = g$, $\gamma.heap = h$, and $\gamma.next = j$. A program configuration γ is called *syntactically secure* if every command in $\gamma.pool$ and every command in a waiting queue of $\gamma.heap$ 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 $nextlow(\gamma) = j \bmod (\#\gamma.pool)$ where j is the least number such that $j \geq \gamma.next$ and $low?(j \bmod (\#\gamma.pool), \gamma)$.

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 $\gamma \rightarrow^* \gamma'$ and $\delta \rightarrow^* \delta'$ where γ', δ' are terminal configurations, then $\gamma' =_L \delta'$. Here $\rightarrow^*$ 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

5

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 Multi-threaded 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,

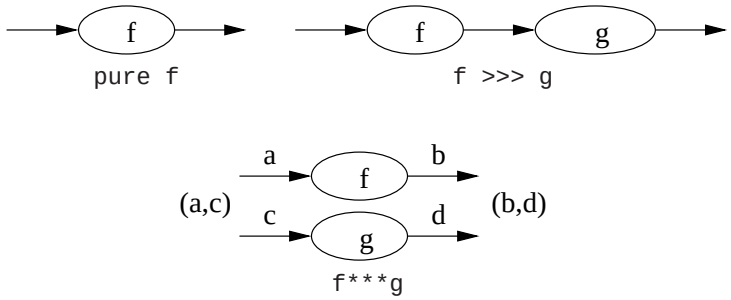

Fig. 1. Basic arrow combinators.

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) ()
```

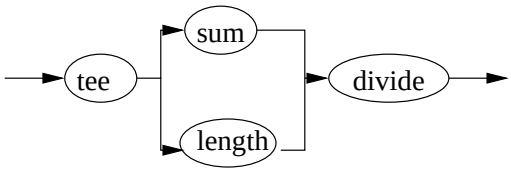

Fig. 2. Average of a list.

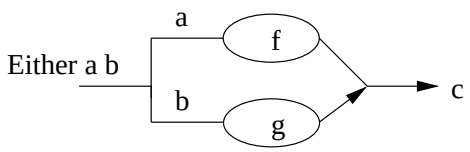

Fig. 3. Choice between `f` and `g`.

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`$\leq$`MEDIUM`$\leq$`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,
```

$$\frac{}{\vdash \texttt{pure}\ f : \ell \rightarrow \ell} \qquad \frac{C_1 \vdash f : \ell_1 \rightarrow \ell_2 \quad C_2 \vdash g : \ell_3 \rightarrow \ell_4}{C_1, C_2, \ell_2 \sqsubseteq \ell_3 \vdash f \texttt{>>>} g : \ell_1 \rightarrow \ell_4}$$

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 $\ell_1 \rightarrow \ell_2$ (`Trans l1 l2`), or the "polymorphic" $\ell \rightarrow \ell$ 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

$$\frac{f \;::\; \tau_1 \;\rightarrow\; \tau_2}{\emptyset \vdash \texttt{pure}\; f \;:\; \tau_1 \mid s_1^{\mathbb{L}} \;\rightarrow\; \tau_2 \mid \mathbf{only}\; join(s_l^{\mathbb{L}})}$$

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 $(\texttt{HIGH}, \texttt{HIGH})$. 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 $s^{\mathbb{L}}$:

$$s^{\mathbb{L}} \;::=\; \ell \mid (s^{\mathbb{L}}, s^{\mathbb{L}}) \mid (\mathbf{either}\; s^{\mathbb{L}}\; s^{\mathbb{L}}\;)^{\ell} \mid \mathbf{only}\; \ell$$

Security type $\mathbf{only}\; \ell$ 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` $\ell\ p$ 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.

$$\frac{C \vdash f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid s_2^{\mathbb{L}}}{C \vdash \texttt{lowerA}\ \ell\ f \ : \ \tau_1 \mid s_1^{\mathbb{L}} \ \rightarrow \ \tau_2 \mid \rho(\ell, \tau_2)}$$

Fig. 12. Typing rule for `lowerA`

$$\frac{}{\rho(int, \ell) \ \rightarrow \ \ell}$$

$$\frac{\rho(\tau, \ell) \ \rightarrow \ s_1^{\mathbb{L}}}{\rho(\tau \ ref, \ell) \ \rightarrow \ s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell}}$$

$$\frac{\rho(\tau_1, \ell) \ \rightarrow \ s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \ \rightarrow \ s_2^{\mathbb{L}}}{\rho((\tau_1, \tau_2), \ell) \ \rightarrow \ (s_1^{\mathbb{L}}, s_2^{\mathbb{L}})}$$

$$\frac{\rho(\tau_1, \ell) \ \rightarrow \ s_1^{\mathbb{L}} \quad \rho(\tau_2, \ell) \ \rightarrow \ s_2^{\mathbb{L}}}{\rho(either\ \tau_1\ \tau_2, \ell) \ \rightarrow \ (\mathbf{either}\ s_1^{\mathbb{L}}\ s_2^{\mathbb{L}})^{\ell}}$$

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:

$$\frac{s_1^{\mathbb{L}} = s_2^{\mathbb{L}} \quad \ell_1 \sqsubseteq \ell_2}{s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell_1} \sqsubseteq s_2^{\mathbb{L}}\ \mathbf{ref}^{\ell_2}} \tag{1}$$

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

$$(PURE)\frac{f\ :\ \tau_1\ \rightarrow\ \tau_2}{\top, \emptyset \vdash \mathtt{pure}\ f\ :\ \tau_1 \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau_2 \mid \mathbf{only}\ \ell}$$

$$(SEQ)\frac{pc_1, C_1, \vdash f_1\ :\ \tau_1 \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2 \vdash f_2\ :\ \tau_2 \mid s_3^{\mathbb{L}}\ \rightarrow\ \tau_4 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup \{s_2^{\mathbb{L}} \sqsubseteq s_3^{\mathbb{L}}\} \vdash f_1\ \ggg\ f_2\ :\ \tau_1 \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau_4 \mid s_4^{\mathbb{L}}}$$

$$(CHOICE)\frac{pc_1, C_1 \vdash f_1\ :\ \tau_1 \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau_2 \mid s_2^{\mathbb{L}} \quad pc_2, C_2, \vdash f_2\ :\ \tau_3 \mid s_3^{\mathbb{L}}\ \rightarrow\ \tau_2 \mid s_4^{\mathbb{L}}}{pc_1 \sqcap pc_2, C_1 \cup C_2 \cup C_3 \vdash f_1\ |||\ f_2\ :\ flow}$$

$$flow\ =\ either\ \tau_1\ \tau_3 \mid (\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell}\ \rightarrow\ \tau_2 \mid \uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell)$$

$$C_3 = \{(\mathbf{either} s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft (pc_1 \sqcap pc_2), (\mathbf{either}\ s_1^{\mathbb{L}}\ s_3^{\mathbb{L}})^{\ell} \blacktriangleleft e(\uparrow (s_2^{\mathbb{L}} \sqcup s_4^{\mathbb{L}}, \ell))\}$$

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 ↑

$$(CREATE)\ \frac{}{e(s_1^{\mathbb{L}}), \emptyset \vdash \texttt{createRefA}\ (\mathbf{s_1^{\mathbb{L}}})_v\ \ell\ :\ \tau \mid s_1^{\mathbb{L}}\ \rightarrow\ \tau\ ref \mid s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell}}$$

$$(READ)\ \frac{}{\top, \emptyset \vdash \texttt{readRefA} : \tau\ ref \mid s_1^{\mathbb{L}}\ \mathbf{ref}^{\ell}\ \rightarrow\ \tau \mid \uparrow (s_1^{\mathbb{L}}, \ell)}$$

$$(WRITE)\ \frac{}{e(s^{\mathbb{L}}), \{\ell \lhd s^{\mathbb{L}}\} \vdash \texttt{writeRefA} : (\tau\ ref, \tau) \mid (s^{\mathbb{L}}\ \mathbf{ref}^{\ell}, s^{\mathbb{L}})\ \rightarrow\ () \mid \bot}$$

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:

$$\begin{aligned} &t_1 : (\texttt{if h} > 0 \texttt{ then skip}(120) \texttt{ else skip}(1));\ \texttt{l} := 1 \\ &t_2 : \texttt{skip}(60);\ \texttt{l} := 0 \end{aligned} \tag{2}$$

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: 1000111000011011110010011011111100000011111111111111111

Run	Leaked credit card number	Seconds
1	1010111110011111111110111011111100000111111111111111111	27
2	1100111000011011110111011011110100000011111111111111111	27
3	1010111000011011111010011011111100000011111111111111111	28
4	1000111001011011110010011011111100000011111111111111011	28
5	1000111000011011110010011011111101000011111111111111111	29

Inferred Secret: 1000111000011011110010011011111100000011111111111111111

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 multi-threaded 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.