The Impact of Synchronisation on Secure Information Flow in Concurrent Programs

Andrei Sabelfeld

Department of Computer Science, Chalmers University of Technology
and University of Göteborg, 412 96 Göteborg, Sweden
Fax: +46 31 16 56 55, E-mail: andrei@cs.chalmers.se

Abstract. Synchronisation is fundamental to concurrent programs. This paper investigates the security of information flow in multi-threaded programs in the presence of synchronisation. We give a small-step operational semantics for a simple shared-memory multi-threaded language with synchronisation, and present a compositional timing-sensitive bisimulation-based confidentiality specification. We propose a type-based analysis improving on previous approaches to reject potentially insecure programs.

1 Introduction

1.1 Motivation

This paper focuses on the problem of program *confidentiality*, i.e., determining whether a given shared-memory multi-threaded program has secure information flow. The program runs on data partitioned on *high* (private) and *low* (public) security data, although a more general lattice of security levels can be considered. The program is not trusted, and the program's low output is publicly available as well as, possibly, timing information about the program's execution. Such a program might be received over the Internet and legitimately send its low output over the Internet such that the timing of the program's Internet accesses is observable.

1.2 Background

The problem of confidentiality for various programming languages has been investigated by many researchers including [8, 9, 7, 4, 14, 6, 15, 22, 11, 20, 1, 21, 12, 2, 17, 18, 16]. The issue of secure information flow has become especially important with the growing popularity of mobile code and networked information systems. For distributed programming, the use of multi-threaded programming languages has become extremely popular [5]. However, in the setting of a shared-memory multi-threaded language, the majority of investigations in the area of secure information flow, e.g., [11, 20, 21, 17] do not treat *synchronisation* in the language. Although the security logic of [4] does consider synchronisation primitives, there

In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, July 3-6, 2001. LNCS.

is neither a soundness proof nor a decision algorithm given for the logic. Because synchronisation is fundamental to concurrent programs, it is highly desirable to have a robust security specification and tools that aid in the design of secure programs with synchronisation.

To bridge the gap, this paper presents a compositional bisimulation-based confidentiality specification for multi-threaded programs with synchronisation and proposes a type-based analysis improving on previous approaches to reject potentially insecure programs.

1.3 Insecure Flows to Eliminate

Let us exemplify the types of insecure information flow that are in the focus of this paper. Suppose h is a high security variable and l is a low security one. There are several ways to leak information from h to the low-level observer (attacker). The simple program $l := h$ is an example of a *direct* flow. The program $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$ features an *indirect* flow through branching on a high condition.

The attacker may deduce secret information from the timing behaviour of the program. The program

$$\mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while}\ h < \mathit{MaxInteger}\ \mathsf{do}\ h := h + 1)\ \mathsf{else\ skip}$$

exemplifies a *timing* leak. Notice that the program does not operate on low variables. Nevertheless, assuming the program's execution time is visible for the low-observer, this is clearly a potential leak. Indeed, in case the initial value of h was 1 the program would take more time to execute than otherwise. The program

$$\mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while\ True\ do\ skip})\ \mathsf{else\ skip}$$

is a variation of the timing leak called a *termination* leak. Observing the termination of the program reveals that h was not 1.

Blocking a thread can change the observable behaviour of a computation, e.g., its termination behaviour. If blocking depends on high data, then the attacker might learn secrets through the observable behaviour. Concrete examples of *synchronisation* leaks are postponed until Section 4 where concrete synchronisation primitives will be available.

1.4 Overview

The rest of the paper is organised as follows. Section 2 introduces the syntax and semantics of a multi-threaded language. Section 3 motivates and specifies a bisimulation-based security definition. Section 4 gives a type system for detecting secure programs and shows its correctness with respect to the security definition. Section 5 gives an example of secure programming with synchronisation. Section 6 concludes by discussing related work.

2 A Multi-threaded Language with Synchronisation

To set the scene, let us begin by defining the syntax and semantics of a simple multi-threaded language with dynamic thread creation and synchronisation. Semaphores are widely used as a synchronisation primitive in shared-memory languages. As pointed out by Andrews ([5], p.viii): *"Semaphores were the first high-level concurrent programming mechanism and remain one of the most important."* Semaphores are general enough to implement both *mutual exclusion* and *condition synchronisation*. In this section we introduce semaphore primitives and give a small-step operational semantics for the multi-threaded language.

2.1 Semaphores

A *semaphore* [10] is a special variable (call it *sem*) that can only be manipulated by two commands: $\mathsf{wait}(sem)$ and $\mathsf{signal}(sem)$. The value of *sem* ranges over nonnegative integers. Initially, the value is 0 for every semaphore. The $\mathsf{wait}(sem)$ command blocks until *sem* is positive. Once *sem* is positive, it gets decremented by 1. The $\mathsf{signal}(sem)$ command increments *sem* by 1.

2.2 Semaphores by Busy Waiting

One approach to introducing semaphores in the language is to use macrodefinitions and define the synchronisation primitives by while-loop-based *busy waiting* (as in, e.g., [4]):

$$\begin{aligned} \mathsf{wait}(sem) &= (\mathsf{while}\ sem = 0\ \mathsf{do}\ \mathsf{skip}); sem := sem - 1 \\ \mathsf{signal}(sem) &= sem := sem + 1 \end{aligned}$$

Although these definitions are intuitive, there are two major problems with them. First, a waiting process *occupies the CPU with idle spinning.* Also, *delay and decrement must be a single atomic action.* Otherwise two $\mathsf{wait}(sem)$ threads might succeed when the initial value of *sem* is 1! Atomicity may be hard to implement in a distributed setting (not to mention that timing behaviour will spin out of control since one atomic action no longer takes one time unit). *Blocked waiting*, which commonly underlies semaphore implementations, does not have the disadvantages above. It is important that we adapt blocked waiting, since the dynamics of thread (un)blocking in a program's execution needs to be explicitly modelled due to its potential to affect the program's security.

2.3 Semaphores by Blocked Waiting

In order to define blocked waiting, we will use a special pool of waiting processes. The idea is that blocked processes should be sleeping (as opposed to spinning in busy waiting) until the respective signal is sent.

The syntax of the language is given in Figure 1. Let $C, D, E, \ldots$ range over commands *Com*, and let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$.

Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\vec{Com} = \cup_{n \in \mathbb{N}} Com^n$, the set of thread pools (or programs). A *state* $s \in \mathbf{St}$ is a finite mapping from variables (including special semaphore variables) to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables, respectively. Define *low-equivalence* by $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same.

The small-step semantics is given by transitions between *configurations*, i.e., between pairs each containing a program and a state. The deterministic part of the semantics is defined by the transition rules in Figure 2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoondown$ transitions are deterministic. There are two kinds of deterministic transitions: unlabelled and labelled. Labels are used in the semantics in order to propagate (un)blocking information. The general form of an unlabelled deterministic transition is either $\langle\!|C, s|\!\rangle \rightharpoondown \langle\!|\langle\rangle, s'|\!\rangle$, which means termination with the final state s', or $\langle\!|C, s|\!\rangle \rightharpoondown \langle\!|C'\vec{D}, s'|\!\rangle$. Here, one step of computation that starts with a command C in a state s gives a new main thread C', a (possibly empty) vector of spawned processes $\vec{D}$ and a new state s'. Command $\mathsf{fork}(C, \vec{D})$ dynamically creates a new vector of processes $\vec{D}$ which run in parallel with the main thread C. This has the effect of adding the vector of threads to the configuration.

Let sem be a variable from the special set Sem of semaphore variables. The general form of a labelled deterministic transition is either $\langle\!|C, s|\!\rangle \overset{\alpha}{\rightharpoondown} \langle\!|\langle\rangle, s'|\!\rangle$ or $\langle\!|C, s|\!\rangle \overset{\alpha}{\rightharpoondown} \langle\!|C', s'|\!\rangle$ where the label α ranges over the set of possible labels $\alpha \in \{\otimes sem, \odot sem \mid sem \in Sem\}$. These labels correspond to blocking and unblocking upon a respective semaphore. The $\mathsf{wait}(sem)$ command emits the "block" label $\otimes sem$ in case the value of sem is 0. Otherwise it decreases the value of sem. The $\mathsf{signal}(sem)$ emits the "unblock" label $\odot sem$. The labels are propagated through the sequential composition to the top level.

The concurrent part of the semantics is presented in Figure 3. The $\rightarrow$ transitions are nondeterministic. The nondeterminism is resolved by the scheduler in a particular implementation. The scheduler might be probabilistic and history-dependent, but, aiming at scheduler-independent security, we make no assumptions about peculiarities of a particular scheduler. The $\rightarrow$ transitions have the form $\langle\!|\vec{C}, w, s|\!\rangle \rightarrow \langle\!|\vec{C'}, w', s'|\!\rangle$ where configurations are equipped with queues of waiting processes $w, w' : Sem \rightarrow \vec{Com}$. Whenever the top level receives a $\otimes sem$ signal, the blocked thread is put in the end of the FIFO queue associated with sem. If the top level receives an $\odot sem$ signal, the first thread in the FIFO gets awakened or, in case, the FIFO is empty, the value of sem gets incremented.

We can extract a simple model of the timing behaviour of multi-threaded programs from the small-step semantics. This is done by assuming that each $\rightharpoondown$ transition takes a single unit of time to execute. This approach gives only a rough approximation of the actual timing behaviour, but simple extensions are possible in order to make it sensitive to the timing behaviour of particular commands (cf. [2]).

$$
\begin{aligned}
C ::= \;& \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\
& \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C, \vec{D}) \mid \mathsf{wait}(Sem) \mid \mathsf{signal}(Sem)
\end{aligned}
$$

Fig. 1. Command syntax

$$\langle \mathsf{skip}, s\rangle \rightharpoondown \langle\langle\rangle, s\rangle \qquad \frac{\langle exp, s\rangle \downarrow n}{\langle x := exp, s\rangle \rightharpoondown \langle\langle\rangle, [x := n]s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightharpoondown \langle\langle\rangle, s'\rangle}{\langle C_1; C_2, s\rangle \rightharpoondown \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \rightharpoondown \langle C_1'\vec{D}, s'\rangle}{\langle C_1; C_2, s\rangle \rightharpoondown \langle (C_1'; C_2)\vec{D}, s'\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \rightharpoondown \langle C_1, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \rightharpoondown \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \rightharpoondown \langle C; \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle} \qquad \frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \rightharpoondown \langle\langle\rangle, s\rangle}$$

$$\frac{\langle sem, s\rangle \downarrow n \quad n > 0}{\langle \mathsf{wait}(sem), s\rangle \rightharpoondown \langle\langle\rangle, [sem := sem - 1]s\rangle} \qquad \frac{\langle sem, s\rangle \downarrow 0}{\langle \mathsf{wait}(sem), s\rangle \overset{\otimes sem}{\rightharpoondown} \langle\langle\rangle, s\rangle}$$

$$\langle \mathsf{fork}(C, \vec{D}), s\rangle \rightharpoondown \langle C\vec{D}, s\rangle \qquad \frac{\langle C_1, s\rangle \overset{\alpha}{\rightharpoondown} \langle C_1', s'\rangle \quad \alpha \in \{\otimes sem, \odot sem\}}{\langle C_1; C_2, s\rangle \overset{\alpha}{\rightharpoondown} \langle C_1'; C_2, s'\rangle}$$

$$\langle \mathsf{signal}(sem), s\rangle \overset{\odot sem}{\rightharpoondown} \langle\langle\rangle, s\rangle$$

Fig. 2. Small-step deterministic semantics of commands

$$\frac{\langle C_i, s\rangle \rightharpoondown \langle \vec{C}, s'\rangle}{\langle\langle C_1 \dots C_n\rangle, w, s\rangle \rightarrow \langle\langle C_1 \dots C_{i-1}\vec{C}C_{i+1} \dots C_n\rangle, w, s'\rangle}$$

$$\frac{\langle C_i, s\rangle \overset{\otimes sem}{\rightharpoondown} \langle C', s'\rangle \qquad w_{sem} = \vec{D}}{\langle\langle C_1 \dots C_n\rangle, w, s\rangle \rightarrow \langle\langle C_1 \dots C_{i-1}C_{i+1} \dots C_n\rangle, [w_{sem} := \vec{D}C']w, s'\rangle}$$

$$\frac{\langle C_i, s\rangle \overset{\odot sem}{\rightharpoondown} \langle C', s'\rangle \qquad w_{sem} = C\vec{D}}{\langle\langle C_1 \dots C_n\rangle, w, s\rangle \rightarrow \langle\langle C_1 \dots C_{i-1}C'C_{i+1} \dots C_n C\rangle, [w_{sem} := \vec{D}]w, s'\rangle}$$

$$\frac{\langle C_i, s\rangle \overset{\odot sem}{\rightharpoondown} \langle C', s'\rangle \qquad w_{sem} = \langle\rangle}{\langle\langle C_1 \dots C_n\rangle, w, s\rangle \rightarrow \langle\langle C_1 \dots C_{i-1}C'C_{i+1} \dots C_n\rangle, w, [sem := sem + 1]s'\rangle}$$

Fig. 3. Concurrent semantics of thread pools

3 Security Specification

What is a secure program in the language we have just defined? This section focuses on defining confidentiality and motivating the chosen definition.

3.1 Noninterference

The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, the absence of dependencies). If direct, indirect, and timing flows are considered then, intuitively, a program has the extensional *noninterference* property if *varying the high input will not change the possible low-level observations*, i.e., low inputs, low outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully pursued the extensional view, including [7, 22, 11, 20, 1, 21, 12, 2, 17, 18, 16] for the justification of security analyses and verification techniques for different languages. We follow the extensional approach and focus on the extensional security for our language.

The main idea behind the bisimulation-based approach promoted by [17] is to formalise the indistinguishability of the behaviours of two programs C and D for the attacker by $C \sim_L D$, where $\sim_L$ is a *low-bisimulation.* Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply:

$$C \text{ is secure iff } C \sim_L C$$

For our purposes we adapt a variation of the *strong low-bisimulation* [17].

Definition 1 *Define the* strong *low-bisimulation* $\cong_L$ *to be the union of all symmetric relations* R *on thread pools of equal size, such that if* $\langle C_1 \ldots C_n \rangle \; R \; \langle D_1 \ldots D_n \rangle$ *then*

$$\forall s_1 =_L s_2 \forall i \forall \alpha . \langle\!| C_i, s_1 |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| \vec{C}', s_1' |\!\rangle \Longrightarrow \\ \exists \vec{D}', s_2' . \langle\!| D_i, s_2 |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| \vec{D}', s_2' |\!\rangle, \vec{C}' \; R \; \vec{D}', s_1' =_L s_2'$$

where $\alpha \in \{\epsilon, \otimes sem, \odot sem\}$. The security specification then is:

Definition 2 $\vec{C}$ *is* secure $\Longleftrightarrow \vec{C} \cong_L \vec{C}$.

One can show that the relation $\cong_L$ is transitive, but not reflexive. E.g., $l := h \not\cong_L l := h$, which reflects the fact that the program behaves differently for the attacker, depending on the initial value of h.

The choice of this bisimulation allows for robust security. As argued in [17], *strong low-bisimulation captures timing flows.*[1] If two commands may have a different timing behaviour depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. Also, *strong low-bisimulation is scheduler-independent.* Thus, our notion of security is robust with respect to any choice of a particular scheduler.

Thanks to the clear separation between deterministic and nondeterministic semantics, the underlying theory of [17] is applicable to the new semantics, which, e.g, suggests a generalisation to probabilistic schedulers (although it is outside the scope of this paper). Despite the fact that these features impose restrictions on what can be considered low-bisimilar, the choice of *strong low-bisimulation is adequate* (not too restrictive) for the type-based analysis proposed in Section 4. We will prove that whenever a command is typable in our system, it is secure.

4 Type-based Security Analysis

This section presents an automatic compositional analysis for certifying secure programs, extending and improving on previous approaches [4, 21, 2, 17]. The section concludes by stating soundness results.

4.1 The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct, indirect and synchronisation insecure information flows then it might be accepted by the system and transformed into a program that is also free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, $\vec{C'}$ is the result of its transformation and $\vec{Sl}$ is the type of $\vec{C'}$. The type $\vec{Sl}$ is $\vec{C'}$'s *low slice*, i.e., essentially a copy of $\vec{C'}$ in which assignments to (and conditionals on) high variables have been replaced by skip's. The low slice $\vec{Sl}$ has no occurrences of h and models the timing behaviour of $\vec{C'}$, as observable by other threads.

The typing and transformation rules are presented in Figure 4. The variables h and l have the types *high* and *low* respectively. Value literals n may be considered as either *high* or *low*. An arbitrary expression *Exp* is considered *high*. In all but the If_{high} rule, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. The command skip is its own low slice and therefore its own type. The rule Set_{low} prevents direct insecure information flows—the assignment $l := h$ is not typable. The rule Set_{high} types an assignment to the high variable with the low slice skip. The rules Exp_{low}, Seq, If_{low}, Par and Fork propagate types compositionally. The guard of the while-loop

[1] Nevertheless timing attacks at the layer below our assumptions (using, e.g., non-atomicity of semantics steps (cf. Section 2.3) or cache behaviour [3]) are still possible.

$$[\mathrm{Exp}] \quad h : high \qquad l : low \qquad n : \tau \qquad Exp : high$$

$$[\mathrm{Exp}_{low}] \quad \frac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$$

$$[\mathrm{Skip}] \quad \mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$$

$$[\mathrm{Set}_{low}] \quad \frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$$

$$[\mathrm{Set}_{high}] \quad h := Exp \hookrightarrow h := Exp : \mathsf{skip}$$

$$[\mathrm{Seq}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$$

$$[\mathrm{While}] \quad \frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

$$[\mathrm{Par}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \ \ldots \ C_n \hookrightarrow C_n' : Sl_n}{\langle C_1 \ldots C_n \rangle \hookrightarrow \langle C_1' \ldots C_n' \rangle : \langle Sl_1 \ldots Sl_n \rangle}$$

$$[\mathrm{Fork}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\mathsf{fork}(C_1, \vec{C_2}) \hookrightarrow \mathsf{fork}(C_1', \vec{C_2'}) : \mathsf{fork}(Sl_1, \vec{Sl_2})}$$

$$[\mathrm{Wait}] \quad \frac{sem : low}{\mathsf{wait}(sem) \hookrightarrow \mathsf{wait}(sem) : \mathsf{wait}(sem)}$$

$$[\mathrm{Signal}] \quad \frac{sem : low}{\mathsf{signal}(sem) \hookrightarrow \mathsf{signal}(sem) : \mathsf{signal}(sem)}$$

$$[\mathrm{If}_{low}] \quad \frac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$$

$$[\mathrm{If}_{high}] \quad \frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

Fig. 4. Security type system

in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard. The rules Wait and Signal guarantee that only low semaphores are used for synchronisation.

In addition to the insecure flows exemplified in Section 1, there are further ways to leak information through blocking. Synchronising on a high semaphore variable leads to (un)blocking of a thread and, clearly, may affect the termination behaviour of the program. As a consequence, neither $\mathsf{wait}(sem)$ nor $\mathsf{signal}(sem)$ on a high semaphore is allowed by the type system. The rule If_{high} prevents indirect insecure flows and timing flows. An indirect flow might be performed through synchronisation on a low semaphore depending on a high guard, e.g., $\mathsf{if}\ h = 1\ \mathsf{then}\ \mathsf{wait}(sem)\ \mathsf{else}\ \mathsf{skip}$. Thus, the type system prevents from synchronisation in the branches of an if-on-high. Let $al(C)$ be a boolean function returning True whenever there is a syntactic occurrence of either an assignment to low variable l or a synchronisation primitive (wait or signal) in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) = \mathsf{False}$ prevents indirect leaks. Both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure (preventing timing leaks) it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other (cf. Figure 5). The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

4.2 A Modification of the Cross-copying Rule

A modification of the rule If_{high} might be used to guarantee that dummy computation is inserted "evenly" and that it does not block useful computation within the branches of the resulting program. Such a rule makes use of the parallel composition instead of the sequential one when cross-copying (cf. Figure 6). Such an alternative rule If'_{high} is:

$$[\mathrm{If}'_{high}]\quad \frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\begin{array}{r}\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \mathsf{fork}(C_1'; \mathsf{wait}(sem),\ Sl_2; \mathsf{signal}(sem))\\ \mathsf{else}\ \mathsf{fork}(Sl_1; \mathsf{wait}(sem),\ C_2'; \mathsf{signal}(sem))\\ : \mathsf{skip}; \mathsf{fork}(Sl_1; \mathsf{wait}(sem),\ Sl_2; \mathsf{signal}(sem))\end{array}}$$

where sem is a fresh (low) semaphore, unused elsewhere in the program. The use of the semaphore in the rule ensures that both main threads in each branch of the resulting code have terminated before passing the control outside the if. This is necessary to prevent potential changes in the order of execution of the if and the following commands.

Interestingly, the alternative rule turns out to be less restrictive, as demonstrated in the following example. Let us denote $\mathsf{loop} = \mathsf{while\ True\ do\ skip}$. Consider the program on the left in Figure 7. Transforming the program by the rule If_{high} (as in the type systems of [2, 17]) gives a program that gets stuck with the dummy while-loop (which appears in the first position in the sequential

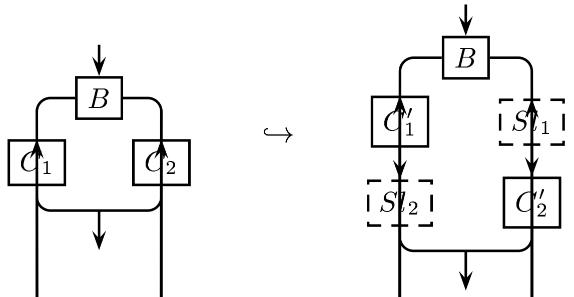

Fig. 5. Padding in rule If_{high}

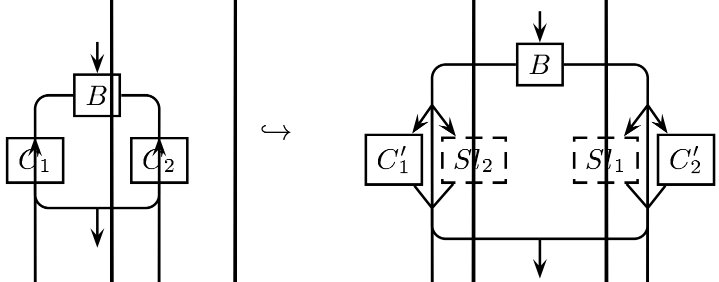

Fig. 6. Padding in rule If'_{high}

composition in both branches). This implies that the fragment $h := -h$ never gets to be executed. The result of the transformation is on the right in Figure 7. (In the examples we omit the low slices for simplicity.) As depicted in Figure 8, the modified rule If'_{high} transforms the program into a program where the high computation (the fragment $h := -h$) is enabled while the dummy loop executes in parallel.

4.3 Correctness of the Analysis

The key to straightforward correctness proofs is the *compositionality of the security property* (Definition 2). In the standard security terminology, this is called the *hook-up* property [13]. Having the hook-up property facilitates modular development of secure code.

Clearly, an arbitrary context is not necessarily secure, i.e., the low-bisimulation is not a congruence. The program $l := h$ is an example of an insecure ground context (recall that $l := h \not\cong_L l := h$). Let us go through the program constructs of the language and investigate what constraints they impose for building *secure contexts*. We identify secure contexts and give a formal proof.

Termination leak		**Padded transformation result**
$\mathsf{if}\ h > 0$	$\hookrightarrow$	$\mathsf{if}\ h > 0$
$\quad\mathsf{then\ loop}$		$\quad\mathsf{then\ loop; skip}$
$\quad\ \mathsf{else}\ h := -h$		$\quad\ \mathsf{else\ loop};\ h := -h$

Fig. 7. Example of applying If_{high}

Termination Leak		**Padded transformation result**
$\mathsf{if}\ h > 0$	$\hookrightarrow$	$\mathsf{if}\ h > 0$
$\quad\mathsf{then\ loop}$		$\quad\mathsf{then\ fork(loop; wait}(sem)\mathsf{,\ skip; signal}(sem)\mathsf{)}$
$\quad\ \mathsf{else}\ h := -h$		$\quad\ \mathsf{else\ fork(loop; wait}(sem)\mathsf{,}\ h := -h\mathsf{; signal}(sem)\mathsf{)}$

Fig. 8. Example of applying If'_{high}

Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is secure iff it has one of these forms:

$$
\begin{aligned}
\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \\
& \mid [\bullet_1]; [\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2] \ (B \text{ is low}) \\
& \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1] \ (B \text{ is low}) \mid \mathsf{fork}([\bullet_1], [\vec{\bullet_2}]) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle \\
& \mid \mathsf{wait}(sem) \ (sem \text{ is low}) \mid \mathsf{signal}(sem) \ (sem \text{ is low})
\end{aligned}
$$

where a (boolean or arithmetic) expression Exp is defined to be *low* iff $\forall s_1 =_L s_2 . \exists n. \langle\!| Exp, s_1 |\!\rangle \downarrow n \ \&\ \langle\!| Exp, s_2 |\!\rangle \downarrow n$. Otherwise, the expression is *high*.

Theorem 1 (Secure congruence) *Let $\vec{C_1} \approxeq_L \vec{C_1'}$ and $\vec{C_2} \approxeq_L \vec{C_2'}$. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$. If $\mathbb{C}[\bullet_1, \bullet_2]$ is an if-on-high context $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2] \approxeq_L \mathbb{C}[C_1', C_2']$ provided $C_1 \approxeq_L C_2$.*

Due to the separation of the semantics into deterministic and nondeterministic parts, we are able to reuse the proof of secure congruence for the language without synchronisation from [17] since it only operates on the deterministic semantics. Adding the secure contexts $\mathsf{wait}(sem)$ and $\mathsf{signal}(sem)$ (for a low sem) poses no substantial problems. An immediate corollary is the hook-up result.

Corollary 1 (Hook-up) *If $\vec{C_1}$ and $\vec{C_2}$ are secure and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided $C_1 \approxeq_L C_2$.*

Proof. Assuming $\vec{C_1}$ and $\vec{C_2}$ are secure, we have $\vec{C_i} \approxeq_L \vec{C_i}$ for $i = 1, 2$. By the security congruence theorem $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1}, \vec{C_2}]$, i.e., $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. $\square$

Since both the security property and the analysis are compositional, the correctness proof is a simple structural induction. We have the following theorem.

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \approxeq_L \vec{Sl}$.

Corollary 2 (Correctness of the Analysis) $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ *is secure.*

To see why the corollary holds, note that Theorem 2 gives $\vec{C'} \approxeq_L \vec{Sl}$. The symmetry and transitivity of $\approxeq_L$ entails $\vec{C'} \approxeq_L \vec{C'}$, i.e., $\vec{C'}$ is secure.

4.4 Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? Clearly, the padding introduced by the transformation can change the timing, but otherwise it is just additional "stuttering". To make this point precise, let us define a *weak (bi)simulation* on configurations. Let $\langle\!| C, s |\!\rangle \rightarrow^{\leq 1} \langle\!| C', s' |\!\rangle$ hold iff $\langle\!| C, s |\!\rangle = \langle\!| C', s' |\!\rangle$ or $\langle\!| C, s |\!\rangle \rightarrow \langle\!| C', s' |\!\rangle$.

Definition 3 *Define the weak simulation $\preceq$ (resp., bisimulation $\simeq$) to be the union of all (resp., symmetric) relations R on thread pools such that if $\vec{C}\ R\ \vec{D}$ and $\forall sem.\, w_{sem}\ R\ v_{sem}$ then for all s, s', w', C' there exist D' and v' such that*

$$\langle\!| \vec{C}, w, s |\!\rangle \rightarrow \langle\!| \vec{C'}, w', s' |\!\rangle \Longrightarrow \langle\!| \vec{D}, v, s |\!\rangle \rightarrow^{\leq 1} \langle\!| \vec{D'}, v', s' |\!\rangle, \\ \vec{C'}\ R\ \vec{D'}, \forall sem.\, w'_{sem}\ R\ v'_{sem}$$

Lemma 1 *$\preceq$ and $\simeq$ are reflexive, transitive and respected by all contexts (not necessarily secure), i.e., $\preceq$ is a precongruence and $\simeq$ is a congruence.*

Theorem 3 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \preceq \vec{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Set_{low}, Set_{high}, Wait and Signal follow from the reflexivity of $\preceq$. The proof for Seq, While, Par, Fork and If_{low} is by the induction hypotheses and the congruence of $\preceq$.

The remaining case is If_{high} (reasoning for If'_{high} is analogous). We have $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C'_1; Sl_2\ \mathsf{else}\ Sl_1; C'_2 : \mathsf{skip}; Sl_1; Sl_2$. By induction, $C'_1 \preceq C_1$ and $C'_2 \preceq C_2$. The side condition guarantees that there are no assignments or synchronisation in the slices Sl_1 and Sl_2 of the branches C_1 and C_2. Thus, Sl_1 and Sl_2 contain only (potentially nonterminating) dummy computation without changing the state. The insertion of such a computation might (infinitely) slow down the computation. By the definition of $\preceq$, we have $C'_1; Sl_2 \preceq C_1$ and $Sl_1; C'_2 \preceq C_2$. The congruence of $\preceq$ yields $C' \preceq C$. □

$$
\begin{array}{rl}
\textsf{Main} : & C; \mathsf{fork}(D_1, D_2) \\
C : & \mathit{empty} := 1;\ \mathit{full} := 0;\ \mathit{counter} := 0;\ \mathit{milcounter} := 0 \\
D_1 : & \textsf{while True} \\
& \quad \textsf{do} \ \ldots \ \text{/* produce a new record when a new form is submitted */} \\
& \qquad \mathsf{wait}(\mathit{empty}); \\
& \qquad \mathit{buf} := \mathit{record};\ \text{/* the record is now in the buffer */} \\
& \qquad \mathsf{signal}(\mathit{full}) \\
D_2 : & \textsf{while True} \\
& \quad \textsf{do}\ \mathsf{wait}(\mathit{full}); \\
& \qquad \mathit{result} := \mathit{buf};\ \text{/* fetch the record */} \\
& \qquad \mathsf{signal}(\mathit{empty}); \\
& \qquad \textsf{if}\ \mathit{result.org} = \textsf{military} \\
& \qquad\quad \textsf{then}\ \mathit{milcounter} := \mathit{milcounter} + 1 \\
& \qquad\quad\ \textsf{else skip} \\
& \qquad \mathit{counter} := \mathit{counter} + 1;
\end{array}
$$

Fig. 9. Example of secure programming with synchronisation

5 Secure Programming with Synchronisation

Despite restrictions imposed by the security requirements, one can still write useful programs with synchronisation. This section gives a simple example of a secure program that uses synchronisation.

Let us use *split binary semaphores* [5] in an example implementation of simple web-form processing. Suppose web forms are used for registering users of a security-sensitive website. As an entry gets processed, the low and high outputs are computed. In particular, the low variable *counter* is incremented. The high variable *milcounter* is incremented whenever the affiliation of the user is military. The program fragment is given in Figure 9.

The high variables are *record*, *buf*, *result* and *milcounter*. The low variables are *counter* and the semaphore variables *empty* and *full*. We assume a language with enriched data structures (records) for the purpose of this example, but this does not affect the security argument. The program is accepted by the type system. Thus, the program is secure and, in particular, free of timing leaks.

6 Conclusions and Related Work

We have investigated the security of information flow in multi-threaded programs in the presence of synchronisation. The main result is that allowing neither synchronisation on high nor any synchronisation (not even on low) in the branches of an if-on-high is sufficient for building up a compositional timing-sensitive security specification from previous definitions that did not consider synchronisation. We have also proposed a type-based analysis that certifies programs according to the security specification. Let us conclude by discussing some improvements of this analysis compared to previous certification systems for security.

Ref	NI	Dec	Sync	Time
Andrews & Reitman [4]	No	No	Yes	No
Heintze & Riecke [11]	No	Yes	No	No
Volpano & Smith [21]	Yes	Yes	No	protect
Agat [2]	Yes	Yes	No	Yes
Sabelfeld & Sands [17]	Yes	Yes	No	Yes
This paper	Yes	Yes	Yes	Yes

Fig. 10. Approaches to security analysis of multi-threaded programs

Figure 10 gives a comparative overview of some of the most related approaches to analysing confidentiality of multi-threaded programs. The first column **Ref** gives references to the related systems. **NI** means whether the system has been proved to certify secure programs under an extensional noninterference-like security property. The third and fifth systems have been proved to guarantee *probabilistic noninterference*. We expect probabilistic noninterference to hold for the presented system (in fact, it has been designed with probabilistic noninterference in mind). Future work includes developing a soundness proof with respect to probabilistic noninterference.

Dec stands for whether the system is given with a decision algorithm. All but the first system are decidable type systems whereas the first system is formulated as a logic with no decision algorithms or soundness proofs. **Sync** indicates whether the underlying language has synchronisation primitives. Only the first and the present investigations consider synchronisation. Furthermore, Agat [2] only considers sequential programs.

Time says whether the system captures timing covert channels. Andrews and Reitman [4] sketch the possibility of taking into account timing leaks in their security logic. However, the proposed mechanism rejects all programs that branch on high variables. Heintze and Riecke [11] stress the importance of timing-style attacks in a concurrent setting but do not give a proof that the type system accepts programs free of such attacks. Volpano and Smith consider timing flows in [21]. They allow branching on high by requiring all if-on-high commands to be embraced by special protect commands. The protect executes atomically by definition, making the timing difference invisible for the attacker. Such a command seems difficult to implement without locking the execution of every atomic command in the language or, as suggested by Smith [19], using additional mechanisms like thread priorities. Even that will not close external timing leaks, i.e., a time-consuming computation will not be hidden by a protect from the attacker with a stop-watch.

Acknowledgements

Thanks are due to David Sands for fruitful discussions and to Johan Agat and Makoto Takeyama for many valuable comments.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.
2. J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.
3. J. Agat and D. Sands. On Confidentiality and Algorithms. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, May 2001.
4. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.
5. Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.
6. J.-P. Banatre, C. Bryce, and D. Le Metayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.
7. E. S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
8. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
10. E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.
11. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.
12. K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1), 2000.
13. D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
14. M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.
15. P. Ørbæk. Can you Trust your Data? In *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, May 1995.
16. A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs.* PhD thesis, Chalmers University of Technology and Göteborg University, May 2001.
17. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000.
18. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.
19. G. Smith. Personal communication, 2000.
20. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 355–364, 19–21 January 1998.
21. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
22. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.