Security of Multithreaded Programs by Compilation

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

[1] INRIA Sophia Antipolis, France
[2] MSR-INRIA
[3] Dept. of Computer Science and Engineering, Chalmers University of Technology, Sweden

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

1 Introduction

Information security is a pressing challenge for mobile code technologies. Current security architectures provide no end-to-end security guarantees for mobile code: such code may either intentionally or accidentally propagate sensitive information to an adversary. However, recent progress in the area of language-based information flow security [22] indicates that insecure flows in mobile code can be prevented by program analysis.

While much of existing work focuses on source languages, recent work has developed security analyses for increasingly expressive bytecode and assembly languages [4, 10, 16, 3, 5]. Given sensitivity annotations on inputs and outputs, these analyses provably guarantee noninterference [11], a property of programs that there are no insecure flows from sensitive inputs to public outputs.

It is, however, unsettling that information flow for multithreaded low-level programs has not been addressed so far. It is especially concerning because multithreaded bytecode is ubiquitous in mobile code scenarios. For example, multithreading is used for preventing screen lock-up in mobile applications [15]. In general, creating a new thread for long and/or potentially blocking computation, such as establishing a network connection, is a much recommended pattern [13].

This paper is the first to propose a framework for enforcing secure information flow for multithreaded low-level programs. We present an approach for deriving security-type systems that provably guarantee noninterference. On the code consumer side, these type systems can be used for checking the security of programs before running them.

Our solution goes beyond guarantees offered by security-type checking to code consumers. To this end, we have developed a framework for security-type preserving compilation, which allows code producers to derive security types for low-level programs

from security types for source programs. This makes our solution practical for the scenario of untrusted mobile code. Moreover, even if the code is trusted (and perhaps even immobile), compilers are often too complex to be a part of the trusted computing base. Security-type preserving compilation removes the need to trust the compiler, because the type annotations of compiled programs can be checked directly at bytecode level.

The single most attractive feature of our framework is that security is guaranteed by source type systems that are no more restrictive than ones for sequential programs. This might be counterintuitive: there are covert channels in the presence of threads, such as internal timing channels [28], that do not arise in a sequential setting. Indeed, special primitives for interacting with the scheduler have been designed (e.g., [18]) in order to control these channels. The pinnacle of our framework is that such primitives are automatically introduced in the compilation phase. This means that source-language programmers do not have to know about their existence and that there are no restrictions on dynamic thread creation at the source level. At the target level, the prevention of internal timing leaks does not introduce unexpected behaviors: the effect of interacting with the scheduler may only result in disallowing certain interleavings. Note that disallowing interleavings may, in general, affect the liveness properties of a program. Such a trade-off between between liveness and security is shared with other approaches (e.g., [26, 28, 24, 25, 18]).

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

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

If hi is originally non-zero, the last command to assign to lo is likely to be $lo := 1$. If hi is zero, the last command to assign to lo is likely to be $lo := 0$. Hence, this program is likely to leak information about hi into lo. In fact, all of hi can be leaked into lo via the internal timing channel, if the timing difference is magnified by a loop (see, e.g., [17]).

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

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

To illustrate the applicability of the framework, we instantiate it with some scheduler examples. These examples clarify what is expected of a scheduler to prevent internal timing leaks. Also, we give an instantiation of the source language with a simple imperative language, as well as an instantiation of the target language with a simple assembly language that features an operand stack, conditions, and jumps. As we will discuss, these instantiations are for illustration only: we expect our results to apply to languages close to Java and Java bytecode, respectively.

Our approach pushes the feasibility of replacing trust assumptions by type checking for mobile-code security one step further. It is especially encouraging that we inherit the main benefit of recent results on enforcing secure information flow by security-type systems [3]: compatibility with bytecode verification, and no need to trust the compiler.

2 Syntax and semantics of multithreaded programs

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

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

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

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

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

We assume the attacker level $k \in \mathsf{Level}$ partitions all elements of Level into *low* and *high* elements. Low elements are no more sensitive than k: an element ℓ is low if $\ell \leq k$. All other elements (including incomparable ones) are high. We assume that the set of high elements is not empty. This partition reduces the set Level to a two-element set $\{low, high\}$, where $low < high$, which we will adopt without loss of generality.

Programs come equipped with a *security environment* [5] that assigns a security level to each program point and is used to prevent *implicit flows* [9]. The security environment is also used by the scheduler to select the thread to execute.

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

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

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

Definition 4 (State).

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

It is convenient to use accessors to extract components from states: we use $s.\mathsf{lst}$ and $s.\mathsf{gmem}$ to denote the first and second components of a state s. Then, we use $s.\mathsf{act}$ to denote the set of active threads, i.e., $s.\mathsf{act} = \mathsf{Dom}(s.\mathsf{lst})$. We sometimes write $s(tid)$ instead of $s.\mathsf{lst}(tid)$ for $tid \in s.\mathsf{act}$. Furthermore, we assume given an accessor pc that extracts the program counter for a given thread from a local state.

We follow a concurrency model [18] that lets the scheduler distinguish between different types of threads. A thread is *low* (resp., *high*) if the security environment marks its program counter as low (resp., high). A high thread is *always high* if the program point corresponding to the program counter is always high. A high thread is *hidden* if it is high but not always high. (Intuitively, the thread is hidden in the sense that the scheduler will, independently from the hidden thread, pick the following low threads.) Formally, we have the following definitions:

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

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

We proceed to defining computation history and secure schedulers, which operate on histories as parameters.

Definition 5 (History).

1. *A history is a list of pairs* (tid, ℓ) *where* $tid \in \mathsf{Thread}$ *and* $\ell \in \mathsf{Level}$*. We denote the empty history by* ϵ^{hist}*.*
2. *Two histories* h *and* h' *are indistinguishable, written* $h \overset{\mathsf{hist}}{\sim} h'$*, if* $h|_{low} = h'|_{low}$*, where* $h|_{low}$ *is obtained from* h *by projecting out pairs with the high level in the second component.*

We denote the set of histories by History. We now turn to the definition of a secure scheduler. The definition below is of a more algebraic nature than that of [18], but captures the same intuition, namely that a secure scheduler: i) always picks an active thread; ii) chooses a high thread whenever there is one hidden thread; and iii) only uses the names and levels of low and the low part of histories to pick a low thread.

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

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

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

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

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

We assume that AT_L and AT_H are functions of s that extract the set of identifiers of low and high threads, respectively, and the partial function $last_\ell$ returns the identity of the most recently picked thread at level ℓ recorded in h, if it exists. The scheduler may only pick active threads (cf. req. 1). In addition to the alternation between high and low threads, the scheduler may only pick a low thread if there are no hidden threads (cf. req. 2). The separation into high and low threads ensures that for low-equivalent histories, the observable choices of the scheduler are the same (cf. req. 3).

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

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

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

Fig. 1. Semantics of multithreaded programs

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

Fig. 2. Typing rules

To define the execution of multithreaded programs, we assume given a (deterministic) sequential execution relation $\rightsquigarrow_{\mathrm{seq}} \subseteq \mathsf{SeqState} \times \mathsf{SeqState}$ that takes as input a current state and returns a new state, provided the current instruction is sequential.

We assume given a function $\lambda_{\mathrm{init}} : \mathcal{P} \to \mathsf{LocState}$ that takes a program point and produces an initial state with program pointer pointing to pc. We also assume given a family of functions fresht_ℓ that takes as input a set of thread identifiers and generates a new thread identifier at level ℓ. We assume that the ranges of fresht_ℓ and $\mathsf{fresht}_{\ell'}$ are disjoint whenever $\ell \neq \ell'$. We sometimes use fresht_ℓ as a function from states to Thread.

Definition 7 (Multithreaded execution). *One step execution* $\rightsquigarrow_{\mathrm{conc}} \subseteq (\mathsf{ConcState} \times \mathsf{History}) \times (\mathsf{ConcState} \times \mathsf{History})$ *is defined by the rules of Figure 1. We write* $s, h \rightsquigarrow_{\mathrm{conc}} s', h'$ *when executing* s *with history* h *leads to state* s' *and history* h'.

The first two rules of Figure 1 correspond to non-terminating and terminating sequential steps. In the case of termination, the current thread is removed from the domain of lst. The last rule describes dynamic thread creation caused by the instruction start pc. A new thread receives a fresh name $ntid$ from $\mathsf{fresht}_{se(i)}$ where $se(i)$ records the security environment at the point of creation. This thread is added to the pool of threads under the name $ntid$. All rules update the history with the current thread id and the security environment of the current instruction. The evaluation semantics of programs can be derived from the small-step semantics in the usual way. We let $main$ be the identity of the main thread.

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

3 Security policy

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

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

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

4 Type system

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

Assumptions on type system for sequential programs. We assume given a set LType of local types for typing local states, with a distinguished local type $\mathrm{T}_{\mathrm{init}}$ to type initial states, and a partial order $\leq$ on local types. Typing judgments in the sequential type system are of the form $se, i \vdash_{\mathrm{seq}} \mathrm{S} \Rightarrow \mathrm{T}$, where se is a security environment, i is a program point in program P, and S and T are local types.

Typing rules are used to establish a notion of typable program [4]; typable programs are assumed to satisfy several properties that are formulated precisely in Section 5.

Type system for multithreaded programs. The typing rules for the concurrent type system have the same form as those of the sequential type system and are given in Figure 2.

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

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

5 Soundness

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

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

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

[4] The notion of typable sequential program is a particular case of typable multithreaded program.

State equivalence. In order to prove noninterference, we rely on a notion of state equivalence. The definition is modular, in that it is derived from an equivalence between global memories $\sim_g$ and a partial equivalence relation $\sim_l$ between local states. (Intuitively, partial equivalence relations on local and global memories represent the observational power of the adversary.) In comparison to [3], equivalence between local states (operand stacks and program counters for the JVM) is not indexed by local types, since these can be retrieved from the program counter and the global type of the program.

Definition 11 (State equivalence). *Two concurrent states s and t are:*

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

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

Definition 12. *Two states s and s' are pc-equivalent, written, $s \overset{\mathsf{pc}}{\sim} s'$ iff $s.\mathsf{lowT} = t.\mathsf{lowT}$ and for every $tid \in s.\mathsf{lowT}$, we have $s.\mathsf{pc}(tid) = t.\mathsf{pc}(tid)$.*

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

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

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

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

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

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

The proof of this and other results can be found in the full version [7] of the paper.

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

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

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

The proofs of the unwinding lemmas are by a case analysis on the semantics of concurrent programs.

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

Hypothesis 4 (Existence of next function)**.** *There exists a function* $\mathsf{next} : \mathcal{P} \rightharpoonup \mathcal{P}$ *such that the* next *properties (NeP) hold:*

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

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

In addition to the above assumptions, we also need another hypothesis that relates the domain of next to the operational semantics of programs. In essence, the hypothesis states that, under the assumptions of the concurrent locally respects unwinding lemma, either the executed instruction is a low instruction, in which case the program counter of the active thread remains equal after one step of execution, or that the executed instruction is a high instruction, in which case the active thread is hidden in one execution (high loop) or both (high conditional).

Hypothesis 5 (Preservation of pc equality)**.** *Assume* $s \sim t$*;* $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = ctid$*;* $s(ctid).\mathsf{pc} = t(ctid).\mathsf{pc}$*;* $s, h_s \rightsquigarrow_{\mathrm{conc}} s', h_{s'}$*; and* $t, h_t \rightsquigarrow_{\mathrm{conc}} t', h_{t'}$*. Then,* $s'(ctid).\mathsf{pc} = t'(ctid).\mathsf{pc}$*; or* $s'(ctid).\mathsf{pc} \in \mathsf{Dom}(\mathsf{next})$*; or* $t'(ctid).\mathsf{pc} \in \mathsf{Dom}(\mathsf{next})$*.*

The final hypothesis is about visibility by the attacker:

Hypothesis 6 (High hypotheses)**.**

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

Theorem 1 follows from the hypotheses above. For the proof details, we refer to the full version of the paper [7].

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

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

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

Fig. 3. Source and target language

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

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

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

Fig. 4. Transfer rules

6 Instantiation

In this section, we apply our main results to a simple assembly language with conditional jumps and dynamic thread creation. We present the assembly language with a semantics and a type system for noninterference but without considering concurrent primitives and plug these definitions into the framework for multithreading. Then, we present a compilation function from a simple while-language with dynamic thread creation into assembly code. The source and target languages are defined in Figure 3. The compilation function allows us to easily define control dependence regions and junction points in the target code. Function next is then defined using that information. Moreover, we prove that the obtained definition of next satisfies the properties required in Section 5. Finally, we conclude with a discussion about how a similar instantiation can be done for the JVM.

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

The enforcement mechanism consists of local types which are stacks of security levels, i.e., $\mathsf{LType} = \mathsf{Stack}(\mathsf{Level})$; we let τ_{init} be the empty stack of security levels. Typing rules are summarized in Figure 4, where $\mathsf{lift}_k(st)$ denotes the point-wise exten-

sion of $\lambda k'.\ k \sqcup k'$ to stacks of security levels, and $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ denotes the region of branching points. We express the chosen security policy by assigning a security level $\Gamma(x)$ to each variable x.

Similarly to [4], the soundness of the transfer rules relies on some assumptions about control dependence regions in programs. Essentially, these regions represent an over-approximation of the range of branching points. This concept is formally introduced by the functions $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ and $\mathsf{jun} : \mathcal{P} \rightharpoonup \mathcal{P}$, which respectively compute the control dependence region and the junction point for a given instruction. Both functions need to satisfy some properties in order to guarantee noninterference in typable programs. These properties are known as SOAP properties [4]. In Section 6, we will show that these properties can be guaranteed by compilation.

In the full version [7] we instantiate definitions of local and global state equivalences to establish the soundness of the type system.

Concurrent extension. As shown in Definition 7, the concurrent semantics is obtained from the semantics for sequential commands together with a transition for the instruction `start`. Moreover, the sequential type system in Figure 4 is extended by the typing rules presented in Figure 2 to consider concurrent programs.

The proof of noninterference for concurrent programs relies on the existence of the function next. Similarly to the technique of [6], we name program points where control flow can branch or writes can occur. We add natural number labels to the source language as follows:

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

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

Definition 13 (function sregion**).** *For each branching command $[c]^n$, $sregion(n)$ is defined as the set of labels that are inside of the command c except for those ones that are inside of* `fork` *commands.*

As in [6], control dependence regions for low-level code are defined based on the function sregion and a compilation function. For a complete source program c, we define the compilation $\mathcal{C}(c)$ in Figure 5. We use symbol $\#$ to compute the length of lists. Symbol :: is used to insert one element to a list or to concatenate two existing lists. The current program point in a program is represented by pc. The function $\mathcal{C}(c)$ calls the auxiliary function $\mathcal{S}$ which returns a pair of programs. The first component of that pair stores the compiled code of the main program, while the second one stores the compilation code of spawned threads. We now define control dependence regions for assembly code and respective junction points.

Definition 14 (function tregion**).** *For a branching instruction $[c]^n$ in the source code, $tregion(n)$ is defined as the set of instructions obtained by compiling the commands $[c']^{n'}$, where $n' \in sregion(n)$. Moreover, if c is a while loop, then $n \in \mathsf{tregion}(n)$. Otherwise, the* goto *instruction after the compilation of the else-branch also belongs to* $\mathsf{tregion}(n)$.

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

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

$$\mathcal{C}(c) = let\ (lc,\ T) = \mathcal{S}(c,\ []); in\ \mathsf{goto}\ (\#T + 2) :: T :: lc :: \mathsf{return}$$

Fig. 5. Compilation function

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

Definition 15 (junction points). *For every branching point* $[c]^n$ *in the source program, we define* $\mathsf{jun}(n) = max\{i | i \in \mathsf{tregion}(n)\} + 1$.

Having defined control dependence regions and junction points for low-level code, we proceed to defining next. Intuitively, next is only defined for instructions that belong to regions corresponding to the outermost branching points whose guards involved secrets. For every instruction i inside of an outermost branching point $[c]^n$, we define $\mathsf{next}(i) = \mathsf{jun}(n)$. Observe that this definition captures the intuition about next given in the beginning of Section 5. However, it is necessary to know, for a given program, what are the outermost branching points whose guards involved secrets. With this in mind, we extend one of the type systems given in [6] to identify such points. We add some rules for outermost branching points that involved secrets together with some extra notations to know when a command is inside of one of those points or not.

A source program c is typable, written $\vdash_\circ c : E$, if its command part is typable with respect to E according to the rules given in Figure 6. The typing judgment has the form $\vdash_\alpha [c]^n_{\alpha'} : E$, where E is a function from labels to security levels. Function E can be seen as a security environment for the source code which allows to easily define the security environment for the target code. If R is a set of points, then $\mathsf{lift}_k(E, R)$ is the security environment E' such that $E'(n) = E(n)$ if $n \notin R$ and $E'(n) = k \sqcup E(n)$ for $n \in R$. For a given program c, $\mathsf{labels}(c)$ returns all the label annotations in c. Variable α denotes if c is part of a branching instruction that branches on secrets ($\bullet$) or public data ($\circ$). Variable α' represents the level of the guards in branching instructions. The most interesting rules are $TOP-H-COND$ and $TOP-H-WHILE$. These rules can be only applied when the branching commands are the outermost ones and when they branch on secrets. Observe that such commands are the only ones that are typable considering $\alpha = \circ$ and $\alpha' = \bullet$. Moreover, the type system prevents *explicit* (via assignment) and *implicit* (via control) flows [9]. To this end, the type system enforces the

$$\frac{\vdash_\alpha c : E \qquad \vdash_\alpha c' : E}{\vdash_\alpha c ; c' \ : \ E} \qquad \frac{\vdash e : L \qquad \vdash_\alpha c \ : \ E}{\vdash_\alpha [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\alpha \ : \ E}$$

$$\frac{\vdash e : L \qquad \vdash_\alpha c : E \qquad \vdash_\alpha c' : E}{\vdash_\alpha [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\alpha \ : \ E} \qquad \frac{\vdash e : H \qquad \vdash_\bullet c \ : \ E}{\vdash_\bullet [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\bullet \ : \ E}$$

$$\frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E}{\vdash_\bullet [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\bullet \ : \ E} \qquad \frac{\vdash_\alpha c : E \qquad E = \mathrm{lift}_\alpha(E, \mathsf{labels}(c))}{\vdash_\alpha [\mathsf{fork}(c)]^n_\alpha \ : \ E}$$

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

TOP-H-WHILE
$$\frac{\vdash e : H \qquad \vdash_\bullet c \ : \ E \qquad E = \mathrm{lift}_H(E, \mathsf{sregion}(n))}{\vdash_\circ [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\bullet \ : \ E}$$

TOP-H-COND
$$\frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E \qquad E = \mathrm{lift}_H(E, \mathsf{sregion}(n))}{\vdash_\circ [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\bullet \ : \ E}$$

Fig. 6. Intermediate typing rules for high-level language commands

same constraints as standard security type systems for sequential languages (e.g., [29]). Explicit flows are prevented by rule $ASSIGN$, while implicit flows are ruled out by demanding a security environment of level H inside of commands that branch on secrets. The type system guarantees information-flow security at the same time as it identifies the outermost commands that branch on secrets. Function next is defined as follows:

Definition 16 (function next**).** *For every branching point c in the source program such that* $\vdash_\circ [c]^n_\bullet$*, we have that* $\forall k \in \mathsf{tregion}(n).\mathsf{next}(k) = \mathsf{jun}(n)$.

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

Lemma 3. *Definition 16 satisfies properties **NePd** and **NeP1–4**.*

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

Corollary 1 (Soundness of the instantiation). *Hypotheses 1–6 from Section 5 are satisfied by the instantiation, and therefore the derived type system guarantees noninterference for multithreaded assembly programs.*

Hypotheses 1–3 follow from the unwinding lemmas of [5]; Hypothesis 4 from Lemma 3, and Hypotheses 5 and 6 from the definitions of next and $High^{\mathsf{lmem}}$, respectively.

Type preserving compilation. The compilation of sequential programs is type-preserving, as shown in previous work [6]. Our framework allows extending type-preservation to multithreading. Moreover, it enables us to obtain a key *non-restrictiveness* result: although the source-level type system is no more restrictive than a typical type system for

a sequential language (e.g., [29]), the compilation of (possibly multithreaded) typable programs is guaranteed to be typable at low-level. Due to the lack of space, we only give an instantiation of this result to the source and target languages of this section:

Theorem 2. *For a given source-level program c, assume $nf(c)$ is obtained from c by replacing all occurrences of* `fork`(d) *by d. If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [29] then $se, \mathcal{S} \vdash \mathcal{C}(c)$ for some se and $\mathcal{S}$.*

This theorem and Theorem 1 entail the following corollary:

Corollary 2. *If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [29] then $\mathcal{C}(c)$ is secure.*

Java Virtual Machine. The modular proof technique developed in the previous section is applicable to a Java-like language. If the sequential type system is compatible with bytecode verification, then the concurrent type system is also compatible with it. This implies that Java bytecode verification can be extended to perform security type checking. Note that the definition of a secure scheduler is compatible with the JVM, where the scheduler is mostly left unspecified. Moreover, it is possible to, in effect, override an arbitrary scheduler from any particular implementation of JVM with a secure scheduler that keeps track of high and low threads as a part of an application's own state (cf. [27]).

However, some issues arise in the definition of a concurrent JVM: in particular, we cannot adapt the semantics and results of [3] directly, because the semantics of method calls is big-step. Instead, we must rely on a more standard semantics where states include stack frames, and prove unwinding lemmas for such a semantics; fortunately, the technical details in [4] took this route, and the same techniques can be used here.

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

7 Related work

Information flow type systems for low-level languages, including JVML, and their relation to information flow type systems for structured source languages, have been studied by several authors [4, 10, 16, 6, 3, 5]. Nevertheless, the present work provides, to the best of our knowledge, the first proof of noninterference for a concurrent low-level language, and the first proof of type-preserving compilation for languages with concurrency.

This work exploits recent results on interaction between the threads and the scheduler [18] in order to control internal timing leaks. Other approaches [26, 28, 24, 25] to handling internal timing rely on `protect`(c) which, by definition, hides the internal timing of command c. It is not clear how to implement `protect`$()$ without modifying the scheduler (unless the scheduler is cooperative [19, 27]). It is possible to prevent internal timing leaks by spawning dedicated threads for computations that involve secrets and carefully synchronizing the resulting threads [17]. However, this implies high synchronization costs. Yet other approaches prevent internal timing leaks in code by disallowing any races on public data [30, 12]. However, they wind up rejecting such innocent programs as $lo := 0 \parallel lo := 1$ where lo is a public variable. Still other approaches prevent internal timing by disallowing low assignments after high branching [8, 2]. Less

related work [1, 23, 20, 21, 14] considers external timing, where an attacker can use a stopwatch to measure computation time. This work considers a more powerful attacker, and, as a price paid for security, disallows loops branching on secrets. For further related work, we refer to an overview of language-based information-flow security [22].

8 Conclusions

We have presented a framework for controlling information flow in multithreaded low-level code. Thanks to its modularity and language-independence, we have been able to reuse several results for sequential languages. An appealing feature enjoyed by the framework is that security-type preserving compilation is no more restrictive for programs with dynamic thread creation than it is for sequential programs. Primitives for interacting with the scheduler are introduced by the compiler behind the scenes, and in such a way that internal timing leaks are prevented.

We have demonstrated an instantiation of the framework to a simple imperative language and have argued that our approach is amenable to extensions to object-oriented languages. The compatibility with bytecode verification makes our framework a promising candidate for establishing mobile-code security via type checking.

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

References

1. J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.
2. A. Almeida Matos. *Typing secure information flow: declassification and mobility*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2006.
3. G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In R. D. Niccola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer, 2007.
4. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
5. G. Barthe, T. Rezk, and A. Basu. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*, 2007.
6. G. Barthe, T. Rezk, and D. Naumann. Deriving an information flow checker and certifying compiler for java. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 230–242. IEEE Computer Society, 2006.
7. G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. Technical report, Chalmers University of Technology, 2007. Located at `http://www.cs.chalmers.se/~russo/esorics07full.pdf`.
8. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
10. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proceedings of VMCAI'05*, volume 3385 of *LNCS*, pages 346–362. Springer-Verlag, 2005.
11. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

12. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.
13. J. Knudsen. Networking, user experience, and threads. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/articles/threading/`, 2002.
14. B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*, pages 47–62. Springer-Verlag, July 2006.
15. Q. H. Mahmoud. Preventing screen lockups of blocking operations. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/ttips/screenlock/`, 2004.
16. R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In M. Coppo, E. Lodi, and G. Pinna, editors, *Proceedings of ICTCS 2005*, volume 3701 of *LNCS*, pages 360–374. Springer-Verlag, 2005.
17. A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference (ASIAN'06)*, LNCS. Springer-Verlag, 2007.
18. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.
19. A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2006.
20. A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.
21. A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.
22. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
23. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
24. G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.
25. G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–13, 2003.
26. 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.
27. T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007. To appear.
28. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.
29. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
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.