Security of Multithreaded Programs by Compilation

Gilles Barthe
IMDEA Software, Madrid, Spain
and
Tamara Rezk
Inria Sophia Antipolis and MSR-INRIA, France
and
Alejandro Russo
Chalmers University of Technology, Sweden
and
Andrei Sabelfeld
Chalmers University of Technology, Sweden

---

1. INTRODUCTION

Motivation. 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 [Sabelfeld and Myers 2003] indicates that insecure flows in mobile code can be tracked by language-based techniques.

While the existing work focuses on source languages, recent work has developed security analyses for increasingly expressive bytecode and assembly languages [Barthe and Rezk 2005; Genaim and Spoto 2005; Medel et al. 2005; Barthe et al. 2007a; Barthe et al. 2007; Zanardini 2006]. Given sensitivity annotations on inputs and outputs, these analyses provably guarantee noninterference [Goguen and Meseguer 1982], a property of programs that says 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 received only little attention [Barthe et al. 2007]. 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 [Mahmoud 2004]. In general, creating a new thread for long and/or potentially blocking computation, such as establishing a network connection, is a much recommended pattern [Knudsen 2002].

Internal timing leaks. Assume variables are given security levels: either secret (*high*) or public (*low*) levels. The attacker has access to low-level variables, and so we say the attacker is at the low level. What is the main problem with reasoning about information flow for multithreaded low-level languages? In addition to explicit flows (as in *public* := *secret*) and *implicit flows* [Denning and Denning 1977] (as in `if` *secret* `then` *public* := *1*) that need to be tracked as in sequential programs, multithreading creates a new channel for information transfer: *internal*

timing [Volpano and Smith 1999].

For an example of an internal timing leak, consider a simple two-threaded source-level program, where *hi* is a high and *lo* is a 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., [Russo et al. 2007]).

Internal-timing attacks are particularly dangerous because the attacker needs no access to a clock to learn the complete secrets in linear time.(In the paper we assume the attacker does not have access to a clock.) In a language where a clock is available programmatically, this implies that the result of reading the clock is secret [Smith and Volpano 1998]. There is a separate line of work on external timing attacks (e.g., [Agat 2000; Sabelfeld and Sands 2000; Sabelfeld 2001; Sabelfeld and Mantel 2002; Köpf and Mantel 2006]), where an attacker can measure computation time. A price paid for security against this kind of more powerful attackers is restrictiveness. For example, loops with secrets guards are disallowed.

The goals and the state of the art: secure multithreading. Our main goal is security enforcement of multithreaded low-level languages that is (i) sound w.r.t. noninterference, (ii) permissive, i.e., not too many useful secure programs are rejected, and (iii) not scheduler-specific, i.e., the security does not break as we vary the scheduler (there should be a clearly-defined class of schedulers in which security is parametric).

When extending security enforcement for sequential low-languages with multithreading, we have several choices offered by the state-of-the-art in analyses of source-level multithreaded programs. While we defer a detailed discussion of the state of the art to Section 9, we remark that the most popular approaches can be roughly categorized as follows: protection/hiding-based approaches [Smith and Volpano 1998; Volpano and Smith 1999; Smith 2001; 2003; Russo and Sabelfeld 2006a], low-determinism-based approaches [Zdancewic and Myers 2003; Huisman et al. 2006; Terauchi 2008], and external timing-based approaches [Agat 2000; Sabelfeld and Sands 2000; Sabelfeld 2001; Sabelfeld and Mantel 2002; Köpf and Mantel 2006].

The protection/hiding-based approaches prevent sensitive timing behavior (as exhibited by the first thread in the example above when it branches on secret) to be observed by other threads. The low-determinism approaches disallow races on public data. The external timing approaches prevent leaks with respect to attackers that may observe real time.

For the goals we have, the external timing-based solutions are sound but too restrictive, as discussed above. Permissiveness is also a concern with low-determinism, where all races on public data are prohibited. This rejects the program above, for example. On the other hand, protection/hiding-based techniques allow accepting the program, given that there is a possibility of runtime support to restrict certain dangerous interleavings. In order for the timing difference of the thread that

branches on *hi* not to influence 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`. This is the choice we follow in the rest of the paper, driven by previous work on interaction between the threads and the scheduler [Russo and Sabelfeld 2006a], which is parametric in a class of schedulers (cf. our goal(iii)).

The goals and the state of the art: security-preserving compilation. Security-preserving compilation is an instance of type-preserving compilation; its aim is to relate security type systems for source programs with security type systems for low-level programs. More formally, the goals of security-preserving compilation are (i) to prove that typable source programs are compiled into typable low-level programs, (ii) in case the target type system requires additional information for type checking, extend compilers to generate certificates that package the required information.

Security-preserving compilation is important in practice because it ensures that applications developed using information-flow aware programming languages are compiled into code that will be analyzed as secure by an information-flow type system for the target language [Barthe et al. 2006]. Security-preserving compilation is essential in the context of Proof Carrying Code [Necula 1997], since it 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, as discussed in the next section. 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 the target level.

Another benefit of security-preserving compilation is that it can reliably protect program implementations while letting programmers safely ignore some security issues that are handled automatically by the compiler [Abadi 1998]. In this paper, we show that internal timing leaks can be handled automatically by a compiler.

Contributions. This paper proposes type-based enforcement methods that provably guarantee secure information flow for multithreaded low-level programs, and security-preserving compilation methods that allow source type systems safely ignore internal timing leaks. On the code consumer side, our type systems can be used for checking the security of programs before running them. On the producer side, the source type systems allow programmers to think about security of multithreaded programs in the same way as for sequential programs; in particular, programmers do not have to know about the existence of internal timing leaks and there are no restrictions on dynamic thread creation at the source level. One implication is that secure source programs that pass security type checking for sequential languages [Volpano et al. 1996] can be securely composed in parallel. This might be counter-intuitive: there are covert channels in the presence of threads, such as internal timing channels [Volpano and Smith 1999], that do not arise in a sequential setting. However, we will show that for a class of security-aware schedulers the compiler will help to enforce a scheduling discipline for the target code so that the

execution of a the compilation of a typable source program is free of internal timing leaks.

A first (minor) contribution of this paper is a new formulation of security-aware schedulers; previous definitions can be found in e.g., [Russo and Sabelfeld 2006a]. Security-aware schedulers take into account the security levels of the program counters of each thread when deciding which thread to execute; to be secure, a security-aware scheduler must correctly hide threads, i.e., it must suspend execution of low threads when another thread has entered a high branch, and there is a potential for an internal timing leak. Thus, schedulers which ignore security information (as a basic round-robin scheduler would do) are insecure; however, we show how to secure the round-robin scheduler by a simple modification that is presented in Section 3. Note that secure schedulers do not introduce unexpected behaviors, but they may disallow certain interleavings. 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., [Smith and Volpano 1998; Volpano and Smith 1999; Smith 2001; 2003; Russo and Sabelfeld 2006a]).

A second contribution of the paper is a sound method for extending information-flow type systems from sequential to multithreaded programs. We start from the intuition that a program is secure if it has no explicit or implicit flows, and no internal timing leaks can arise during its execution. By construction, a secure scheduler prevents internal timing leaks, therefore it is sufficient for a type system to prevent explicit and implicit flows. As information-flow type systems for sequential languages are conceived to prevent exactly such flows, one can hope that it is possible to device provably sound extensions of these type systems to multithreaded programs. In Section 6, we show indeed that it is possible to systematically construct such extensions for type systems that are written in the style of [Barthe et al. 2007a] and enforce noninterference. Modularity is two-fold: first, the framework is parametric in the semantics of sequential programs. The transitions for the sequential part can be arbitrarily extended, which is useful when introducing new sequential features into the underlying programming language. Thus, we have full language-independence from the sequential part. Second, the framework is parametric in the security type system for the sequential part of the language—provided the rules are written in the style of [Barthe et al. 2007a], as mentioned above. Increasing complexity of the underlying language is accompanied by increasing complexity of the typing rules. We have identified sufficient hypotheses that, when satisfied by the type system for the sequential part of the language, guarantee security for the full multithreaded language.

A third contribution of the paper is showing security-preserving compilation. A clear interface (in terms of sufficient hypotheses) determines what needs to be guaranteed by security type preserving compilation for a sequential language in order for the result for type-preserving compilation to hold for the full multithreaded language. In particular, security preserving compilation is also a modular extension of the sequential counterpart.

A final contribution of the paper is an instantiation of the main ingredients of our framework. To illustrate the applicability of the framework, we instantiate it with some scheduler examples. These examples clarify what is expected of a scheduler to

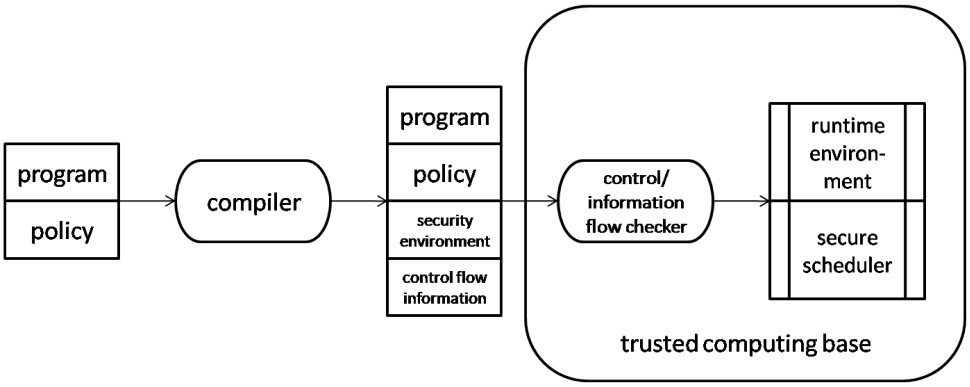

Fig. 1. Proof Carrying Code scenario

prevent internal timing leaks. Also, we give an instantiation of our framework for compiling a simple imperative language into a simple assembly language (featuring an operand stack, conditions, and jumps), and show that the resulting type system is compatible with bytecode verification. The framework is language-independent (any sequential language and type system can be plugged in, as long as appropriate information about joint points can be extracted), and so this instantiation is for illustration only, deliberately leaving richer features of both the source and target language out of the scope. However, we expect our results to extend to languages with objects, methods, and exceptions, for which mechanisms for tracking information flow by security-type systems were recently developed [Barthe et al. 2007a].

Proof Carrying Code scenario. Our approach pushes the feasibility of replacing trust assumptions by type checking for mobile code security one step further, and extends to information flow the principles of typed assembly languages, certifying compilation and Proof Carrying Code by Morrisett, Necula, and others [Morrisett et al. 1999; Necula 1997].

Figure 1 describes the process of producing and verifying a secure application, and illustrates that there is no need to trust the compiler nor the source type system. On the left of the figure, the code producer writes an application that is typable with respect to an information-flow policy. One of the most attractive features of our approach is that the type system needs to make no special provision for concurrency. The application is then transformed to bytecode thanks to an extended compiler that also produces information for tracking implicit flows and internal timing flows: a security environment and additional control-flow information (these concepts are explained in Section 2). The consumer receives the compiled application, the policy, the security environment, and the additional control-flow information. The consumer verifies the correctness of the control-flow information, and that the program is typable w.r.t. the type system. If both verifications succeed, the execution of the program will not reveal information when executed with a secure scheduler. Thus, the *trusted computing base* (TCB) consists of the control-flow information checker, of the type system, and of the scheduler. Note that the

control-flow information checker and the type system are not restricted to compiled programs. However, type-preserving compilation ensures that typable source programs are compiled into typable programs, and that the control-flow information that is generated by the compiler is accepted by the control-flow information checker. Therefore, if the producer generates the security environment and control-flow information following our approach, and the source program is typable, then the consumer will succeed with the consumer-side checks.

Remark. This paper revises and extends an earlier conference version [Barthe et al. 2007] with proofs, explanations, and examples (note that we have renumbered the hypotheses of Section 6 in order to ease reading). More details can be found in the full electronic version [Barthe et al. 2009].

2. MOTIVATING EXAMPLE

The purpose of this section is to illustrate some essential points of our approach, using a program written in the imperative language of Section 7:

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

The policy for such a program is given by a mapping of variables/registers to security levels and a security condition; here the condition is noninterference, i.e., the final values of low variables should not depend on the initial values of high variables. (We discuss different flavors of noninterference that might be included in the policy in Section 4).

In the example, variables hi and hi' store secret data, while lo stores public information. The above program contains a command that branches on secrets and, depending on which branch is taken, takes different timing behavior. Thus, this program suffers from internal-timing leaks: assuming a one-step round-robin scheduler, the last command to assign lo is $lo := 1$ when hi is true; and $lo := 0$ when hi is false.

We start by showing how our approach prevents the compiled program to leak information through such internal timing leaks when executed when a secure scheduler.

The compiler produces the following low-level code:

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

Instruction `push` stores a value into the stack. Instruction `store` moves the top of the stack into a register. Instructions `goto` and `ifeq` represents unconditional and conditional branches, respectively. Instruction `return` finishes the execution of a

thread and returns the value stored on the top of the stack. Instruction `start n` spawns a new thread that starts executing the instruction number `n`. Instructions 2–8 result from compiling the body of the `fork` command, while instructions 10–22 are obtained by compiling the main thread. Instructions 1, 9, and 23 properly plug together the code corresponding to the generated threads.

If the compiled program is executed using an arbitrary scheduler, e.g, a round-robin scheduler, then information is leaked. In order to guarantee that the execution of the program will not leak information, one must extend the compiler so that it propagates to the scheduler information related to hiding to the scheduler. Specifically, the compiler is extended to produce a security environment that assigns a security level to each instruction. To illustrate that in the current example, we show the compiled code where instructions assigned to the high security level are marked with gray. Unmarked instructions are assigned to the low security level.

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

The intuition is that the instructions inside of the `if-then-else` command are assigned to the high security level. More formally, the security environment is built by first computing the security level of the guards of branching expressions in the source program, and then by propagating these levels to the compiled program.

Note that providing a security environment in itself does not guarantee that executing the program will not leak information: if the scheduler ignores the information or uses it incorrectly, executing the program may leak information. However, if the scheduler is itself secure (i.e., it actually suspends threads like the second thread in the example above at the right time), then security is restored, i.e., executing the compiled code will not yield internal timing leaks; as the compiled program does not contain explicit or implicit flows either, it will not leak information.

While the security environment allows to provide information that can be used by a secure scheduler to ensure that the compilation of secure programs is executed securely, the compiler must take additional steps before its result is amenable to security type checking. Indeed, the low-level type system also relies on information about control dependence regions to reject programs with implicit flows, i.e., programs in which a low assignment is taking place in a high branch. Thus, the compiler must also be extended to generate this information (referred to as additional control-flow information in Section 1). In this paper, we capture the necessary information using a (partial) function next from program points to program points that maps every program point deemed high by the security environment to the first reachable low program point—note that one important property of next is precisely to be a function, i.e., the program point, when it exists, is unique. In the example, next assigns 21 to each instruction k that is grey, i.e, $\mathsf{next}(k) = 21$. The next function is in close relation with the control dependence regions used in our

earlier works on bytecode languages.

At this point, the extended compiler has produced all the necessary information for type-checking the compiled code: the security environment, and function next. Type-checking the program is performed on a per-thread basis, with some additional checks to ensure that threads created in high branches do not assign to low variables. In the example, the thread is forked outside of any branching statement, so the latter condition is trivially fulfilled, and we are left to check in isolation the following sequential code fragments:

```
2  push 0
3  store hi'
4  push 0
5  store hi'
6  push 0
7  store lo
8  return
```

```
10  load hi      17  push 0
11  ifeq 15      18  store hi'
12  push 0       19  push 0
13  store hi'    20  store hi'
14  goto 21      21  push 1
15  push 0       22  store lo
16  store hi'    23  return
```

Each fragment can be type-checked successfully using an information flow type system for sequential low-level programs [Barthe et al. 2007a].

Note that the information flow type system of [Barthe et al. 2007a] is in fact a lightweight bytecode verifier, i.e., programs can be checked in one pass using a variant of Kildall's algorithm on a transition function over security states (for the language of Section 7 a security state is simply a stack of security levels). Since information-flow type checking of multithreaded programs is performed on a per thread basis, the information flow type system for multithreaded low-level programs remains compatible with the principles of lightweight bytecode verification. This is a first advantage of making the definition of the information flow type system for multithreaded low-level programs modular in the information flow type system for sequential low-level programs.

However, the question remains whether the information-flow type system for multithreaded programs is sound: informally, the soundness of the type system should follow from: (i) the hypothesis that programs are executed with a secure scheduler, ruling out the possibility of internal timing leaks, and that threads that are created in a high context cannot influence low memory; (ii) the soundness of the type system for sequential programs, ruling out explicit and implicit flows. Section 6 makes this reasoning precise, showing that soundness of the information flow type system for multithreaded programs is a consequence of the soundness of the information flow type system for sequential programs, and of the fact that the next function satisfies some properties. The properties required for soundness are closely related to the properties required for control dependence regions in our earlier works on bytecode languages.

Let us now turn to security-preserving compilation. We have seen with the example that a program is typable if its "sequential parts" (in the case of the example the two framed programs above) are typable, and some local constraints are satisfied. How does it relate to the typability of the source program? If we take

as typing rule

$$\frac{\vdash c : \sigma}{\vdash \mathsf{fork}(c) : \sigma}$$

then we see that the source program is also typable using the rules of [Volpano et al. 1996] for the other constructs. One cannot conclude directly from the typability of the source program that the compiled program is typable. However, one can observe that typability of the source program entails typability of its "sequential parts":

$$(hi' := 0;\ hi' := 0;\ lo := 0);$$
$$\texttt{if}\ hi\ \texttt{then}\ hi' := 0;\ hi' := 0;\ hi' := 0\ \texttt{else}\ hi' := 0;\ lo := 1$$

whose compilation yield the "sequential parts" of the compiled low-level program. Thus, we can appeal to type-preserving compilation for sequential programs [Barthe et al. 2006] to conclude that the compiled multithreaded program is typable. Section 8 makes this reasoning precise, showing that type-preserving compilation can be proved in a modular fashion.

3. 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 deal with covert channels in the presence of multithreading.

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

Definition 3.1 (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 from program points to* Ins*, where* $\mathsf{Ins} = \mathsf{SeqIns} \cup \{\mathsf{start}\ pc\}$ *with* $pc \in \mathcal{P} \setminus \{\text{exit}\}$*. We write $P[i]$ to refer to the instruction of program P at program point 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, pc is not a successor of i. These assumptions help to simplify the formalization in subsequent sections. Notice that any thread can be desugared to a thread with a non-branching start point by using a first instruction with the semantics of a skip or noop instruction. As common, we let $\mapsto^\star$ denote the reflexive and transitive closure of the relation $\mapsto$ (similar notation is used for other relations).

Definition 3.2 (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* [Barthe et al. 2007] that assigns a security level to each program point and is used to prevent *implicit flows* [Denning and Denning 1977]. The security environment is also used by the scheduler to select the thread to execute.

Definition 3.3 (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 3.4 (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 [Russo and Sabelfeld 2006a] 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, threads may be 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 3.5 (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*[1]*, 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 $\mathsf{History}$. We now turn to the definition of a secure scheduler. The definition below is of a more algebraic nature than that of [Russo and Sabelfeld 2006a], 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 3.6 (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* $\ell' = se(s'.\mathsf{pc}(\mathsf{pickt}(s', h')))$*.*

To illustrate how schedulers are expressed with our formalism, we give two examples of round-robin schedulers: an insecure and a secure one.

[1]Throughout the paper, we consistently use the term indistinguishable to mean low-indistinguishable.

Example 3.1. *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 3.2. *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*). For simplicity, we have described a one-step secure scheduler. However, the definition above can be easily extended to schedulers, where threads are scheduled for some fixed number of steps.*

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 for every program point i a local state $\sigma_{\mathrm{init}}(i)$ whose local memory is initialized to default values and whose program counter is 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 3.7 (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 2. 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 2 correspond to non-terminating and terminating sequential steps. In the case of termination, the current thread is removed from the domain of lst. The last rule describes dynamic thread creation caused by the instruction $\mathsf{start}\ pc$. A new thread receives a fresh name *ntid* from $\mathsf{fresht}_{se(i)}$, where $se(i)$ records the security environment at the point of creation. This thread is added to the pool of threads under the name *ntid*. All rules update the history with the current thread id and the security environment of the current instruction. The

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

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

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

Fig. 2. Semantics of multithreaded programs

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

Fig. 3. Typing rules

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

4. 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 4.1 (Noninterfering program)**.** *Let* $\sim_g$ *be an indistinguishability relation on global memories. A program* P *is* noninterfering *if for all memories* $\mu_1, \mu_2, \mu_1', \mu_2'$*:*

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

This policy is *termination-insensitive* noninterference [Volpano et al. 1996], where leaks via (non)termination are ignored. This is a common baseline policy for much of information-flow work (e.g., [Volpano et al. 1996; Pottier and Simonet 2003; Banerjee and Naumann 2005; Barthe et al. 2006; Russo and Sabelfeld 2006a]) and the target policy for the mainstream information-flow tools Jif [Myers et al. 2001], FlowCaml [Simonet 2003], and the SPARK Examiner [Barnes and Barnes 2003; Chapman and Hilton 2004]. An example of a termination leak can be found in program (`while` h `do skip`)$; l := 42$. Upon observing that the final value of l is 42, the attacker deduces that h was 0. In a batch-job model, at most one bit can be leaked per run via the (non)termination channel. The motivation for ignoring

this channel in the above-mentioned work is permissiveness, which allows accepting loops with secret guards without the need for termination analysis. While we do not foresee fundamental difficulties for adapting our framework to *termination-sensitive* noninterference (where program (non)termination may not depend on secrets), we also settle for termination-insensitive noninterference.

5. 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 6, 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 7, we instantiate the framework to a simple assembly language.

5.1 Assumptions on type system for sequential programs

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

Typing rules are used to establish a notion of typable program, which ensures that runs of typable programs verify at each step the constraints imposed by the typing rules.

Definition 5.1 (Typable sequential program)**.** *A sequential program P is typable w.r.t. type $\mathcal{S} : \mathcal{P} \to \mathsf{LType}$ and security environment se, written $se, \mathcal{S} \vdash P$ if*

(1) $\mathcal{S}_1 = \mathrm{T}_{\mathrm{init}}$ (the initial program point is mapped to the initial local type); and

(2) for all $i \in \mathcal{P}$ and $j \in \mathcal{P}$ $i \mapsto j$ implies that there exists $\mathrm{S} \in \mathsf{LType}$ such that $se, i \vdash_{\mathrm{seq}} \mathcal{S}_i \Rightarrow \mathrm{S}$ and $\mathcal{S}_j \leq \mathrm{S}$,

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

The sequential type system is assumed to satisfy further properties e.g. unwinding lemmas, that have already been established for some specific languages and that are formulated precisely in Section 6.

5.2 Type system for multithreaded programs

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

Definition 5.2 (Typable multithreaded program)**.** *A concurrent program P is typable w.r.t. type $\mathcal{S} : \mathcal{P} \to \mathsf{LType}$ and security environment se, written $se, \mathcal{S} \vdash P$, if*

(1) $\mathcal{S}_i = \mathrm{T}_{\mathrm{init}}$ for all initial program points i of P (initial program point of main threads or spawn threads); and

(2) for all $i \in \mathcal{P}$ and $j \in \mathcal{P}$: $i \mapsto j$ implies that there exists $\mathrm{S} \in \mathsf{LType}$ such that $se, i \vdash \mathcal{S}_i \Rightarrow \mathrm{S}$ and $\mathcal{S}_j \leq \mathrm{S}$.

6. SOUNDNESS

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

Theorem 6.1. *If the scheduler is secure and* $se, \mathcal{S} \vdash P$*, then* P *is noninterfering, provided Hypotheses 1-6 hold.*

Throughout this section, we assume that P is a typable program, i.e., $se, \mathcal{S} \vdash P$, and that the scheduler is secure. Moreover, we state some general hypotheses that are used in the soundness proofs. We revisit these hypotheses in Section 7 and show how they can be fulfilled.

State equivalence. In order to prove noninterference, we rely on a notion of state equivalence. The definition is modular, in that it is derived from an equivalence between global memories $\sim_g$ and a partial equivalence relation $\sim_l$ between local states. (Intuitively, partial equivalence relations on local and global memories represent the observational power of the adversary.) In comparison to [Barthe et al. 2007a], 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 6.1 (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 6.2. *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* $\langle \lambda_1, \mu_1 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_1', \mu_1' \rangle$ *and* $\langle \lambda_2, \mu_2 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_2', \mu_2' \rangle$*, 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 $\sigma_{\text{init}}(i) \sim_l \sigma_{\text{init}}(i)$.*

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 6.1 (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 $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$, and $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, then $s' \sim t'$ and $h_{s'} \overset{\mathsf{hist}}{\sim} h_{t'}$.*

We now turn to the second, so-called step consistent, unwinding lemma. The lemma relies on the hypothesis that the current local memory is high, i.e., invisible for 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 $\langle \lambda_1, \mu_1 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_1', \mu_1' \rangle$ and $High^{\mathsf{lmem}}(\lambda_1)$ and $H(i)$, then $\lambda_1' \sim_l \lambda_2$ and $\mu_1' \sim_g \mu_2$.*

Lemma 6.2 (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 $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ 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.

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

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

Note that the hypothesis is formulated w.r.t. concurrent states and concurrent execution. However, it is immediate to derive the above hypothesis from its restriction to sequential states and sequential execution.

We also need an hypothesis about visibility by the attacker:

Hypothesis 5 (High hypotheses)**.**

(1) For every initial program point i, we have $High^{\mathsf{lmem}}(\sigma_{\text{init}}(i))$.

(2) If $\langle \lambda, \mu \rangle \rightsquigarrow_{\text{seq}} \langle \lambda', \mu' \rangle$ 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 $\lambda_1 \sim_l \lambda_2$.

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

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

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

Fig. 4. Source and target language

The next *function.* Finally, the soundness proof relies on the existence of a function next that satisfies several properties. Intuitively, next computes for any high program point its minimal observable successor, i.e., the first program point with a low security level reachable from it. If executing the instruction at program point i can result in a hidden thread (high if or high while), then $\mathsf{next}(i)$ is the first program point such that $i \mapsto^\star \mathsf{next}(i)$ and the thread becomes visible again. The existence of the next function is closely related to control dependence regions, which are discussed in Section 7.1.

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

NePd) $\mathsf{Dom}(\mathsf{next}) = \{i \in \mathcal{P} | H(i) \wedge \neg AH(i)\}$
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) $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge k \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = k$

Intuitively, properties **NeP1**, **NeP2**, and **NeP3** ensure that the next of instructions within an outermost high conditional statement coincides with the junction point of the conditional; in addition, properties **NeP1**, **NeP2**, and **NeP4** ensure that the next of instructions within an outermost high loop coincides with the exit point of the loop. Note that in **NeP2**, it would be equivalent to replace $j \notin \mathsf{Dom}(\mathsf{next})$ by $L(j)$; and in **NeP3**, it would be equivalent to replace $i \notin \mathsf{Dom}(\mathsf{next})$ by $L(i)$.

7. INSTANTIATION

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

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

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

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

Fig. 5. Transfer rules

of next satisfies the properties required in Section 6. Finally, we conclude with a discussion about how a similar instantiation can be done for the JVM.

7.1 Sequential part of the language

The instantiation requires us to define the semantics and a type system to enforce noninterference for the sequential primitives of the language. On the semantics side, we assume that a local state is a pair $\langle os, pc \rangle$, where os is an operand stack, i.e., a stack of values, and pc is a program counter, whereas a global state μ is a map from variables to values. The operational semantics is standard and therefore we omit it. We also define $\sigma_{\mathrm{init}}(pc)$ 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 $\mathrm{T}_{\mathrm{init}}$ be the empty stack of security levels. Typing rules are summarized in Figure 5, where $\mathsf{lift}_k(st)$ denotes the point-wise extension of $\lambda k'.\ k \sqcup k'$ to stacks of security levels, and $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ denotes the region of branching points. We express the chosen security policy by assigning a security level $\Gamma(x)$ to each variable x. Then, we say that a program p is typable, written $\vdash_{ssl} p$, if there exists se, reg, jun and $\mathcal{S}$ such that $se, \mathcal{S} \vdash p$ (as in Definition 5.1) and additionally, $(\mathsf{reg}, \mathsf{jun})$ satisfy the so-called SOAP properties[2].

The definition of state equivalence is inspired from [Barthe and Rezk 2005]: two global memories are indistinguishable iff they coincide on all low variables. Equivalence between local memories is defined relative to a mapping of program points to stack types, using the notion of operand stack indistinguishability used in [Barthe and Rezk 2005]. Formally, we instantiate the definitions of local and global state equivalence, and of high stacks as follows:

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

where $\overset{\mathsf{os}}{\sim}$, $\overset{\mathsf{vmap}}{\sim}$, and highos are defined as in [Barthe and Rezk 2005]:

$\mu_1 \overset{\mathsf{vmap}}{\sim} \mu_1'$ iff $\mu_1(x) = \mu_1'(x)$ for all $x \in \mathcal{V}$ such that $\Gamma(x) \leq k$

[2]Thus, the notion of typable program in the instantiation is an exact instance of Definition 5.1. It is possible to amend this minor mismatch, but clearer to abide to presentation in [Barthe and Rezk 2005].

$\mathsf{highos}(os, \mathrm{S})$ iff os and S have the same length n, and $\mathrm{S}(i) \not\leq k$ for all $1 \leq i \leq n$

$s \overset{\mathsf{os}}{\sim}_{\mathrm{S},\mathrm{S}'} s'$ is defined by the clauses

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

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

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

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

We conclude this section by showing that all hypotheses, except Hypotheses 6, follow immediately from definitions, or from the results of [Barthe and Rezk 2005]. Note that the latter rely on some assumptions about control dependence regions in programs. Essentially, these regions represent an over-approximation of the range of branching points. This concept is formally introduced by the functions $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ and $\mathsf{jun} : \mathcal{P} \rightharpoonup \mathcal{P}$, which respectively compute the control dependence region and the junction point for a given instruction. Both functions need to satisfy some properties in order to guarantee noninterference in typable programs. These properties, which are known as SOAP properties [Barthe and Rezk 2005], are given in the Appendix, and can be guaranteed by compilation.

Lemma 7.1. *Hypotheses 1, 2, 3, 4 and 5 hold for all programs p such that $\vdash_{ssl} p$.*

7.2 Concurrent extension

The semantics of concurrent programs is derived from the semantics of sequential instructions as prescribed by Definition 3.7. Likewise, the sequential type system in Figure 5 is extended by the typing rules presented in Figure 3 to consider concurrent programs.

The soundness of the type system for concurrent programs holds for programs P for which there exists a function next satisfying the **NeP** properties. In the setting of certifying compilation, the code consumer receives this function, together with the security environment, and must check that the next function complies with the properties of Hypothesis 6. This check can be performed using ideas of control dependence regions. In the sequel, we focus on the existence of a next function with the expected properties. There are two possible strategies for showing the existence of a function. The more general strategy would be to derive Hypothesis 6 from assumptions on sequential programs, and show that compilers which generate sequential target programs that satisfy these assumptions can be naturally extended to compilers which generate concurrent target programs for which Hypothesis 6 holds. To conclude, one would just need to prove that in the case of our instantiation the compiler for sequential programs does indeed verify with the required assumptions. While this strategy is clearly in line with the approach followed in this paper, it requires to introduce a significant amount of material. For this reason, we follow a more direct strategy and prove that compiled programs verify Hypothesis 6. The proof method is closely related to our earlier work on type-preserving compilation [Barthe et al. 2006].

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

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

Fig. 6. Compilation function

Similarly to the technique of [Barthe et al. 2006], 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 7.1 (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 [Barthe et al. 2006], control dependence regions for low-level code are defined based on the function sregion and a compilation function. For a complete source program c, we define the compilation $\mathcal{C}(c)$ in Figure 6. We use symbol # to compute the length of lists. Symbol :: is used to insert one element to a list or to concatenate two existing lists. The current program point in a program is represented by pc. The function $\mathcal{C}(c)$ calls the auxiliary function $\mathcal{S}$ which returns a pair of programs. The first component of that pair stores the compiled code of the main program, while the second one stores the compilation code of spawned threads. We now define control dependence regions for assembly code and respective junction points.

Definition 7.2 (function tregion)**.** *For a branching instruction $[c]^n$ in the source code, tregion(n) is defined as the set of instructions obtained by compiling the commands $[c']^{n'}$, where $n' \in$ sregion(n). Moreover, if c is a while loop, then $n \in$ tregion(n) as well as the instructions obtained from compiling the guard of the loop. Otherwise, the* goto *instruction after the compilation of the else-branch also belongs to* tregion(n).

Junction points are computed by the function jun. The domain of this function

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

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

$$\frac{\vdash e : H \qquad \vdash_H c \,:\, E, F \qquad E(n) = F(n) = H}{\vdash_H [\mathsf{while}\ e\ \mathsf{do}\ c]^n_H \,:\, E, F}$$

$$\frac{\vdash e : H \qquad \vdash_H c : E, F \qquad \vdash_H c' : E, F \qquad E(n) = F(n) = H}{\vdash_H [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_H \,:\, E, F}$$

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

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

Top-H-While
$$\frac{\vdash e : H \qquad \vdash_H c \,:\, E, F \qquad E(n) = L \qquad F(n) = H}{\vdash_L [\mathsf{while}\ e\ \mathsf{do}\ c]^n_H \,:\, E}$$

Top-H-Cond
$$\frac{\vdash e : H \qquad \vdash_H c : E, F \qquad \vdash_H c' : E, F \qquad E(n) = L \qquad F(n) = H}{\vdash_L [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_H \,:\, E, F}$$

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

consist of every branching point in the program. We define jun as follows:

Definition 7.3 (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 6. 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 [Barthe et al. 2006] 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 $\vdash_L c : E, F$, if its command part is typable with respect to E and F according to the rules given in Figure 7. The typing judgment has the form $\vdash_\alpha [c]^n_{\alpha'} : E, F$, where E and F are functions from

labels to security levels. Function E and F play the role of security environment for the source code which easily allows to define the security environment for the target code (see Definition A.7 in Appendix). Specifically, the security level $E(n)$ can be thought as the security level of the program counter (pc) when running instruction number n. Function F agrees on every label that does not correspond to an outermost branch involving secrets on the guard. This difference indicates that low level instructions generated from compiling outermost branches should also be considered with a program counter H. Function E is defined as in [Barthe et al. 2006], while function F can be seen as a modular addition to the type system presented by the authors. We omit writing E and F in type judgments when expressing properties only related with source code. Variable α denotes if c is part of a branching instruction that branches on secrets (H) or public data (L). 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 $\alpha = L$ and $\alpha' = H$. Moreover, the type system prevents *explicit* (via assignment) and *implicit* (via control) flows [Denning and Denning 1977]. To this end, the type system enforces the same constraints as standard security type systems for sequential languages (e.g., [Volpano et al. 1996]). 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 7.4 (function next)**.** *For every branching point c in the source program such that $\vdash_L [c]^n_H$, we have that $\forall k \in \mathsf{tregion}(n).\mathsf{next}(k) = \mathsf{jun}(n)$.*

This definition satisfies the properties from Section 6, as shown by the following lemma.

Theorem 7.1. *Definition 7.4 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 7.1 (Soundness of the instantiation)**.** *The derived type system guarantees noninterference for multithreaded assembly programs.*

7.3 The compilation example (revised)

We now illustrate how our definitions and intermediate type system provide the necessary information to build the security environment *se* and function **next** for the example given in Section 2.

The producer labels the source code in such a way that instructions in the compiled code match the instructions that generated them at the source level. More

specifically, the source code is labeled as follows:

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

Observe that instructions related to stack operations are not associated to instructions at the source level.

The program has only one branching point. The compiler consequently obtains $\mathsf{sregion}(11) = \{16, 18, 20, 13\}$, $\mathsf{tregion}(11) = \{12, 13, 14, 15, 16, 17, 18, 19, 20\}$, and $\mathsf{jun}(11) = 21$ by applying Definitions 7.1, 7.2, and 7.3, respectively.

By applying the intermediate type system in Figure 7 to the labeled source, the compiler also obtains that functions E and F are defined as follows.

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

By Definition A.7 (see Appendix), the security environment se for the compiled code is determined as follows.

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

Observe that the instructions inside of the `if-then-else` command have H as their security environment. Finally, the compiler obtains function next by applying Definition 7.4. Specifically, $\forall k \in \mathsf{tregion}(11).\mathsf{next}(k) = \mathsf{jun}(11) = 21$ since $\vdash_L [\texttt{if}\ hi\ \texttt{then}\ hi' := 0;\ hi' := 0;\ hi' := 0\ \texttt{else}\ hi' := 0;]^{11}_H$ by the type derivation of the program on the intermediate type system.

7.4 Discussion: towards information-flow type systems for the Java Virtual Machine

The modular proof techniques developed in this paper can be extended to more realistic languages. One original motivation for this work is in fact to build an information-flow bytecode verifier that is able to guarantee noninterference for a concurrent fragment of the JVM (Java Virtual Machine). Since an existing type system for information flow checking is compatible with bytecode verification [Barthe et al. 2007a], then also the concurrent type system that extends it as prescribed in Definition 5.2 shall also be compatible with it. Furthermore, 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 the JVM with a secure scheduler that keeps track of high and low threads as a part of an application's own state (cf. [Tsai et al. 2007]).

Thus, the main issue is to show that our modular approach can accommodate language features that exist in Java but lack in the minimalist imperative language of Section 7: synchronization, objects, exceptions and methods. Dealing with these features is outside of the scope of this paper, and left for future work. Nevertheless we discuss each of them below.

Synchronization. The semantics of the multithreaded JVM obtained by the method described in Section 3 only partially reflects the JVM specification. In particular, it ignores object locks, which are used to perform synchronization throughout program execution. Thus, in order to instantiate our results to the JVM it would be necessary to add synchronization in the framework, and to modify the scheduler so that it only picks threads among active ones. Including synchronization is an interesting topic for future work.

Objects. In order to define state equivalence in presence of objects, one can define indistinguishability relative to a bijection between object references: to our best knowledge, this approach was suggested in [Banerjee and Naumann 2005] and was later adopted in other works, including [Barthe et al. 2007a]. We do not expect any particular difficulty in extending our modularity results to the object fragment of the language considered by [Barthe et al. 2007a], and thus in achieving a sound type system for a concurrent object oriented language.

One potential issue with objects is that many JVMs do not provide an opaque implementation of pointers, i.e., they make it possible to cast a reference to a natural number. Hedin and Sands [Hedin and Sands 2006] have observed that opaque pointers are dangerous when specific API are called, and proposed a type system that remains sound in presence of non-opaque pointers. One could also consider instantiating our framework to a bytecode variant of their type system.

Methods. Methods are largely orthogonal to multithreading issues, and our framework is applicable to languages with method calls. There is a catch however: the semantics of programs with method calls is often given in mix-step style, i.e., method calls are executed in one-step; for example, [Barthe et al. 2007a] adopts this style of semantics. One problem with mix-step semantics is that it is not compatible with concurrency: indeed, the semantics of multithreaded programs that one obtains by applying the rules of Figure 2 to a mix-step semantics is inaccurate.

To remedy to this problem, one must adopt a small-step semantics, and prove the unwinding lemmas for this new semantics. It is immediate to give a small-step semantics for methods, but it complicates the notion of state considerably, as states must now carry a stack frame storing the local state of all method calls currently executing. Furthermore, indistinguishability must be modified to accommodate this new notion of state. The overall development is substantially more complicated than the one in [Barthe et al. 2007a], but the proofs go through—in earlier unpublished work, we gave a proof of noninterference of the JVM, including methods, using a small-step semantics.

Exceptions. Multiple exceptions for analysis of information flow are handled in JIF [Myers et al. 2001] for Java, and in [Barthe et al. 2007a] for unstructured languages. Type preserving compilation with multiple exceptions from Java to

bytecode appears in [Rezk 2006]. As explained in [Myers 1999], static treatment of exceptions as discriminated unions would result in an acceptable loss of precision: if all code for different exception handlers is included in the same control dependence region then many programs will be deemed insecure by the static checker. Hence a precise treatment of exceptions with a static checker for information flow that avoid implicit flows should take account of different termination paths and distinguish control dependence regions according to the path to which instructions belong.

In order to adapt our results to multithreaded Java with multiple exceptions we need to adapt the sequential part of the language to consider different termination paths, in particular control dependence regions should distinguish dependencies caused by normal or abnormal termination due to different kind of exceptions (see e.g., [Barthe et al. 2007a]).

Exceptions in multithreaded Java are handled and propagated as in sequential Java, except maybe by the interrupt mechanism that is not a concern by the termination-insensitive noninterference policy that we consider here.

Exceptions that a thread may throw are handled by the stack frame corresponding to the thread, if handled. Threads with exception handlers that execute code that is observable, can still cause internal timing leaks. Consider for example an internal timing leak caused by a simple two-threaded program, where hi is a high variable and $size$, lo, and lo' are low variables:

$\texttt{if}\ hi\ \{\texttt{sleep}(100)\};$
$\texttt{try}\ \{\texttt{if}\ (lo' < size)\ \{\texttt{throw}\ (\texttt{new}\ \textbf{SizeException})\}\}\ \texttt{catch}\ (\textbf{SizeException}\ se)\ \{lo := 1\}$
$\parallel \texttt{sleep}(50); lo := 0$

The first thread should be considered as hidden, even if without throwing exceptions it only executes high code. Otherwise, if an exception is thrown the value of variable lo reveals the value of variable hi.

Threads with exception handlers containing low code, should be treated as hidden threads, exactly as defined in our framework without exceptions.

Hence, our hypothesis on the scheduler should remain the same, but the tracking of implicit flows for the sequential part of the language should implement a fine-grained treatment of exceptions as shown in previous works mentioned above.

Java Memory Model. Another point is that the semantics of the multithreaded JVM obtained by the method described in Section 3 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.

In addition, Java semantics is defined relative to a relaxed memory model, which is not captured by our interleaving semantics except for the cases where programs do not have races. We also discuss this issue at the end of the paragraph.

8. TYPE PRESERVING COMPILATION

Type-preserving compilation is an essential tool to ensure that applications developed using information-flow aware programming languages are compiled into code that will be analyzed as secure by an information-flow type system for the target language. The purpose of this section is to prove a modular type-preservation re-

sult for concurrent programs, under the assumption that type-preservation holds for sequential programs. Then, we instantiate our result with the source and target languages of the previous section. The instantiation 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., [Volpano et al. 1996]), the compilation of typable programs is guaranteed to be typable at low-level.

8.1 Modular proof

We assume given a source language, an information flow type system $\vdash_s$ for sequential source programs, a compiler $\mathcal{C}$, and an information flow type system $\vdash_t$ built from an information flow type system for sequential programs as prescribed in Definition 5.2. The goal is to prove that programs that are typable with $\vdash_s$ are compiled into programs that are typable with $\vdash_t$. For the purpose of this section, we hide the fact that the compiler might need to generate additional information for the target type system, by allowing "target programs" to be programs extended with additional information.

Our starting hypothesis is that the compiler preserves typability of sequential programs.

Hypothesis 7. *For all sequential source programs p, if $\vdash_s p$ then $\vdash_t \mathcal{C}(p)$.*

Now, consider an extended source language with a fork operator. We extend the type system to parallel programs by defining a mapping T_s from concurrent to source programs; note that the transformation is not assumed to preserve semantics. Rather, the transformation captures at an abstract level the typing rule for forking; see the next paragraph for an example.

Definition 8.1. *For all concurrent programs, $\vdash_s p$ iff $\vdash_s T_s(p)$.*

Since the concurrent type system is meant to extend the sequential type system, we require that the transformation T_s acts as the identity on source programs. Besides, we assume given a matching transformation T_t at target level, and assume that the transformation commutes with compilation.

Hypothesis 8. *For all concurrent source programs p, $T_s(p)$ is sequential; furthermore $T_s(p) = p$ whenever p is sequential. Finally, for all concurrent source programs p, $\mathcal{C}(T_s(p)) = T_t(\mathcal{C}(p))$.*

To conclude, we must assume that at target level typability of transformed programs entails typability of initial programs.

Hypothesis 9. *For all concurrent target programs p, if $\vdash_t T_t(p)$ then $\vdash_t p$.*

Under these hypotheses, one can prove type-preserving compilation.

Theorem 8.1 (Type-preserving compilation)**.** *For all concurrent source programs p, if $\vdash_s p$ then $\vdash_t \mathcal{C}(p)$.*

Proof. Assume $\vdash_s p$. Then $\vdash_s T_s(p)$ by definition, and by Hypothesis 8, $T_s(p)$ is sequential. Thus, by Hypothesis 7, we have $\vdash_t \mathcal{C}(T_s(p))$. By Hypothesis 8, $\mathcal{C}(T_s(p)) = T_t(\mathcal{C}(p))$, and so $\vdash_t T_t(\mathcal{C}(p))$. By Hypothesis 9, we conclude $\vdash_t \mathcal{C}(p)$. □

8.2 Instantiation to imperative language

In order to apply Theorem 8.1, we must prove that Hypotheses 7, 8 and 9 hold. Hypothesis 7 follows from [Barthe et al. 2006], where compilation of sequential Java programs is established.

For Hypothesis 8, we must first define T_s and T_t:

T_s is defined recursively; all clauses, except for fork, simply propagate the transformation; the clause for fork is

$$T_s(\texttt{fork}(c)) = \texttt{if True then skip else } T_s(c)$$

(A simpler alternative is to replace $\texttt{fork}(c)$ by c, but defining the corresponding transformation at target level is cumbersome.)

T_t replaces every instruction start pc by the two instructions (push 1) :: (ifeq pc'). Since the transformation introduces new instructions, pc' needs to be recomputed from pc accordingly.

By construction, the transformations commute with compilation, thus Hypothesis 8 holds. Moreover, note that the definition of $\vdash_s p$ as given in Definition 8.1 is equivalent to the type system of [Volpano et al. 1996] with additional rule for fork:

$$\frac{\vdash c : k}{\vdash \texttt{fork}(c) : k}$$

where k is a security level.

To conclude, one must proves Hypothesis 9, i.e., that $se, \mathcal{S} \vdash T_t(p)$ implies $se, \mathcal{S} \vdash p$. This is a rather direct consequence of the definition of typability and of the typing rules.

9. 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 [Barthe and Rezk 2005; Genaim and Spoto 2005; Medel et al. 2005; Barthe et al. 2006; Barthe et al. 2007a; Barthe et al. 2007]. Nevertheless, the present work (and its predecessor [Barthe et al. 2007]) 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 [Russo and Sabelfeld 2006a] in order to control internal timing leaks. The interaction is modeled by *hide* and *unhide* primitives that communicate to the scheduler whether a thread's timing behavior should be "hidden". In this paper, there is no need for explicit hide/unhide primitives because scheduler is driven by the security environment. If a thread is inside of a conditional with a high guard, then it executes in a high security environment and thus its timing behavior is automatically hidden from threads that run in a low security environment.

Other approaches [Smith and Volpano 1998; Volpano and Smith 1999; Smith 2001; 2003] to handling internal timing rely on $\texttt{protect}(c)$ which, by definition, hides the internal timing of command c. It is not difficult to implement protect()

under a cooperative scheduler [Russo and Sabelfeld 2006b; Tsai et al. 2007]). For a preemptive scheduler, our work can be (very roughly) interpreted as providing realistic means of implementing `protect`(). In addition, we allow more interleavings (several "protected" threads might execute without blocking each other), and we do not assume a specific scheduler but parametrize in any deterministic scheduler that is secure. It is possible to prevent internal timing leaks by spawning dedicated threads for computations that involve secrets and carefully synchronizing the resulting threads [Russo et al. 2007]. Yet other approaches prevent internal timing leaks in code by disallowing any races on public data [Zdancewic and Myers 2003; Huisman et al. 2006; Terauchi 2008]. One implication is that some programs that do not involve secrets (as $lo := 0 \parallel lo := 1$, where lo is a public variable) are considered insecure. Still other approaches prevent internal timing by disallowing low assignments after high branching [Boudol and Castellani 2002; Almeida Matos 2006]. Less related work [Agat 2000; Sabelfeld and Sands 2000; Sabelfeld 2001; Sabelfeld and Mantel 2002; Köpf and Mantel 2006] considers external timing, where the attacker can use a clock to measure computation time. This work considers a more powerful attacker, and, as a price paid for security, disallows loops with secret guards.

Further afield, different flavors of possibilistic noninterference have been explored in process-calculus settings [Honda et al. 2000; Focardi and Gorrieri 2001; Ryan 2001; Honda and Yoshida 2002; Pottier 2002], but without considering the impact of scheduling. Most recently, van der Meyden and Zhang [van der Meyden and Zhang 2008] have investigated how the choice of a scheduler can affect security definitions in an abstract automata-based setting. However, they leave enforcement mechanisms and treatment of dynamic thread creation unaddressed. For additional related work, we refer to an overview of language-based information-flow security [Sabelfeld and Myers 2003].

10. 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.

Acknowledgments

This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905 and HATS project FP7-231620 and in part by the Swedish research agencies SSF and VR.

REFERENCES

Abadi, M. 1998. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming (ICALP '98)*. Lecture Notes in Computer Science, vol. 1443. Springer-Verlag, 868–883.

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

Almeida Matos, A. 2006. Typing secure information flow: declassification and mobility. Ph.D. thesis, Ecole Nationale Supérieure des Mines de Paris.

Banerjee, A. and Naumann, D. A. 2005. Stack-based access control and secure information flow. *Journal of Functional Programming 15,* 2 (Mar.), 131–177.

Barnes, J. and Barnes, J. 2003. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

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

Barthe, G., Pichardie, D., and Rezk, T. 2007b. A certified lightweight non-interference Java bytecode verifier. Tech. rep., INRIA. Extended version of [Barthe et al. 2007a].

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

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

Barthe, G., Rezk, T., and Naumann, D. 2006. 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)*. IEEE Computer Society, 230–242.

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

Barthe, G., Rezk, T., Russo, A., and Sabelfeld, A. 2009. Security of multithreaded programs by compilation. Full version. Tech. rep. Located at `http://www.cse.chalmers.se/~russo/tissecfull.pdf`.

Boudol, G. and Castellani, I. 2002. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science 281,* 1, 109–130.

Chapman, R. and Hilton, A. 2004. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters 24,* 4, 39–46.

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

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

Genaim, S. and Spoto, F. 2005. Information Flow Analysis for Java Bytecode. In *Proceedings of VMCAI'05*, R. Cousot, Ed. LNCS, vol. 3385. Springer-Verlag, 346–362.

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

Hedin, D. and Sands, D. 2006. Noninterference in the presence of non-opaque pointers. In *Proceedings of CSFW'06*. IEEE Computer Society Press, 255–269.

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

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

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

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

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

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

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

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From System F to typed assembly language. *ACM TOPLAS 21,* 3 (May), 528–569.

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

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. 2001. Jif: Java information flow. Software release. `http://www.cs.cornell.edu/jif`.

Necula, G. C. 1997. Proof-carrying code. In *Proc. ACM Symp. on Principles of Programming Languages*. 106–119.

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

Pottier, F. and Simonet, V. 2003. Information flow inference for ML. *ACM TOPLAS 25,* 1 (Jan.), 117–158.

Rezk, T. 2006. Verification of confidentiality policies for mobile code. Ph.D. thesis, Université de Nice Sophia-Antipolis.

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

Russo, A. and Sabelfeld, A. 2006a. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Symposium*. 177–189.

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

Ryan, P. 2001. Mathematical models of computer security—tutorial lectures. In *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Eds. LNCS, vol. 2171. Springer-Verlag, 1–62.

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

Sabelfeld, A. and Mantel, H. 2002. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*. LNCS, vol. 2477. Springer-Verlag, 376–394.

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

Sabelfeld, A. and Sands, D. 2000. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Symposium*. 200–214.

Simonet, V. 2003. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`.

Smith, G. 2001. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*. 115–125.

Smith, G. 2003. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Symposium*. 3–13.

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

Terauchi, T. 2008. A type system for observational determinism.

Tsai, T. C., Russo, A., and Hughes, J. 2007. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium.*

van der Meyden, R. and Zhang, C. 2008. Information flow in systems with schedulers.

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

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

Zanardini, D. 2006. Abstract non-interference in a fragment of java bytecode. In *SAC*, H. Haddad, Ed. ACM, 1822–1826.

Zdancewic, S. and Myers, A. C. 2003. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Symposium.* 29–43.

A. APPENDIX

Proof of soundness of concurrent type system

Proofs of unwinding lemmas.

Lemma 6.1 (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 $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$, and $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, then $s' \sim t'$ and $h_{s'} \overset{\mathsf{hist}}{\sim} h_{t'}$.

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

(1) The instruction to be executed is a sequential instruction. By definition of the semantics: $\langle s(ctid), s.\mathsf{gmem}\rangle \rightsquigarrow_{\text{seq}} \langle s'(ctid), s'.\mathsf{gmem}\rangle$ and $\langle t(ctid), t.\mathsf{gmem}\rangle \rightsquigarrow_{\text{seq}} \langle t'(ctid), t'.\mathsf{gmem}\rangle$. By hypothesis, we have $s \overset{\mathsf{gmem}}{\sim} t$ and $s(ctid) \sim_l t(ctid)$. Thus by the sequential LR unwinding hypothesis, we have $s' \overset{\mathsf{gmem}}{\sim} t'$ and $s'(ctid) \sim_l t'(ctid)$. Since $s'.\mathsf{lowT} = t'.\mathsf{lowT}$, we conclude that $s' \overset{\mathsf{lmem}}{\sim} t'$ and hence $s' \sim t'$ by definition of state equivalence.

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

□

Lemma 6.2 (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 $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $s'.\mathsf{lowT} = t.\mathsf{lowT}$, then $s' \sim t$ and $h_{s'} \overset{\mathsf{hist}}{\sim} h_t$.

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

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

and from $s'.\mathsf{lowT} = t.\mathsf{lowT}$ that $s' \overset{\mathsf{lmem}}{\sim} t$, and that $s' \sim_g t$ by definition of state equivalence.

(2) The instruction to be executed is of the form $\mathsf{start}\ pc$. By the hypotheses and the definition of state equivalence, it is sufficient to notice that the created thread is not observable, i.e., $H(pc)$ which follows from $se(i) \leq se(pc)$ by typability, and $H(i)$.

□

Lemma 7.1. Hypotheses 1, 2, 3, 4 and 5 hold for all programs p such that $\vdash_{ssl} p$.

Proof. *Hypothesis 1.* is an instance of the low lemma of [Barthe and Rezk 2005]. The lemma can be formulated as:

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

where

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

Indeed, assume that $se, \mathcal{S} \vdash P$ and define $S = S' = \mathcal{S}_i$. Applying the lemma, we conclude that $t \overset{\bullet}{\sim}_{T,T'} t'$ for $T \leq \mathcal{S}_j$ and $T' \leq \mathcal{S}_{j'}$, where $j = t.\mathsf{pc}$ and $j' = t.\mathsf{pc}'$. There are two cases to consider: either $j = j'$, in which case we can apply the double monotony lemma, see [Barthe et al. 2007b], to conclude, or $j \neq j'$, in which case $\mathcal{S}_j$ and $\mathcal{S}_{j'}$ are high, in which case we conclude by definition of operand stack indistinguishability.

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

Hypothesis 3. is an instance of the high lemma of [Barthe and Rezk 2005]. The lemma can be formulated as:

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

Indeed, assume that $se, \mathcal{S} \vdash P$ and define $S = \mathcal{S}_i$. Applying the lemma, we conclude that $t \overset{\bullet}{\sim}_{T,S'} s'$ for $T \leq \mathcal{S}_j$, where $j = t.\mathsf{pc}$. We can apply the single monotony lemma, see [Barthe et al. 2007b], to conclude.

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

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

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

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

Fig. 8. Auxiliary execution relations

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

□

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

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

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

Lemma A.1. *If $P, \mu_1 \Downarrow \mu_1'$, then $s_{\text{init}}(\mu_1), \epsilon^{\mathsf{hist}}(\rightsquigarrow^{\mathsf{vis}}_{\text{conc}})^{\star} s$ with $s.\mathsf{lowT} = \emptyset$ and $s.\mathsf{gmem} = \mu_1'$.*

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

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

Lemma A.2. *If $s, h_s \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} s', h_{s'}$ and $s \sim t$ and $s \overset{\mathsf{pc}}{\sim} t$ then $s' \sim t$ and $h_s \overset{\mathsf{hist}}{\sim} h_{s'}$, and $s' \overset{\mathsf{pc}}{\sim} t$, and $\mathsf{next}(s) = \mathsf{next}(s')$.*

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

If $s, h \rightsquigarrow_{\text{conc}} s', h'$. Since $s, h_s \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} s', h_{s'}$, we have $s.\mathsf{lowT} = s'.\mathsf{lowT}$; besides, $s.\mathsf{lowT} = t.\mathsf{lowT}$ as $s \overset{\mathsf{pc}}{\sim} t$. Hence $s'.\mathsf{lowT} = t.\mathsf{lowT}$. Let $\mathsf{pickt}(s, h_s) = ctid$ and $s.\mathsf{pc}(ctid) = i$. As $\mathsf{GH}_1(s)$, we have $H(i)$ (since the scheduler is secure), and thus $High^{\mathsf{lmem}}(s(ctid))$. Item i) follows from the concurrent SC unwinding lemma (Lemma 6.2). Item ii) follows from the fact that $s \overset{\mathsf{pc}}{\sim} s'$, and item iii) follows from the observation that if $i \in \mathsf{Dom}(\mathsf{next})$, i.e., $s.\mathsf{hidT} = \{ctid\}$, then $s'.\mathsf{pc}(ctid) = i' \in \mathsf{Dom}(\mathsf{next})$, and hence by **NeP1**, $\mathsf{next}(i) = \mathsf{next}(i')$, hence $\mathsf{next}(s) = \mathsf{next}(s')$; otherwise, if $i \notin \mathsf{Dom}(\mathsf{next})$, then $\mathsf{next}(s) = \mathsf{next}(s')$ holds trivially.

If $s, h \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} s_0, h_0$ and $s_0, h_0 \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} s', h_{s'}$, then we can apply the induction hypothesis to both reduction sequences, using the conclusions of the first application of the induction hypothesis to apply the second one, to conclude that $s_0 \sim t$ and $h \overset{\mathsf{hist}}{\sim} h_0$ and $s_0 \overset{\mathsf{pc}}{\sim} t$ and $\mathsf{next}(s) = \mathsf{next}(s_0)$ and $s' \sim t$ and $h_0 \overset{\mathsf{hist}}{\sim} h_{s'}$ and $s' \overset{\mathsf{pc}}{\sim} t$ and $\mathsf{next}(s_0) = \mathsf{next}(s')$. We are done by transitivity of history equivalence and pc equivalence and equality.

□

Next, we prove a locally respects lemma for $\rightsquigarrow^{\mathsf{vis}}_{\text{conc}}$ by using Lemma A.2.

Lemma A.3. *If $s, h_s \rightsquigarrow^{\mathsf{vis}}_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow^{\mathsf{vis}}_{\text{conc}} t', h_{t'}$ and $s \sim t$ and $s \overset{\mathsf{pc}}{\sim} t$ and $h_s \overset{\mathsf{hist}}{\sim} h_t$ then $s' \sim t'$ and $s' \overset{\mathsf{pc}}{\sim} t'$ and $h'_s \overset{\mathsf{hist}}{\sim} h'_t$.*

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

if $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$. Note that $s.\mathsf{lowT} = t.\mathsf{lowT}$ follows from $s \overset{\mathsf{pc}}{\sim} t$. Hence, by definition of secure scheduler, there are two cases to treat: either $ctid_s = ctid_t$ and $k_s = k_t$, or else $k_s \not\leq k$ and $k_t \not\leq k$.

In the first case, observe that necessarily $s'.\mathsf{lowT} = s.\mathsf{lowT}$ and $t'.\mathsf{lowT} = t.\mathsf{lowT}$, and thus $s'.\mathsf{lowT} = t'.\mathsf{lowT}$. Furthermore, $i_s = i_t$. Item i) follows by Lemma 6.1; item ii) follows from Hypothesis 4; item iii) follows from the fact that $h_{s'} = \langle ctid_s, k_s \rangle :: h_s$ and $h_{t'} = \langle ctid_t, k_t \rangle :: h_t$.

In the second case, the result is a direct consequence of the definitions.

if $s, h_s \rightsquigarrow_{\text{conc}} s_1, h_{s_1} \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} s_2, h_{s_2} \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\text{conc}} t_1, h_{t_1} \rightsquigarrow^{\mathsf{hid}}_{\text{conc}} t_2, h_{t_2} \rightsquigarrow_{\text{conc}} t', h_{t'}$. In this case, we must have $ctid_s = ctid_t$ (so we drop subscripts) and $k_s = k_t$, and furthermore $s_1.\mathsf{lowT} = t_1.\mathsf{lowT}$ and $i_s = i_t$. We apply Lemma 6.1 to s and t to conclude that $s_1 \sim t_1$ and $h_{s_1} \overset{\mathsf{hist}}{\sim} h_{t_1}$. Furthermore, $s_1 \overset{\mathsf{pc}}{\sim} t_1$, and by **NeP3**, $\mathsf{next}(s_1) = \mathsf{next}(t_1)$.

By applying Lemma A.2 to s_1, s_2 and t_1, we conclude that $s_2 \sim t_1$, and $h_{s_1} \overset{\mathsf{hist}}{\sim} h_{s_2}$, and $s_2 \overset{\mathsf{pc}}{\sim} t_1$, and $\mathsf{next}(s_1) = \mathsf{next}(s_2)$. Using these facts and by applying Lemma A.2 on t_1, t_2 and s_2, we conclude that $t_2 \sim s_2$, and $h_{t_1} \overset{\mathsf{hist}}{\sim} h_{t_2}$, and $t_2 \overset{\mathsf{pc}}{\sim} s_2$, and $\mathsf{next}(t_1) = \mathsf{next}(t_2)$.

To prove that $s' \sim t'$, we apply Hypothesis 3 to conclude that $s_2(ctid) \sim_l s'(ctid)$ and $s_2 \overset{\mathsf{gmem}}{\sim} s'$. Likewise, $t_2(ctid) \sim_l t'(ctid)$ and $t_2 \overset{\mathsf{gmem}}{\sim} t'$. By Hypothesis 5, we also have $s_2(ctid) \sim_l t_2(ctid)$, and hence $s'(ctid) \sim_l t'(ctid)$,

from which it is easy to conclude.
To conclude that $s' \overset{\mathsf{pc}}{\sim} t'$, we use the fact that $\mathsf{next}(s_2) = \mathsf{next}(t_2)$ and apply **NeP2**.
To conclude that $h_{s'} \overset{\mathsf{hist}}{\sim} h_{t'}$, we use the fact that $h_{s_1} \overset{\mathsf{hist}}{\sim} h_{t_1}$ and that $h_{s_1} \overset{\mathsf{hist}}{\sim} h_{s'}$ and $h_{t_1} \overset{\mathsf{hist}}{\sim} h_{t'}$.

□

We can now conclude the proof of soundness (Theorem 6.1) by repeatedly applying Lemma A.3, and by Lemma A.1.

Soundness of the instantiation

Definition A.1 (Source labels and control flows)**.** *Natural numbers are added as labels to the source syntax to identify program points where control flow can branch. Therefore, commands are described by the following grammar:*

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

Definition A.2 (Branching commands)**.** *The branching commands are those of the form* $\texttt{if } e \texttt{ then } c \texttt{ else } c$ *and* $\texttt{while } e \texttt{ do } c$ *. The set* $\mathcal{LL}^{\#}$ *consists on all the labels of branching commands in the program.*

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

Definition A.3 (Contexts)**.** *A context* C *for commands is defined as an element of the following grammar:*

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

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

The definition of contexts for unlabeled commands is very similar to Definition A.3. Therefore, we abuse of notation and denote C as contexts for labeled or unlabeled commands. We define the size of context as follows.

Our compilation function $\mathcal{S}$ takes two arguments: the code to compiled and the compiled code belonging to different threads. For technical reasons, it is necessary to identify what is the value of the second argument when the compilation of commands are performed. We then introduce the following judgment.

Definition A.4. *Given commands* c *and* c'*, and sequences of compiled instructions* T*, and* T'*, the judgment* $\mathcal{S}(c, T) ::\vdash \mathcal{S}(c', T')$ *is defined by the following rules.*

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

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

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

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

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

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

Intuitively, $\mathcal{S}(c, T) ::\vdash \mathcal{S}(c', T')$ denotes the fact that when compiling the command c, the function $\mathcal{S}$ receives T' as a second argument when compiling c'. For simplicity, we write $\mathcal{S}(c, []) ::\vdash \mathcal{S}(c', T')$ as $\mathcal{C}(c) ::\vdash \mathcal{S}(c', T')$.

We assume that two instructions are the same iff they are located in the same position of in the compiled code. The following function is useful to define regions at the target code.

Definition A.5 (Function $\odot$)**.** *Given a program P and its compilation $l_p = \mathcal{C}(P)$, the function $\odot :: l_p \to [1..\#l_p]$ is defined as $\odot(i) =$ the position of the instruction i in the sequence l_p.*

We then define regions in the target code.

Definition A.6 (Compiler regions)**.** *Given a branching command $[c]^n$ in a source program P. Then, we define* $\mathsf{tregion}(n)$ *as follow:*

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

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

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

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

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

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

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

Definition A.7 (*se* determined by E and F)**.** *Given a program P and an instruction $i \in \mathcal{C}(P)$, we define $se(\odot(i))$ as follows:*

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

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

Lemma A.4. *Given se obtained as described in Definition A.7, program P, a context C, a branching command $[c]^n$ such that $P = C[[c]^n]$, $\vdash_L P$, $\vdash_L [c]^n_H$ is in the type derivation of P, $l_p = \mathcal{C}(P)$, $i \in \mathsf{tregion}(n)$; then $\exists j \in l_p . i \mapsto^* \odot(j) \wedge se(\odot(j)) = L$*

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

Lemma A.5 (From inside of top-level-branches)**.** *Given commands d and c, and label n such that $\vdash_L d$, $\vdash_H [c]^n_H$ is in the type derivation of $\vdash_L d$ and $n \in \mathsf{labels}(d)$, then there exists a command c' and a label k such that $k \in \mathsf{labels}(d)$, $\vdash_L [c']^k_H$ is in the type derivation of $\vdash_L d$, and $n \in \mathsf{labels}(c')$.*

Theorem A.1 (NePd)**.** $\mathsf{Dom}(\mathsf{next}) = \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\ i \mapsto^\star j \wedge \neg H(j)\}$

Proof. In order to prove this equality, we need to prove inclusion of sets. Firstly, $\mathsf{Dom}(\mathsf{next}) \subseteq \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\ i \mapsto^\star j \wedge \neg H(j)\})$ is proved by inspecting Definition 7.4, and Lemma A.4. Lastly, $\{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\ i \mapsto^\star j \wedge \neg H(j)\} \subseteq \mathsf{Dom}(\mathsf{next}))$ is proved by contradiction. We assume that $k \in \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\ i \mapsto^\star j \wedge \neg H(j)\} \wedge k \notin \mathsf{Dom}(\mathsf{next})$. Then, we do case analysis on the command which compilation generated the instruction i such that $\odot(k) = i$, and based on Lemma A.5, contradictions are obtained. □

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

Lemma A.6 (**next** is not defined for join points)**.** *Given a branching point $[c]^n$ in the typable source program $\vdash_L P$ such that $\vdash_L [c]^n_H$, then $\mathsf{next}(\mathsf{jun}(n))$ is undefined.*

Proof. By contradiction. □

Theorem A.2 (NeP1)**.** $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = \mathsf{next}(j)$

Proof. Since $i \in \mathsf{Dom}(\mathsf{next})$, there exists a command c_i and a number n_i such that $\vdash_L [c_i]^{n_i}_H$ and $\forall k \in \mathsf{tregion}(n_i).\mathsf{next}(k) = \mathsf{jun}(n_i)$. On the other hand, since $j \in \mathsf{Dom}(\mathsf{next})$, there exists another command c_j and a number n_j such that $\vdash_L [c_j]^{n_j}_H$

and $\forall k \in \mathsf{tregion}(n_j).\mathsf{next}(k) = \mathsf{jun}(n_j)$. By instantiating SOAP 1 with i and j, we have that $i \mapsto j \wedge (i = n_i \vee i \in \mathsf{tregion}(n_i)) \Rightarrow j \in \mathsf{tregion}(n_i) \vee j = \mathsf{jun}(n_i)$, which we split into:

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

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

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

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

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

$j = \mathsf{jun}(n_i)$). By Lemma A.6, $\mathsf{next}(j)$ is undefined. However, $j \in \mathsf{Dom}(\mathsf{next})$ by Hypothesis, which implies that $\mathsf{next}(j)$ is defined. Contradiction.

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

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

$j = \mathsf{jun}(n_i)$). By Lemma A.6, $\mathsf{next}(j)$ is undefined. However, $j \in \mathsf{Dom}(\mathsf{next})$ by Hypothesis, which implies that $\mathsf{next}(j)$ is defined. Contradiction.

□

Theorem A.3 (NeP2)**.** $i \in \mathsf{Dom}(\mathsf{next}) \wedge j \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = j$

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

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

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

We proceed by doing case analysis on i.

$i = n_i$). Observe that this can happen when c_i is a `while`-loop. By (3), we have that $j \in \mathsf{tregion}(n) \vee j = \mathsf{jun}(n)$.

$j \in \mathsf{tregion}(n)$). It cannot happen. Observe that we assume that $j \notin \mathsf{tregion}(n)$ by Hypothesis.

$j = \mathsf{jun}(n)$). Since $i \in \mathsf{tregion}(n)$, we know that $\mathsf{next}(i) = \mathsf{jun}(n)$ and $j = \mathsf{jun}(n)$, which implies that $\mathsf{next}(i) = j$ as expected.

$i \in \mathsf{tregion}(n_i)$). It proceeds similarly as when $i = n_i$ but applying (4) instead.

□

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

Lemma A.7. *Given a command c_p with two branching instructions $[c_1]^{n_1}$ $[c_2]^{n_2}$, where $n_1 \neq n_2$, and two contexts C_1 and C_2 with only a hole each one. If $c_p = C_1[[c_1]^{n_1}]$ and $c_p = C_2[[c_2]^{n_2}]$, then*

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

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

Lemma A.8. *Given program P and command $[c]^i$ such that $\vdash_\alpha [c]^i_\alpha$ is in the type derivation of $\vdash_L P$, $j, k \in \mathsf{Dom}(\mathsf{next}), i \notin \mathsf{Dom}(\mathsf{next}), i \mapsto j, i \mapsto k$, and $j \neq k$, where $\alpha \in \{L, H\}$, then $\mathsf{next}(k) = \mathsf{next}(j)$.*

Proof. It consists on proving that the hypothesis does not hold. To do that, we consider, based on Lemma A.7, how target regions associated to k and i are included. □

Lemma A.9. *Given program P and command $[c]^i$ such that $\vdash_L [c]^i_H$ is in the type derivation of $\vdash_L P$, $j, k \in \mathsf{Dom}(\mathsf{next}), i \notin \mathsf{Dom}(\mathsf{next}), i \mapsto j, i \mapsto k$, and $j \neq k$, then $\mathsf{next}(k) = \mathsf{next}(j)$.*

Proof. By case analysis on $[c]^i$. □

Theorem A.4 (NeP3)**.** $j, k \in \mathsf{Dom}(\mathsf{next}) \wedge i \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = \mathsf{next}(k)$

Proof. We have that i is a branching command. Consequently, $[c]^i$ can be typed as $\vdash_L [c]^i_L$, $\vdash_H [c]^i_H$, or $\vdash_L [c]^i_H$ in the type derivation of the program. By case analysis on the typing of $[c]^i$, the result follows based on Lemmas A.8 and A.9. □

Theorem A.5 (NeP4)**.** $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge k \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = k$

Proof. By applying Theorem A.2 with i and j, we obtain that $\mathsf{next}(i) = \mathsf{next}(j)$. By applying Theorem A.3 with i and k, we obtain that $\mathsf{next}(i) = k$. Therefore, we have that $\mathsf{next}(j) = \mathsf{next}(i) = k$. □

Theorem 6.1 *Definition 7.4 satisfies properties **NePd** and **NeP1–4**.*

Proof. The proof easily follows from Theorems A.1, A.2, A.3, A.4, and A.5. □

Typability Preservation

The following lemma claims that a source expression typable compiles to typable target code. Besides the conclusion of typability for the target code, the lemma also states that the final security operand stacks are of the form $k : st$, with k being the type of the source expression, and st being the initial operand stack used in the transfer rules for the compiled code. Since the new concurrent features of source and target languages in this paper do not include new expressions, this lemma is equivalent to previous work for sequential languages, and a proof can be found in [Rezk 2006].

Lemma A.10. *Let e be an expression in $[c]^n$ such that $[c]^n$ is the inner-most command that encloses e and $\Gamma \vdash c : E, F$ and $\Gamma \vdash e : k$, and $\mathcal{S}(c)[i..j] = \mathcal{E}(e)$. Let se be the security environment determined by E, F. Then for any $st_i \in \mathcal{ST}$ there exist $st_{i+1}, ..st_j$ such that the following hold:*

(1) *for every* $l \mapsto l'$ *in* $i..j$ *then* $l, se \vdash st_l \Rightarrow st_{l'}$;
(2) $j, se \vdash st_j \Rightarrow (k \sqcup se(i)) :: st_i$.

Theorem 6.2 *For a given source-level program c, assume* $nf(c)$ *is obtained from c by replacing all occurrences of* `fork`(d) *by d. If command* $nf(c)$ *is typable under the Volpano-Smith-Irvine type system [Volpano et al. 1996] then* $se, \mathcal{S} \vdash \mathcal{C}(c)$ *for some se and* $\mathcal{S}$.

Proof. First we define how to obtain intermediate typing from the high level typing as follows: Let D be a typing derivation for a source program SP in the high level type system. Define security environments E and F as follows:

> If n belongs to some region of a branching label n' of a command c' in SP such that the intro judgement for c' types it with write effect H, then $E(n)$ and $F(n)$ are defined as the write level of the intro judgement for $[c]^n$ in D. That is, if $D ::\vdash c : H$ then $F(n) = E(n) = H$.
>
> Otherwise, $E(n) = L$. If n is a branching label not contained in any region such that its intro judgement for types it with write effect H, then $F(n) = H$.

By induction in the structure of the source commands.

Case: $c \equiv [x := e]^n$. We have to prove that if $[x := e]^n$ is typable:

$$\frac{\Gamma \vdash e : k \qquad k \leq \Gamma(x)}{\Gamma \vdash x := e \ : \Gamma(x)}$$

then the constraint $k \sqcup E(n) \leq \Gamma(x)$ from its corresponding intermediate typing rule holds. By definition of E above, if $\Gamma(x) = H$ and n is inside a high region then $E(n) = H$. Otherwise $E(n) = L$. So the constraint $k \sqcup E(n) \leq \Gamma(x)$ is satisfied because of the contraint of the $(Assign)$ high level typing rule for, $k \leq \Gamma(x)$. By Definition A.7, for all program points j included in the compilation of $[x := e]^n$, then $se(j) = E(n)$. By Lemma A.10, we have that typability of compilation of expression e leads to an security operand stack of the form $(k \sqcup se(i)) :: st_i$. Typability of the store instruction in n follows by constraint $k \sqcup E(n) \leq \Gamma(x)$ and the fact that $E(n) = se(j)$ for all j in n.

Case: $c \equiv [\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$. In the high level type system:

$$\frac{\textsc{Cond} \quad \vdash e : k \qquad \vdash c_1 : k \qquad \vdash c_2 : k}{\vdash [\mathsf{if}\ e\ \mathsf{then}\ \ c\ \mathsf{else}\ c']^n \ \ : \ \ k}$$

We need to show that the if command is typable by the intermediate type system with the definition of E, F given above. We need to show that if $\vdash c_i : H$ then $\vdash_\alpha c_i : E, F$. This follows by inductive hypothesis. Furthermore if n does not belong to any high region, then we need to show the hypothesis of the $TOP - H - COND$ rule on E, that is $F(n) = L$. This follows by definition of F above. To prove that compilation of command n is typable, recall that by definition of source regions, c_1 and c_2 are included in the region of n and then by definition of E above $F(n') = E(n') = H$ for all program points n' inside a high region. By Definition A.7, $se(j) = H$ for all instructions j inside the high region of n. Thus the constraint of the target typing rule $se = \mathsf{lift}_k(se)$ holds. By Lemma A.10, we have that

typability of compilation of expression e leads to an security operand stack of the form $(k \sqcup se(i)) :: \epsilon$. By semantics of the if instruction, the operand stack is empty. Thus the lift of the security operand stack holds and we conclude.

Case: $c \equiv [\mathsf{while}\ e\ \mathsf{do}\ c1]^n$. The proof is analog to the if case.

Case: $c \equiv c'; c''$. By inductive hypothesis.

Case: $c \equiv [\mathsf{fork}\ d]^n$. By hypothesis, $\mathsf{fork}\ d$ is tranformed into command d. Typability in the intermediate type system follows by inductive hypothesis. Recall that compilation of $\mathsf{fork}\ d$ gives a start instruction for the current thread and compilation of d for another thread. Typability of d follows by the fact that d is typable applying inductive hypothesis. To prove typability of compilation of start, we need to verify the following typing rule:

$$\frac{P[i] = \mathsf{start}\ pc \quad se(i) \leq se(pc)}{se, i \vdash st \Rightarrow st}$$

By Definition A.7, $se(\odot(\mathsf{start})) = E(n)$, where n corresponds to a skip command. Since i belongs to the compilation of n then $E(n) = se(i)$. We have that $se(i) \leq se(pc)$ and we conclude. □