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 [21]. 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 [25]. 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:

$$\begin{array}{l} \texttt{if}\ h \geq k\ \texttt{then}\ \texttt{skip}; \texttt{skip}\ \texttt{else}\ \texttt{skip}; \\ l := 1 \end{array} \quad \| \quad \begin{array}{l} \texttt{skip}; \\ \texttt{skip}; \\ l := 0 \end{array} \qquad \text{(Internal timing leak)}$$

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 [25].

```
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 [25, 27, 23, 24] suggests a special $\mathtt{protect}(c)$ 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 $\mathtt{protect}()$-based security typed systems ensure that computation that branches on secrets is wrapped by $\mathtt{protect}()$ commands. However, implementing $\mathtt{protect}()$ is a major challenge [22, 19, 16] because while a thread runs $\mathtt{protect}()$, 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 [16]. However, a drawback of this approach (and, arguably, any approach that implements $\mathtt{protect}()$ 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 [25, 27, 23, 24, 16].

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 [5, 28]. The transformation ensures that the rest of insecurities (due to internal timing) are repaired.

It is seemingly possible to remove internal timing leaks by applying the following naive transformation. Suppose a command (program) c only has two variables h and l to store a secret and a public value, respectively. Assume that c does not have insecurities other than due to internal timing (this can be achieved by disallowing explicit and implicit flows, defined later in the paper). Then the following program does not leak any information about h, while it computes output as intended for c (or diverges):

$$h_i := h;\ l_i := l;\ h := 0;\ c;\ bar;\ l_o := l;\ h := h_i;\ l := l_i;\ c;\ bar;\ l := l_o$$

where bar is a barrier command that ensures that all other threads have terminated before proceeding. This transformation suffers from at least two drawbacks. Firstly, the program c is run twice, which is inefficient. Secondly, it is hard to ensure that any kind of nondeterminism (e.g., due to the scheduler, random number generator, or input channels) in c is resolved in the same way in both copies. For example, the transformation does not scale up naturally when c uses input channels. It is not obvious how to communicate inputs between the two copies of the program.

Another attempt to remove internal timing leaks could be done by applying slicing techniques, which can automatically split the original program into low and high parts. Unfortunately, these techniques in presence of concurrency are not enough to preserve the semantics of the original program. The reason for that is simple: public variables, which are updated by threads, might affect the computation of secrets. Therefore, an explicit communication of public values to the high part is required.

2 Language

Although our technique is applicable to fully-fledged programming languages, we use a simple imperative language to formalize the transformation. The language includes a command $\mathtt{fork}((\lambda\vec{x}.c) \mathbin{@} \vec{e})$, which dynamically creates and runs a new thread with local variables $\vec{x}$ with initial values given by the expressions $\vec{e}$. When the list of local variables is empty, we sometimes use simpler notation: $\mathtt{fork}(c)$. The command c may also use the program's global variables. The transformation requires dynamically allocated semaphores, so these too are included in the language defined in this section.

Without making it precise, we assume that each variable is of type integer or type semaphore. There are no expressions of type semaphore other than semaphore variables. A main program is a single command c, in the grammar of Fig. 3. Its free variables 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

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

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

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

$$\frac{(s, m) \downarrow r \qquad h(r).cnt > 0 \qquad h' = h[r.cnt := r.cnt - 1]}{\langle\!| \texttt{P}(s), m, h |\!\rangle \rightarrow \langle\!| \texttt{stop}, m, h' |\!\rangle}$$

$$\frac{(s, m) \downarrow r}{\langle\!| \texttt{V}(s), m, h |\!\rangle \xrightarrow{\odot r} \langle\!| \texttt{stop}, m, h |\!\rangle}$$

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

Fig. 4. Commands semantics

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 [29], and we thus omit these rules. The rules for concurrent commands are given in Fig. 4.

Configurations have the form $\langle\!| c, m, h |\!\rangle$, where c is a command, m is a memory (mapping variables to their values), and h is a heap for dynamically allocated semaphores. The expression language does not include dereferencing of semaphore references, so evaluation of expressions does not depend on the heap. We write $(e, m) \downarrow n$ to say that n is the value of e in memory m. A *heap* is a finite mapping from semaphore references (which we take to be naturals) to records of the form $(\text{cnt} = n, \text{que} = ws)$ where n is a natural number and ws is the list of blocked thread states.

Let α range over the following *events*, which label command transitions for use in the threadpool semantics: $\odot r$, to indicate the semaphore at reference r is signaled; $\otimes r$, to indicate it is waited; or a pair $\lambda\vec{x}.c, \vec{v}$ where $\vec{v}$ is a sequence of values that match $\vec{x}$.

Threadpool configurations have the form $\langle\!| \langle (c_0, m_0) \dots (c_i, m_i) \dots (c_{n-1}, m_{n-1}) \rangle$ $g, h, j |\!\rangle$, where each (c_i, m_i) is the state of thread i which is not blocked, g maps global variables to their values, h is the heap, $j \in 0 \dots n-1$ is the index of the thread that will take the next step. For all i, $dom(m_i)$ is disjoint from $dom(g)$. Numbering threads $0 \dots n-1$ slightly simplifies some definitions related to round-robin scheduling.

The threadpool semantics is defined for any scheduler relation SC. We interpret $(i, n, n', i') \in SC$ to mean that i is the current thread taking a step, n is the current pool size, n' is the size of the pool after that step, and i' is the next thread chosen by the scheduler. This model is adequate to define a round-robin scheduler for which thread activation, suspension, and termination do not affect the interleaving of other threads, and also to model full nondeterminism. The fully nondeterministic scheduler ND is defined by $(i, n, n', i') \in ND$ if and only if $0 \leq i < n$ and $0 \leq i' < n'$.

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 $0 \leq i < n$ and equation (1) holds.

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

The threadpool semantics is given in Fig. 5. Note that memories in command configurations are disjoint unions $m_i \cup g$, where m_i is the thread-local memory, and g is the global one. We write $h[r.\text{que} := (r.\text{que} :: (c, m))]$ to abbreviate an update of the record at r in h to change its que field by appending (c, m) at the tail. Although semaphores are stored in a heap, we streamline the semantics by not including a null reference. Thus, an initial heap is needed. It is defined to initialize semaphores to 1, which is an arbitrary choice. The security condition defined later refers to initial values for all global variables, for simplicity, but only integer inputs matter.

Definition 1. *The* initial heap of size k *is the mapping* h_k *with domain* $1 \ldots k$ *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* i*th semaphore variable (under some enumeration) is mapped to* i *(*$i \in dom(h_k)$*).*

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

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

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.

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

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

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

$$\frac{\begin{array}{c}\langle\!| c_i, m_i \cup g, h |\!\rangle \overset{\otimes r}{\rightharpoonup} \langle\!| c_i', m_i' \cup g', h' |\!\rangle \\ h'' = h'[r.\text{que} := (r.\text{que} :: (c_i', m_i'))] \qquad (i, n, n-1, j) \in SC\end{array}}{\langle\!| \langle \ldots (c_i, m_i) \ldots \rangle, g, h, i |\!\rangle \to \langle\!| \langle \ldots (c_{i-1}, m_{i-1})(c_{i+1}, m_{i+1}) \ldots \rangle, g', h'', j |\!\rangle}$$

$$\frac{\begin{array}{c}\langle\!| c_i, m_i \cup g, h |\!\rangle \overset{\odot r}{\rightharpoonup} \langle\!| c_i', m_i' \cup g', h' |\!\rangle \\ h'(r).\text{que} = (c, m) :: ws \qquad h'' = h'[r.\text{que} := ws] \qquad (i, n, n+1, j) \in SC\end{array}}{\langle\!| \langle \ldots (c_i, m_i) \ldots (c_{n-1}, m_{n-1}) \rangle, g, h, i |\!\rangle \to \langle\!| \langle \ldots (c_i', m_i') \ldots (c_{n-1}, m_{n-1})(c, m) \rangle, g', h'', j |\!\rangle}$$

$$\frac{\begin{array}{c}\langle\!| c_i, m_i \cup g, h |\!\rangle \overset{\odot r}{\rightharpoonup} \langle\!| c_i', m_i' \cup g', h' |\!\rangle \\ h'(r).\text{que} = \langle\rangle \qquad h'' = h'[r.\text{cnt} := r.\text{cnt} + 1] \qquad (i, n, n, j) \in SC\end{array}}{\langle\!| \langle \ldots (c_i, m_i) \ldots \rangle, g, h, i |\!\rangle \to \langle\!| \langle \ldots (c_i', m_i') \ldots \rangle, g', h'', j |\!\rangle}$$

Fig. 5. Threadpool semantics (for scheduler SC)

The definition says that low equality of initial global memories implies low equality of final global memories. Note that this definition is termination-insensitive [21], 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 [5] 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 Fig. 6 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 $\Gamma \vdash c \looparrowright c'$, where command c is transformed into c' under Γ. These rules guarantee that high `if`'s and `while`'s do not have assignments to low variables in their bodies. These rules for tracking explicit and implicit flows are adopted from security-type systems for sequential programs [28].

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 (which we also prove in Section 7), 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:

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

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, we can rewrite program (3) 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:

$$(\texttt{if}\ h_1\ \texttt{then}\ h_2 := 2 * h_2 + l; \texttt{skip}\ \texttt{else}\ \texttt{skip}); l := 1 \parallel d \tag{4}$$

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

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

$$\text{(H-ASG)}\ \frac{\Gamma \vdash e \looparrowright e' \qquad \Gamma(x) = \mathit{high}}{\Gamma; w, s, a, b, m \vdash x := e \hookrightarrow \begin{array}[t]{l} s := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w} \hat{s}.\texttt{P}(\hat{w}); x := e'; \texttt{V}(\hat{s}))\,@\,ws); \\ w := s \end{array}}$$

$$\text{(L-ASG)}\ \frac{\Gamma \vdash e : \mathit{low} \qquad \Gamma(x) = \mathit{low} \qquad \Gamma \vdash e \looparrowright e'}{\Gamma; w, s, a, b \vdash x := e \hookrightarrow \begin{array}[t]{l} s := \texttt{newSem}(0); \\ \texttt{P}(m);\ x := e;\ b := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w} \hat{s} \hat{a} \hat{b}.\texttt{P}(\hat{w}); \texttt{P}(\hat{a}); h_x := e'; \texttt{V}(\hat{b}); \texttt{V}(\hat{s}))\,@\,wsab) \\ a := b; \texttt{V}(m); \\ w := s \end{array}}$$

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

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

$$\text{(H-IF)}\ \frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash e \looparrowright e' \qquad (\Gamma \vdash c_i \looparrowright c_i')_{i=1,2} \qquad c_t = \texttt{if}\ e'\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}{\Gamma; w, s, a, b, m \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \hookrightarrow \begin{array}[t]{l} s := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w} \hat{s}.\texttt{P}(\hat{w}); c_t; \texttt{V}(\hat{s}))\,@\,ws); \\ w := s \end{array}}$$

$$\text{(H-W)}\ \frac{\Gamma \vdash e : \mathit{high} \qquad \Gamma \vdash e \looparrowright e' \qquad \Gamma \vdash c \looparrowright c' \qquad c_t = \texttt{while}\ e'\ \texttt{do}\ c'}{\Gamma; w, s, a, b, m \vdash \texttt{while}\ e\ \texttt{do}\ c \hookrightarrow \begin{array}[t]{l} s := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w} \hat{s}.\texttt{P}(\hat{w}); c_t; \texttt{V}(\hat{s}))\,@\,ws); \\ w := s \end{array}}$$

$$\frac{\Gamma; w', s', a, b, m \vdash d \hookrightarrow d' \quad c_t = \texttt{fork}((\lambda \hat{w} \hat{s} \hat{w}'.\texttt{P}(\hat{w}); \texttt{V}(\hat{w}); \texttt{V}(\hat{s}); \texttt{V}(\hat{w}'))\,@\,\hat{w} \hat{s} w') \quad w', s'\ \mathit{fresh}}{\Gamma; w, s, a, b, m \vdash \texttt{fork}(d) \hookrightarrow \begin{array}[t]{l} s := \texttt{newSem}(0); \\ \texttt{fork}((\lambda \hat{w} \hat{s}.w' := \texttt{newSem}(0); c_t; d')\,@\,ws); \\ w := s \end{array}}$$

$$\frac{}{\Gamma \vdash e \looparrowright e[h_x/x]_{\Gamma(x)=low}} \qquad \frac{}{\Gamma \vdash \texttt{skip} \looparrowright \texttt{skip}} \qquad \frac{\Gamma(v) = \mathit{high} \qquad \Gamma \vdash e \looparrowright e'}{\Gamma \vdash v := e \looparrowright v := e'}$$

$$\frac{\Gamma \vdash e \looparrowright e' \qquad (\Gamma \vdash c_i \looparrowright c_i')_{i=1,2}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \looparrowright \texttt{if}\ e'\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'} \qquad \frac{\Gamma \vdash d \looparrowright d'}{\Gamma \vdash \texttt{fork}(d) \looparrowright \texttt{fork}(d')}$$

$$\frac{(\Gamma \vdash c_i \looparrowright c_i')_{i=1,2}}{\Gamma \vdash c_1; c_2 \looparrowright c_1'; c_2'} \qquad \frac{\Gamma \vdash e \looparrowright e' \qquad \Gamma \vdash c \looparrowright c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \looparrowright \texttt{while}\ e'\ \texttt{do}\ c'}$$

Fig. 6. Transformation rules

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 1, waits ($\mathtt{P}(\hat{w})$) for the first one to finish, and the first one indicates when the second one should start ($\mathtt{V}(\hat{s})$). 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.

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

The thread d also needs to be modified to include an update to h_l. Let us rewrite d as in (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. 7, 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 .

$$
\begin{array}{l}
w_d := \mathtt{newSem}(1);\\
\mathtt{skip}; \mathtt{skip};\\
l := 0; s_d := \mathtt{newSem}(0);\\
\mathtt{fork}((\lambda \hat{w_d}\hat{s_d}.\mathtt{P}(\hat{w_d}); h_l := 0; \mathtt{V}(\hat{s_d}))\\
\qquad @w_d s_d);\\
w_d := s_d
\end{array}
\tag{6}
$$

Let us modify program (4) by adding assignments to high and low variables:

$$(\mathtt{if}\ h_1\ \mathtt{then}\ h_2 := 2 * h_2 + l; \mathtt{skip\ else\ skip}); l := 1; h_2 := h_2 + 1; l := 3 \parallel d \tag{7}$$

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

$$
\begin{array}{ll}
c'_1 : w := \texttt{newSem}(1); & d'_1 : w_d := \texttt{newSem}(1);\\
\quad s := \texttt{newSem}(0); & \quad \texttt{skip}; \texttt{skip};\\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{P}(\hat{w}); & \quad s_d := \texttt{newSem}(0);\\
\qquad \texttt{if } h_1 \texttt{ then } h_2 := 2 * h_2 + h_l; & \quad \texttt{P}(m); l := 0; b := \texttt{newSem}(0);\\
\qquad\qquad \texttt{skip}; & \quad \texttt{fork}((\lambda \hat{w}_d\hat{s}_d\hat{a}\hat{b}.\texttt{P}(\hat{w}_d); \texttt{P}(\hat{a}); h_l := 0;\\
\qquad\quad \texttt{else skip}; & \qquad \texttt{V}(\hat{b}); \texttt{V}(\hat{s}_d))@w_d s_d ab);\\
\qquad \texttt{V}(\hat{s}))@ws); & \quad a := b; \texttt{V}(m);\\
\quad w := s & \quad w_d := s_d\\
\quad s := \texttt{newSem}(0); & \\
\quad \texttt{P}(m); l := 1; b := \texttt{newSem}(0); & \\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{P}(\hat{w}); \texttt{P}(\hat{a}); h_l := 1; & \\
\qquad \texttt{V}(\hat{b}); \texttt{V}(\hat{s}))@wsab); & \\
\quad a := b; \texttt{V}(m); & \\
\quad w := s &
\end{array}
$$

Fig. 7. Transformed code for program (4)

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 is shown in (8). Note that spawned dedicated threads are executed in the same order as they appear in the main thread.

$$
\begin{array}{l}
c'_2 : c'_1; s := \texttt{newSem}(0);\\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}.\texttt{P}(\hat{w}); h_2 := h_2 + 1; \texttt{V}(\hat{s}))\,@\,ws)\\
\quad w := s;\\
\quad \texttt{P}(m); l := 3; b := \texttt{newSem}(0);\\
\quad \texttt{fork}((\lambda \hat{w}\hat{s}\hat{a}\hat{b}.\texttt{P}(\hat{w}); \texttt{P}(\hat{a}); h_l := 3; \texttt{V}(\hat{b});\\
\qquad \texttt{V}(\hat{s}))@wsab);\\
\quad a := b; \texttt{V}(m);\\
\quad \parallel d'_1
\end{array}
\tag{8}
$$

Let us modify program (7) to introduce a `fork` as in (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 in (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

$$
\begin{array}{l}
\texttt{if } h_1 \texttt{ then } h_2 := 2 * h_2 + l; \texttt{skip else skip};\\
l := 1; h_2 := h_2 + 1; l := 3;\\
\texttt{fork}(h_2 := 5) \parallel d
\end{array}
\tag{9}
$$

by passing-the-baton technique: whichever thread receives a signal first ($\mathtt{P}(\underline{\hat{w}})$) passes it to another thread ($\mathtt{V}(\underline{\hat{w}})$).

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 program. 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.

$$
\begin{array}{l}
c_2';\\
s := \mathtt{newSem}(0);\\
\mathtt{fork}((\lambda \hat{w}\hat{s}.w' := \mathtt{newSem}(0);\\
\qquad \mathtt{fork}((\lambda \underline{\hat{w}}\underline{\hat{s}}\hat{w}'.\mathtt{P}(\underline{\hat{w}}); \mathtt{V}(\underline{\hat{w}});\\
\qquad\qquad \mathtt{V}(\hat{s}); \mathtt{V}(\hat{w}'))@\hat{w}\hat{s}w'); d^*)\\
\qquad @ws);\\
w := s; \parallel d_1'
\end{array}
\tag{10}
$$

6 Geo-localization example

Inspired by a scenario from mobile computing [1], 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 [10] for mobile devices includes API support for obtaining the current position of the handset [11]. 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.

$$
\begin{array}{l}
\mathit{hotel}_l := \mathit{nextHotel}();\\
\mathit{hotelLoc}_l := \mathit{getHotelLocation}(\mathit{hotel}_l);\\
d_h := \mathit{distance}(\mathit{hotelLoc}_l, \mathit{userLoc}_h);\\
\mathit{closest}_h := \mathit{hotel}_l;\\
\mathtt{while}\ (\mathit{moreHotels?}())\ \mathtt{do}\\
\quad \mathit{hotel}_l := \mathit{nextHotel}();\\
\quad \mathit{hotelLoc}_l := \mathit{getHotelLocation}(\mathit{hotel}_l);\\
\quad d_h' := \mathit{distance}(\mathit{hotelLoc}_l, \mathit{userLoc}_h);\\
\quad \mathtt{if}\ (d_h' < d_h)\ \mathtt{then}\ d_h := d_h'; \mathit{closest}_h := \mathit{hotel}_l\\
\qquad\qquad \mathtt{else\ skip}\\
i_h := 0;\\
\mathtt{while}\ (\mathit{moreTypeRooms?}(\mathit{closest}_h))\ \mathtt{do}\\
\quad \mathit{type}_h := \mathit{nextTypeRoom}(\mathit{closest}_h);\\
\quad \mathit{showTypeRoom}(\mathit{type}_h, i_h);\\
\quad i_h := i_h + 1;
\end{array}
$$

Fig. 8. Geo-localization example

Consider the code fragment in Fig. 8. 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 [12, 14].

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 $\mathit{hotelLoc}_l$ contain the public name and location for a given hotel, respectively. The location of the mobile device is stored in the high variable $\mathit{userLoc}_h$. Variables d_h and d_h' are used to compute the distance to a given hotel. Variable $\mathit{closest}_h$ stores the location of the closest hotel in the area. Variable i_h is used to index the type of rooms at the closest hotel. Variable type_h stores a room type, i.e., single, double, etc. Function $\mathit{nextHotel}()$ returns the next available hotel in the area (for simplicity, we assume there is always at least one). Function $\mathit{getHotelLocation}()$ provides the location of a given hotel, and function $distance()$ 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 [3, 4] reject the example because both high conditionals and low assignments appear in the body of a loop. Transformations in [22, 13] also reject the example due to the presence of a high loop in the code.

7 Soundness

This section shows that a transformed program is secure and refines the source program 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, $\mathtt{fork}((\lambda\vec{x}.c')\,@\,\vec{e})$, in c satisfies the following: there are no high conditionals or high loops or $\mathtt{V}()$ or $\mathtt{P}()$ operations related to synchronize high threads, except inside high threads forked in c'; and (iii) in high threads, there are neither low assignments nor forks of low threads.*

Lemma 1. *If $\Gamma \vdash_t c \hookrightarrow c'$ then c' is syntactically secure.*

We let γ and δ range over threadpool configurations. We assume, for convenience in the notation, that $\gamma = \langle\!|\langle (c_0, m_0) \ldots\rangle, g, h, j|\!\rangle$. 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 $nextlow(\gamma) = j \textbf{ mod } (\#\gamma.pool)$ where j is the least number such that $j \geq \gamma.next$ and $low?(j \textbf{ mod } (\#\gamma.pool), \gamma)$.

Definition 4 (Low equality). *Define $P =_L P'$ for threadpools $P = \langle(c_1, m_1)\ldots\rangle$ and $P' = \langle(c'_1, m'_1)\ldots\rangle$ (not necessarily the same length) if and only if $c_i \equiv c'_j$ for all i, j such that $low?(m_i)$, $low?(m'_j)$, and $lowpos(i, P) = lowpos(j, P')$. Define $\gamma =_L \delta$ if and only if γ and δ are syntactically secure, $\gamma.globals =_L \delta.globals$, $\gamma.pool =_L \delta.pool$, $lowpos(nextlow(\gamma), \gamma) = lowpos(nextlow(\delta), \delta)$, and all threads blocked in $\gamma.heap$ and $\delta.heap$ are high.*

Theorem 1. *Let γ and δ be configurations such that $\gamma =_L \delta$. If $\gamma \rightarrow^* \gamma'$ 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.

Theorem 2. *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 [7, 6, 18, 8, 15], but without considering the impact of scheduling.

As discussed in the introduction, a series of work by Volpano and Smith [25, 27, 23, 24] 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 [17]).

Boudol and Castellani [3, 4] 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 [26] suggest a transformation to split a program into high and low components. Jif/split [31] 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 or refinement.

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 [22] as well as languages with synchronization [19] by Sabelfeld and message passing [20] by Sabelfeld and Mantel. Typically, padding techniques [2, 22, 13] 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 [30] and improved by Huisman et al. [9]. 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

[1] Report on resource and information flow security requirements, Mar. 2006. Deliverable D1.1 of the EU IST FET GC2 MOBIUS project, `http://mobius.inria.fr/`.
[2] J. Agat. Transforming out timing leaks. In *Proc. POPL'02*, pages 40–53, Jan. 2000.
[3] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*, pages 382–395. Springer-Verlag, July 2001.
[4] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.
[5] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
[6] 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.
[7] 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.

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

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

[10] JSR 118 Expert Group. Mobile information device profile (MIDP), version 2.0. Java specification request, Java Community Process, Nov. 2002.

[11] JSR 179 Expert Group. Location API for J2ME. Java specification request, Java Community Process, Sept. 2003.

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

[13] 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.

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

[15] 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.

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

[17] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. PSI'06*, volume 4378 of *LNCS*. Springer-Verlag, June 2006.

[18] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *FOSAD*, volume 2171 of *LNCS*. Springer-Verlag, 2001.

[19] 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.

[20] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*. Springer-Verlag, Sept. 2002.

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

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

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

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

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

[26] S. F. Smith and M. Thober. Refactoring programs to secure information flows. In *PLAS '06*, pages 75–84, New York, NY, USA, 2006. ACM Press.

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

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

[29] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

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

[31] 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.