Available online at www.sciencedirect.com

ScienceDirect

Computer Languages, Systems & Structures 33 (2007) 35–59

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES

www.elsevier.com/locate/cl

Security types preserving compilation[☆]

Gilles Barthe[a], Tamara Rezk[a,*], Amitabh Basu[b,1]

[a]*INRIA Sophia-Antipolis, France*
[b]*Stony Brook University, USA*

Received 21 January 2005; accepted 16 May 2005

Abstract

Starting from the seminal work of Volpano and Smith, there has been growing evidence that type systems may be used to enforce confidentiality of programs through non-interference. However, most type systems operate on high-level languages and calculi, and "low-level languages have not received much attention in studies of secure information flow" (Sabelfeld and Myers, [Language-based information-flow security. IEEE Journal on Selected Areas in Communications 2003; 21:5–19]). Therefore, we introduce an information flow type system for a low-level language featuring jumps and calls, and show that the type system enforces termination-insensitive non-interference.

Furthermore, information flow type systems for low-level languages should appropriately relate to their counterparts for high-level languages. Therefore, we introduce a compiler from a high-level imperative programming language to our low-level language, and show that the compiler preserves information flow types.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Security; Non-interference; Program analysis; Low-level languages

[☆] Partially supported by the ACI Sécurité SPOPS, by the RNTL project CASTLES and by the IST project INSPIRED.

[*] Corresponding author. Tel.: +33 492 387 816; fax: +33 492 385 060.

E-mail addresses: gilles.barthe@sophia.inria.fr (G. Barthe), tamara.rezk@sophia.inria.fr (T. Rezk), abasu@cs.sunysb.edu (A. Basu).

[1] This work was performed during an INRIA International Internship during Summer 2003, while the author was studying at IIT Delhi, India.

1477-8424/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2005.05.002

1. Introduction

Type systems are popular artefacts to enforce safety properties in programming languages. They are also increasingly being used in the context of mobile code to enforce security policies. For example, it is natural to assign to program variables a security level that stipulates its confidentiality status (such as secret or public), and to guarantee that programs do not leak secret information through execution. The absence of information leakage can be made precise with non-interference, as defined in the work of Goguen and Meseguer [2], and can be enforced via information flow type systems [3]. Such information flow type systems have been thoroughly studied in the literature, see e.g. [1] for a survey. However, most works focus on high-level calculi, including λ-calculus, see e.g. [4], π-calculus, see e.g. [5], and ς-calculus [6], or high-level programming languages, including Java [7,8] and ML [9]. In contrast, relatively little is known about non-interference for low-level languages, in particular because their lack of structure renders control flow more intricate; in fact many existing works, among which [10,11], use model-checking and abstract interpretation techniques to detect illegal information flows, but do not provide proofs of non-interference for programs that are accepted by their analysis.

The first part of this paper is devoted to the definition of an information flow type system for a low-level language with jumps and procedure calls, and to a proof that the type system enforces termination-insensitive non-interference. Informally, the security policy is expressed as a mapping $\Gamma : \mathscr{X} \to \mathscr{S}$ that assigns to each register a security level taken from $\{H, L\}$. As usual H denotes confidential data and L denotes public data, so registers $x \in \mathscr{X}$ such that $\Gamma(x) = L$ correspond to the registers that are observable by the attacker. Then non-interference is expressed as

$$\rho \sim \rho' \text{ and } P, \rho \Downarrow \mu \text{ and } P, \rho' \Downarrow \mu' \text{ imply } \mu \sim \mu'$$

where $P, \rho \Downarrow \mu$ denotes that executing program P with initial memory ρ yields the final memory μ, and $\rho \sim \rho'$ denotes that ρ and ρ' coincide on all variables $x \in \mathscr{X}$ such that $\Gamma(x) = L$.

In order to enforce non-interference of programs, we follow the principles of Java bytecode verification, see e.g. [12], namely to provide an abstract transition relation between typed states, and to compute types through a dataflow analysis based on the abstract transition relation. The proof of soundness of the type system relies on a general method. Informally, we define a notion $\sim$ of L-equivalence between states; the idea is that two states are L-equivalent if they cannot be distinguished from one another by an attacker. Then, writing $s \leadsto u$ to denote that performing one-step execution from state s results in a new state u, we show that:

- if $s \leadsto s'$, and $u \leadsto u'$, and $s \sim s'$, and the program counters of s and u coincide, then $u \sim u'$; furthermore, the program counters of s' and u' coincide, or the program counters of s' and u' belong to the control dependence region of a branching instruction that performs a test on confidential data;
- if the program counter of s belongs to the control dependence region of a branching instruction that performs a test on confidential data, and $s \leadsto u$, then $s \sim u$, and either the program counter of u remains in the control dependence region to which s belongs, or the program counter of u is the "exit" of this region.

By appealing to properties of control dependence regions, and by combining the two results in an adequate way, and appealing to transitivity of L-equivalence, one shows that typable programs are non-interfering.

The second part of this paper is devoted to proving that information flow types can be preserved by compilation. As suggested by Abadi [13], information flow type systems for low-level languages should appropriately relate to their counterparts for high-level languages, and one would expect that compilation preserves information flow typing. Indeed, we show for a high-level language and an information flow type system that closely resemble those of [3] that compilation function preserves typing. The proof that compilation preserves typing proceeds by induction on the structure of derivations, and can be viewed as a procedure to compute, from a certificate of well-typing at the source program, another certificate of well-typing for the compiled program. It is thus very close in spirit to a certifying compiler [14].

Contents. The remaining paper is organized as follows. Section 2 motivates the issue with non-interference in an assembly language. In Section 3 we define an assembly language that shall serve as the compiler target, endow it with an information flow type system, and prove that the type system is decidable and enforces termination-insensitive non-interference. In Section 5, we introduce a high-level imperative language with procedures and its associated type system. Furthermore, we introduce a compiler that we show to preserve information flow typing; we also show how type-preserving compilation can be used to lift non-interference to the high-level language. We conclude in Section 6, with related work and directions for further research.

2. Motivating examples

Any sound information flow type system for an assembly language must prevent information leakages through direct flows, as illustrated in Example 1, and through indirect flows, as illustrated in Examples 2–5.

Example 1 (*Direct flows*). Consider the following program, where x_L is a low variable and y_H is a high variable:

```
load $y_H$
store $x_L$
return
```

The first instruction pushes the value held in y_H on top of the operand stack, while the second instruction stores the top of the operand stack in x_L. Thus this program fragment stores in the variable x_L the value held in the variable y_H, and thus leaks information.

We avoid such information leakages by assigning a security level to each value in the operand stack, via a so-called *stack type*, and by rejecting programs that attempt storing a value in a low variable when the top of the stack type is high. By forcing the top of the stack type to high after executing the load instruction, the program is thus rejected.

Example 2 (*Indirect flows via assignments*). As is well-known for high-level languages, assignments within branching instructions may lead to information leakages. The following example demonstrates how information may be leaked through assignments within the scope of a branching instruction. Consider the following program, where x_L is a low variable and y_H is a high variable:[2]

```
1  load y_H
2  if 6
3  prim 0
4  store x_L
5  goto 8
6  prim 1
7  store x_L
8  return
```

The program yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Indeed, the final value of x_L is 0 if the initial value of y_H is 0, and 1 otherwise. The problem is caused by an assignment to x_L in the scope of an if instruction.

We avoid such information leakages by defining the scope of a branching instruction, and by requiring that no assignment to a low variable is performed within its scope if the control flow is influenced by a high variable. Technically, this is achieved by tracking for each program point the security level under which they execute, via a so-called *security environment*.

Example 3 (*Indirect flows via abrupt termination*). Abrupt termination in the scope of branching instructions may also cause information leakage. Consider the following program, where x_L is a low variable and y_H is a high variable:

```
1  prim 0
2  store x_L
3  load y_H
4  if 6
5  return
6  prim 1
7  store x_L
8  return
```

The program yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Indeed, the final value of x_L is 1 if the initial value of y_H is 0, and 0 otherwise. The problem is caused by a return instruction in the scope of an if instruction.

We avoid such information leakages by constraining the use of return instructions in the scope of branching statements.

Furthermore, branching instructions may cause information leakage, even if there are no assignments to low variables or return instructions within their scope.

[2] The if n bytecode branches to n if the top of the operand stack is 0, and to the next instruction after if n otherwise.

Example 4 (*Indirect flows via operand stack*). Consider the following program, where x_L is a low variable and y_H is a high variable:

```
1  prim 3
2  prim 4
3  load $y_H$
4  if 6
5  store $y_H$
6  store $x_L$
7  return
```

The program yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Indeed, the final value of x_L is 3 if the initial value of y_H is 0, and 4 otherwise. The problem is caused by an instruction that manipulates the operand stack in the scope of an if instruction (we use a store y_H instruction but a pop instruction would have a similar effect).

We avoid such information leakages by lifting all elements of the stack type to high upon entering a branching instruction whose control flow is influenced by a high variable. Then the assignment to x_L is not allowed, since the values 3 and 4 are tagged as H by the stack type.

Example 5 (*Indirect flows via operand stack*). Consider the following program, where x_L is a low variable and y_H is a high variable:

```
1  prim 3
2  load $y_H$
3  if 6
4  prim 1
5  prim +
6  store $x_L$
7  return
```

The program yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Indeed, the final value of x_L is 3 if the initial value of y_H is 0, and 4 otherwise. The problem is caused by an arithmetic instruction that manipulates the operand stack in the scope of an if instruction.

We avoid such information leakages as in the previous example, i.e. through lifting the operand stack.

3. Assembly language

In this section we introduce a simple assembly language with jumps and procedures that is used in Section 5 as a target of our compiler.

3.1. Syntax of programs

Program are defined as a set of procedures, each of which consists of an array of instructions. Instructions are either branching instructions (both conditional and unconditional), arithmetic instructions, instructions to manipulate registers, or instructions to call and return from a procedure.

$$\begin{array}{rlll} instr & ::= & \mathsf{prim}\ op & \text{primitive value/operation} \\ & \mid & \mathsf{load}\ x & \text{load value of } x \text{ on stack} \\ & \mid & \mathsf{store}\ x & \text{store top of stack in } x \\ & \mid & \mathsf{if}\ j & \text{conditional jump} \\ & \mid & \mathsf{goto}\ j & \text{unconditional jump} \\ & \mid & \mathsf{call}\ f & \text{procedure call} \\ & \mid & \mathsf{return} & \text{return} \end{array}$$

Fig. 1. Instruction set.

Formally, we assume given a finite set $\mathcal{X}$ of registers, and a finite set $\mathcal{F}$ of procedure names, with a distinguished procedure $\mathsf{main} \in \mathcal{F}$. In addition, we define the set $\mathcal{V}$ of values to be $\mathbb{Z}$, and assume given a set $\mathbb{AO} \subseteq \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ of arithmetic operations, and a set $\mathbb{BO} \subseteq \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$ of comparison operators. Then, we define the set Instr of instructions in Fig. 1, where $op \in \mathcal{V} \cup \mathbb{AO} \cup \mathbb{BO}$; f ranges over $\mathcal{F}$, x ranges over $\mathcal{X}$, and j ranges over $\mathbb{N}$. Finally, we define a *program* P as an $\mathcal{F}$-indexed set of *procedures*, with each procedure defined as an array of instructions. We let P_f be the procedure associated to f, and $P_f[i]$ be the ith instruction in P_f.

3.2. Semantics

Program semantics are defined in terms of a small-step operational semantics which describes one step execution of programs. Formally, the operational semantics of programs is given as a transition relation between states. A state consists of a call string, which captures the current sequence of procedure calls, an operand stack, and a register map. In order to guarantee decidability of type-checking, we impose an upper bound both on the size of call strings, and on the size of operand stacks.

Definition 6 (*Operational semantics*). Let P be a program.

1. The set $\mathcal{PP}$ of *programs points* is the set of pairs $\langle f, i\rangle$ with $f \in \mathcal{F}$, and $i \in dom(P_f)$.
2. The set $\mathcal{CS}$ of *call strings* is the set of $\mathcal{PP}$-stacks of length less than max.
3. The set $\mathcal{RM}$ of *register maps* is defined as $\mathcal{X} \to \mathcal{V}$.
4. The set $\mathcal{OS}$ of *operand stacks* is defined as the set of $\mathcal{V}$-stacks of length less than MAX.
5. The set state of *states* is defined as $\mathcal{CS} \times \mathcal{RM} \times \mathcal{OS}$.
6. The *operational semantics* of the assembly language is given by the rules of Fig. 2; we write $\rho \oplus \{x \mapsto v\}$ to denote the unique function ρ' s.t. $\rho'(y) = \rho(y)$ if $y \neq x$ and $\rho'(x) = v$. We let $\leadsto^\star$ denote the reflexive-transitive closure of the relation $\leadsto$. Note that all rules implicitly enforce the size requirements on call strings and operand stacks.
7. We write $P, \rho \Downarrow \mu$ if $\langle\langle \mathsf{main}, 1\rangle :: \varepsilon, \rho, \varepsilon\rangle \leadsto^\star \langle \varepsilon, \mu, \varepsilon\rangle$, where ε denotes the empty stack.

Observe that procedure calls do not activate a new frame with its own local variables and operand stacks, as e.g. in the JVM; in fact, procedures are closer to JVM subroutines than they are to JVM method invocations.

$$\frac{P_f[i] = \mathsf{prim}\ n \qquad n \in \mathbb{Z}}{\langle\langle f, i\rangle :: cs, \rho, s\rangle \leadsto \langle\langle f, i+1\rangle :: cs, \rho, n :: s\rangle}$$

$$\frac{P_f[i] = \mathsf{if}\ j \qquad v \neq 0}{\langle\langle f, i\rangle :: cs, \rho, v :: s\rangle \leadsto \langle\langle f, j\rangle :: cs, \rho, s\rangle}$$

$$\frac{P_f[i] = \mathsf{prim}\ op \qquad op \in \mathbb{AO} \cup \mathbb{BO} \qquad op(n_1, n_2) = n}{\langle\langle f, i\rangle :: cs, \rho, n_2 :: n_1 :: s\rangle \leadsto \langle\langle f, i+1\rangle :: cs, \rho, n :: s\rangle}$$

$$\frac{P_f[i] = \mathsf{if}\ j \qquad v = 0}{\langle\langle f, i\rangle :: cs, \rho, v :: s\rangle \leadsto \langle\langle f, i+1\rangle :: cs, \rho, s\rangle}$$

$$\frac{P_f[i] = \mathsf{load}\ x}{\langle\langle f, i\rangle :: cs, \rho, s\rangle \leadsto \langle\langle f, i+1\rangle :: cs, \rho, \rho(x) :: s\rangle}$$

$$\frac{P_f[i] = \mathsf{goto}\ j}{\langle\langle f, i\rangle :: cs, \rho, s\rangle \leadsto \langle\langle f, j\rangle :: cs, \rho, s\rangle}$$

$$\frac{P_f[i] = \mathsf{store}\ x}{\langle\langle f, i\rangle :: cs, \rho, v :: s\rangle \leadsto \langle\langle f, i+1\rangle :: cs, \rho \oplus \{x \mapsto v\}, s\rangle}$$

$$\frac{P_{f'}[i] = \mathsf{call}\ f}{\langle\langle f', i\rangle :: cs, \rho, s\rangle \leadsto \langle\langle f, 1\rangle :: \langle f', i\rangle :: cs, \rho, s\rangle}$$

$$\frac{P_f[i] = \mathsf{return}}{\langle\langle f, i\rangle :: \langle f', i'\rangle :: cs, \rho, s\rangle \leadsto \langle\langle f', i'+1\rangle :: cs, \rho, s\rangle}$$

$$\frac{P_f[i] = \mathsf{return}}{\langle\langle f, i\rangle :: \epsilon, \rho, s\rangle \leadsto \langle \epsilon, \rho, s\rangle}$$

Fig. 2. Operational semantics of assembly language.

Note that the operational semantics yields a successor relation $\mapsto \subseteq \mathscr{CS} \times \mathscr{CS}$, which is defined by the clauses:

- if $P_f[i] = \mathsf{return}$ then $\langle f, i\rangle :: \langle f', j\rangle :: cs' \mapsto \langle f', j+1\rangle :: cs'$ and furthermore $\langle f, i\rangle :: \varepsilon \mapsto \varepsilon$;
- if $P_f[i] = \mathsf{call}\ f'$ then $\langle f, i\rangle :: cs \mapsto \langle f', 1\rangle :: \langle f, i\rangle :: cs$;
- if $P_f[i] = \mathsf{goto}\ j$ then $\langle f, i\rangle :: cs \mapsto \langle f, j\rangle :: cs$;
- if $P_f[i] = \mathsf{if}\ j$ then $\langle f, i\rangle :: cs \mapsto \langle f, k\rangle :: cs$ for $k \in \{i+1, j\}$;
- otherwise, $\langle f, i\rangle :: cs \mapsto \langle f, i+1\rangle :: cs$.

It is easy to see that $cs \mapsto cs'$ for every states $\langle cs, \rho, s\rangle$ and $\langle cs', \rho', s'\rangle$ such that $\langle cs, \rho, s\rangle \leadsto \langle cs', \rho', s'\rangle$.

3.3. Control dependence regions

In order to prevent indirect flows, we must identify for every if instruction program points that execute under its control condition.

3.3.1. Definitions and properties

For the purpose of this paper, control dependence regions are treated axiomatically. That is, we define for every program P it set of conditional program points:

$$\mathcal{PP}_{\mathrm{if}} = \{\langle f, i\rangle \in \mathcal{PP} \mid P_f[i] = \mathsf{if}\ j\}$$

and assume given two functions:

$$\begin{array}{l}\mathsf{reg} : \mathcal{PP}_{\mathrm{if}} \rightarrow \wp\,(\mathcal{PP})\\ \mathsf{jun} : \mathcal{PP}_{\mathrm{if}} \rightharpoonup \mathcal{PP}\end{array}$$

that respectively compute the control dependence region of an if, and the junction point of the two branches of the if. While there exist algorithms to compute such regions in unstructured languages, see e.g. [15], the soundness of the type system does not hinge on the exact computation of such control dependence regions. That is, the soundness of the type system does not rely on control dependence regions to be minimal in the sense that they only contain program points that indeed belong to the branch of the if. On the other hand, the minimality of control dependence regions and junction points (on compiled programs) is required to establish that compilation preserves typing, as in Theorem 16. Concretely, we establish soundness of the type system from the following assumptions:

Region inclusion property RIP. For every $\langle f, i\rangle, \langle f', i'\rangle \in \mathcal{PP}_{\mathrm{if}}$, such that $\langle f', i'\rangle \in \mathsf{reg}(f, i)$ we have $\mathsf{reg}(f', i') \subseteq \mathsf{reg}(f, i)$;

Safe over approximation property SOAP. For every $\langle f, i\rangle \in \mathcal{PP}_{\mathrm{if}}$ and execution path

$$\langle\langle f, i\rangle :: cs, \rho, s\rangle \rightsquigarrow \langle cs_1, \rho_1, s_1\rangle \rightsquigarrow \cdots \rightsquigarrow \langle cs_n, \rho_n, s_n\rangle$$

one of the following holds:

- $cs_n = \langle f_s, i_s\rangle ::\ \ldots\ : \langle f_1, i_1\rangle :: cs$ with $\langle f_k, i_k\rangle \in \mathsf{reg}(f, i)$ for $1 \leqslant k \leqslant s$;
- there exists $1 \leqslant l \leqslant n$ such that $cs_l = \mathsf{jun}(f, i) :: cs$;
- there exists $1 \leqslant l \leqslant n$ such that $cs_l = \langle f, j\rangle :: cs$ and $p_f[j] = \mathsf{return}$.

We shall use the function reg in the type system, and the assumption about execution paths and jun in the proof of non-interference.

Remark. The assumptions can be simplified to a great extent if we omit procedure calls. On the contrary, defining regions becomes more complex if JVM subroutines are adopted instead of procedure calls. See e.g. [16] for a brief discussion on these points.

3.3.2. Implementation and example

We have implemented (in Haskell) a simple algorithm to compute control dependence regions for our assembly language. The algorithm is based on the notion of dominators, classical in the literature (see e.g. [17, p. 379]). Roughly, a program point cs dominates another program point cs' if everypath from $(main, 1)$ to cs' must go through cs. The algorithm to compute the region of a program point cs first calculates the set D_{cs} of all the program points that are dominated by cs. Then, it proceeds as follows: let $Succs$ be the set of all successors of cs, and $cs' \in Succs$,

- If cs' dominates all its own successors that are in D_{cs}, then discard cs' from $Succs$ and iterate.
- If cs' does not dominate all its own successors that are in D_{cs}, then add cs' to the region of cs and add all successors of cs' that are in D_{cs} to $Succs \backslash \{cs'\}$, then iterate.

The program terminates when there are no more elements in $Succs$. This always happens since the set of program points of every program is finite, and the same element is not added twice to the set of $Succs$.

This algorithm for regions satisfies the SOAP property, but does not provide any guarantee w.r.t. the RIP property since it computes regions one by one. Given the set R of all the regions of a program P calculated by the algorithm above, it is possible to apply an algorithm to obtain regions for P s.t. its regions satisfy the RIP property. Such an algorithm should check that any two regions $r_1(p_1)$ and $r_2(p_2)$ in R are related by inclusion, whenever their intersection is not empty. If their intersection is not empty and one is not included in the other, then take the union of $r_1(p_1)$ and $r_2(p_2)$ and make the union the new region for program points p_1 and p_2. However, we do not detail such an algorithm that enforces RIP since the algorithm given above is already sufficient for programs that are compiled with the compiler described in Section 5.2 (i.e. it enforces RIP for such programs).

Consider the following program:

```
   main=        f=
1  prim 0    1  push 2
2  if 5      2  return
3  call f
4  goto 6
5  prim 3
6  return
```

Using the algorithm to compute regions described above, the region for instruction 2 of procedure $main$ is: $(main, 3)$, $(main, 4)$, $(main, 5)$, $(f, 1) :: (main, 3)$, $(f, 2) :: (main, 3)$.

Consider the following program with a cycle:

```
   main=
1  load x
2  if 4
3  goto 1
4  return
```

This program can be thought as a while in a high level language. The region for instruction at 2 is: $(main, 2)$, $(main, 3)$, $(main, 1)$.

Consider the following program:

```
  main=
1 if 6
2 load x
3 if 1
4 push v
5 goto 7
6 push v
7 return
```

Notice that in this program not only there are two branches depending of instruction 1 (as in a standard if in a high level language) but there is also a cycle produced by instruction 3, so instructions 4 and 5 depend on both instructions 1 and 3.

The region for instruction at 1 includes all program point of the form $(main, i)$ where $2 \leqslant i \leqslant 6$. Instruction 3 does not dominate 1, therefore the region for 3 only includes program points $(main, 4)$ and $(main, 5)$. Thus this program satisfies the RIP property.

4. Non-interference for the assembly language

The purpose of this section is to define an information flow type system that enforces non-interference for programs of the assembly language. Although different notions of non-interference are applicable to our setting, we focus on so-called termination insensitive non-interference, which guarantees that two terminating program executions that start from initial states that are equivalent from the point of view of an attacker will terminate with final states that are also equivalent from the point of view of an attacker.

4.1. Defining non-interference

Non-interference is defined relative to a security policy $\Gamma : \mathscr{X} \rightarrow \mathscr{S}$ that assigns to each register a security level from the set $\mathscr{S} = \{H, L\}$. As usual, we assume that $L \leqslant H$. Note that we assume that the security level of a register is fixed throughout execution.

Definition 7 (*Non-interfering program*). 1. Let $k \in \mathscr{S}$. Two values v and v' are *k-equivalent*, written $v \sim_k v'$, iff $k = H$ or $v = v'$.

2. Two register maps ρ and ρ' are *L-equivalent*, written $\rho \sim \rho'$, if $(\rho\ x) \sim_{(\Gamma\ x)} (\rho'\ x)$ for every $x \in \mathscr{X}$.

3. A program P is *non-interfering*, written $\mathsf{NI}_\Gamma(P)$, if for every register maps $\rho, \rho', \mu, \mu' : \mathscr{X} \rightarrow \mathscr{V}$,

$$\rho \sim_\Gamma \rho' \text{ and } P, \rho \Downarrow \mu \text{ and } P, \rho' \Downarrow \mu' \text{ imply } \mu \sim_\Gamma \mu'.$$

There are at least two obvious directions in which the security policy could be generalized: first, by considering a lattice of security levels instead of the two-point set $\mathscr{S}$, and second, by letting security levels of registers vary throughout execution. The first generalization would add technicalities without adding insight, whereas the second generalization is considered in [18].

Note that the programs given in the introduction can easily been shown to be interfering.

4.2. Abstract transition system

The abstract transition system is given by transfer rules of the

$$\frac{cs = \langle f, i\rangle :: cs_0 \quad P_f[i] = \text{instruction}}{\Gamma, cs \vdash st, se \Rightarrow st', se'},$$

where Γ is the security policy, st, st' are stack types, i.e. stacks of security levels, and se, se' are security environments, i.e. maps that assign a security level to each program point. The transfer rules impose some typing constraints on cs and its successors: more precisely, st, se determine typing constraints for cs, and st', se' determine typing constraints for the successors of cs.

Definition 8 (*Typed states*). 1. The set $\mathcal{ST}$ of stack types is defined as the set of $\mathcal{S}$-stacks of length less than max.
2. The set $\mathcal{SE}$ of security environments is defined as $\mathcal{PP} \to \mathcal{S}$.
3. The set tstate of typed states is defined as $\mathcal{ST} \times \mathcal{SE}$.

As mentioned above, the abstract transition system involves statements of the form $\Gamma, cs \vdash st, se \Rightarrow st', se'$, where cs is a call string, and st, se and st', se' are typed states.

Definition 9 (*Typing transfer rules*). 1. The abstract transition system is defined by the typing transfer rules in Fig. 3.
2. The relation $\Gamma \vdash cs, st, se \Rightarrow cs', st', se'$ is defined as $\Gamma, cs \vdash st, se \Rightarrow st', se'$ and $cs \mapsto cs'$. We let $\Gamma \vdash \cdot, \cdot, \cdot \Rightarrow^{\star} \cdot, \cdot, \cdot$ be the transitive closure of $\Gamma \vdash \cdot, \cdot, \cdot \Rightarrow \cdot, \cdot, \cdot$.

As with the operational semantics, all rules implicitly enforce the size requirements on call strings and operand stacks.

We conclude this section by observing that the transfer function rules define a partial function, i.e. $\Gamma, cs \vdash st, se \Rightarrow st_1, se_1$ and $\Gamma, cs \vdash st, se \Rightarrow st_2, se_2$ implies $st_1 = st_2$ and $se_1 = se_2$.

4.3. Typing programs

Following the type systems for polyvariant subroutines in the JVM, see e.g. [19,20,12], our type system assigns sets of typed states to each program point.

Definition 10 (*Typable programs*). 1. A *security type* is a map $S : \mathcal{CS} \to \wp(\mathsf{tstate})$. Given a security type S and a call string cs we let S_{cs} denote $S(cs)$.
2. A *typing judgment* is a triple of the form $\Gamma, S \vdash P$, where Γ is the security policy under which the program P must be typed, and S is a security type.
3. The program P *has type* S (w.r.t. Γ), written $\Gamma, S \vdash P$, if $\Gamma, S \vdash P$ can be derived from the typing rule:

$$\frac{\begin{array}{c}\forall cs, cs' \in \mathcal{CS}.\ \forall st, se \in S_{cs}.\ cs \mapsto cs' \text{ implies} \\ \exists st', se' \in S_{cs'}.\ \Gamma, cs \vdash st, se \Rightarrow st', se'\end{array}}{\Gamma, S \vdash P}$$

4. The program P *is typable* (w.r.t. Γ), written $\Gamma \vdash P$, if $\Gamma, S \vdash P$ for some security type S.

$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{load}\ x}{\Gamma, cs \vdash st, se \Rightarrow (\Gamma(x) \sqcup se(f,i)) :: st, se}$$
$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{store}\ x \quad k \sqcup se(f,i) \leq \Gamma(x)}{\Gamma, cs \vdash k :: st, se \Rightarrow st, se}$$
$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{prim}\ n}{\Gamma, cs \vdash st, se \Rightarrow se(f,i) :: st, se}$$
$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{prim}\ op}{\Gamma, cs \vdash k_1 :: k_2 :: st, se \Rightarrow (k_1 \sqcup k_2 \sqcup se(f,i)) :: st, se}$$

$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{if}\ j}{\Gamma, cs \vdash k :: st, se \Rightarrow \mathrm{lift}_k(st), \mathrm{lift}_k(se, \mathsf{reg}(f,i))}$$
$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] = \mathsf{return} \quad se(f,i) = L \vee f \neq \mathsf{main}}{\Gamma, cs \vdash st, se \Rightarrow st, se}$$
$$\frac{cs = \langle f, i\rangle :: cs' \quad P_f[i] \in \{\mathsf{goto}\ j, \mathsf{call}\ f'\}}{\Gamma, cs \vdash st, se \Rightarrow st, se}$$

where

- $\sqcup$ denotes the lub of two security levels;
- $\mathrm{lift}_k(st)$, where $k \in \mathcal{S}$, denotes the pointwise extension to the stack type st of the function $\lambda l.\ k \sqcup l$;
- $\mathrm{lift}_k(se, R)$, where $k \in \mathcal{S}$, denotes the pointwise extension for all program points in R of the function $\lambda l.\ k \sqcup l$.

Fig. 3. Transfer rules for instructions.

It is possible to compute the type of a program, by using a dataflow analysis that explores abstract execution paths. The algorithm computes S as follows:

- initially, the call string $\langle \mathsf{main}, 1\rangle : \varepsilon$ is mapped to the typed state $\langle \varepsilon, \lambda p.L\rangle$;
- next, the algorithm repeatedly performs the iterative step, i.e. selects one call string cs, and one typed state $st, se \in S_{cs}$, then performs its abstract execution, i.e. computes the typed state st', se' such that $\Gamma, cs \vdash st, se \Rightarrow st', se'$, and adds st', se' to all sets $S_{cs'}$ where $cs \mapsto cs'$. In case the typed state st', se' does not exist, the algorithm returns an error;
- finally, the algorithm terminates when for every call string cs, and typed state $st, se \in S_{cs}$ and typed state st', se' such that $\Gamma, cs \vdash st, se \Rightarrow st', se'$, we have $st', se' \in S_{cs'}$ for every $cs \in \mathcal{CS}$ such that $cs \mapsto cs'$.

The algorithm terminates because the sets tstate and $\mathcal{CS}$ are finite, and because an auxiliary bitmap indicates on which call strings the algorithm must perform the iterative step. Furthermore, the algorithm returns an error, or computes a type for the program P.

Proposition 11. *It is decidable whether a program P is typable.*

Note that there are a number of variants and optimizations for computing the typing of programs, see e.g. [12].

4.4. Examples

Before proving the soundness of the type system, we consider some examples of typable programs, and of non-typable programs. We start with the non-typable programs of Section 2:

- The first example (direct flow) is rejected by our type system. Indeed, the transition of an instruction store x_L is restricted to the case where the type in the top of the stack type is low, whereas the instruction load y_H pushes a H on the stack type.
- The second and third examples (indirect flow) are rejected by our type system. Indeed, the transition of an instruction if l sets to H the security level of all program points in its control dependence region, whenever the type on the top of the stack type is H, which is the case after executing the load y_H instruction. Then, the type system rejects low assignments that are performed at program points which are high w.r.t. the security environments, and return instructions occurring in the main procedure and moreover at program points which are high w.r.t. the security environments.
- The fourth and fifth examples (indirect flow via operand stack) are rejected by our type system. Indeed, the transition of an instruction if l sets to H the security level of all elements in the stack, whenever the type on the top of the stack type is H, which is the case after executing the load y_H instruction. Then, the type system rejects assignments of high values to low registers.

On the positive side, we shall show in Section 5.2 that all high-level programs that are typable w.r.t. some information flow type system at source code level are compiled into programs that are typable w.r.t. our type system.

We conclude this section with an example of a program that is non-interfering, but that is not accepted by our type system. Consider the following program, where x_L is a low variable and y_H is a high variable:

```
1  load y_H
2  if 5
3  prim 1
4  store x_L
5  prim 1
6  store x_L
```

Instruction 4 necessarily belongs to the region of 2. The transfer rule for if lifts to H all instructions in its region, thus the security environment at program point 4 is H. Hence the store to a low variable at 4 is disallowed by the type system. However, this program terminates with $x_L = 1$ for all initial values of y_H, and hence is non-interfering.

4.5. Soundness

Typable programs are non-interfering.

Theorem 12. *If* $\Gamma \vdash P$ *then* $\mathsf{NI}_\Gamma(P)$.

The idea of the proof is as follows: first, we prove in Lemma 14 that L-equivalence is preserved under one step of execution, if the program is typable. Second, we prove in Lemma 15 that one step execution in a high-level environment yields a result state that is L-equivalent to the original one. By combining these results together, we conclude.

4.5.1. Defining L-equivalence between states

The statements of the main lemmas towards Theorem 12 rely on a notion of L-equivalence between states. This notion is defined in terms of L-equivalence between register maps, as defined in Definition 7, and of L-equivalence between operand stacks. In order for the proofs to go through in the case of high-level environments, the definition of L-equivalence between stacks requires a slight generalization of the pointwise order on stacks. The intuition is that we require operand stacks to be L-equivalent pointwise on some common top part, and then to be high in the bottom part on which they may not coincide.

Definition 13 (*Operand stack and state L-equivalence*). 1. Let *st* be a stack type. We write $\mathsf{high}\ st$ if *st* has length n and $st[i] = H$ for every $1 \leqslant i \leqslant n$.

2. Let *s* be an operand stack and *st* be a stack type; we write $\mathsf{high}\ (s, st)$ if *s* and *st* have the same length n and $st[i] = H$ for every $1 \leqslant i \leqslant n$.

3. Let s, s' be operand stacks and $st, st' \in \mathcal{ST}$. Then $s \sim_{st,st'} s'$ is defined inductively as follows:

$$\frac{\mathsf{high}\ (s, st) \quad \mathsf{high}\ (s', st')}{s \sim_{st,st'} s'}, \qquad \frac{s \sim_{st,st'} s' \quad v \sim_k v'}{v :: s \sim_{k::st,k::st'} v' :: s'}$$

4. Let $\sigma = \langle cs, \rho, s\rangle$ and $\sigma' = \langle cs', \rho', s'\rangle$ be states. Then *state L-equivalence* between σ and σ' w.r.t. register type Γ and stack types *st* and st', written $\sigma \sim_{\Gamma,st,st'} \sigma'$, is defined as $s \sim_{st,st'} s' \wedge \rho \sim_\Gamma \rho'$.

4.5.2. Soundness proof

In the sequel, we use $s \cdot \mathsf{cs}$ to denote the call string of a state *s*.

The first lemma establishes that L-equivalence is preserved under one-step execution. Informally, if $s \sim s'$, and $s \rightsquigarrow u$, and $s' \rightsquigarrow u'$, then $u \sim u'$.

Lemma 14 (*One-step non-interference in low-level environments*). *Suppose* $\Gamma, S \vdash P$. *Let* s_1, s_2, s_1', s_2' *be states such that* $s_1 \cdot \mathsf{cs} = s_2\ \mathsf{cs}$ *and* $s_1 \rightsquigarrow s_1'$, *and* $s_2 \rightsquigarrow s_2'$. *Further let* $(st_1, se), (st_2, se) \in S_{s_1 \cdot \mathsf{cs}}$ *be security types s.t.* $s_1 \sim_{\Gamma,st_1,st_2} s_2$.

Then there exist $(st_1', se') \in S_{s_1' \cdot \mathsf{cs}}$ *and* $(st_2', se') \in S_{s_2' \cdot \mathsf{cs}}$ *s.t.* $s_1' \sim_{\Gamma,st_1',st_2'} s_2'$. *Furthermore, one of the following holds*:

- $s_1' \cdot \mathsf{cs} = s_2' \cdot \mathsf{cs}$;
- $s_1 \cdot \mathsf{cs} = \langle f, i\rangle :: cs$ *and* $s_1' \cdot \mathsf{cs} = \langle f', i'\rangle :: cs$ *and* $s_2' \cdot \mathsf{cs} = \langle f'', i''\rangle :: cs$ *and* $(f, i) \in \mathcal{PP}_{\mathrm{if}}$ *with* $(f', i'), (f'', i'') \in \mathsf{reg}(f, i)$, *and* $se(f_1, i_1) = H$ *for all* $(f_1, i_1) \in \mathsf{reg}(f, i)$, *and* $\mathsf{high}\ st_1$ *and* $\mathsf{high}\ st_2$.

Proof. By a case analysis on the instruction that is executed. $\square$

The second lemma establishes that, in high-level environments, the execution relation is included in the L-equivalence relation. Informally, if $s \rightsquigarrow u$, then $s \sim u$.

Lemma 15 (*One-step non-interference in high-level environments*). *Suppose $\Gamma, S \vdash P$. Let s, s' be states such that $s \rightsquigarrow s'$ and assume $s \cdot \mathsf{cs} = \langle f, i \rangle :: cs_0$. Let $(st, se) \in S_{s \cdot \mathsf{cs}}$ be a security type s.t. $\mathsf{high}\ st$. Let $(f_0, i_0) \in \mathscr{P}\mathscr{P}_{\text{if}}$ s.t. $(f, i) \in \mathsf{reg}(f_0, i_0)$ and $se(f_1, i_1) = H$ for all $(f_1, i_1) \in \mathsf{reg}(f_0, i_0)$. Then there exists $(st', se) \in S_{s' \cdot \mathsf{cs}}$ s.t. $\mathsf{high}\ st'$, and $s \sim_{\Gamma, st, st'} s'$, and $\Gamma, s \cdot \mathsf{cs} \vdash (st, se) \Rightarrow (st', se)$. Furthermore, one of the following holds*:

- $s' \cdot \mathsf{cs} = \langle f', i' \rangle :: cs'_0$ *with* $(f', i') \in \mathsf{reg}(f_0, i_0)$;
- $s' \cdot \mathsf{cs} = \mathsf{jun}(f_0, i_0) :: cs_0$.

Proof. By a case analysis on the instruction that is executed. □

Proof of Theorem 12. Consider the two execution paths

$$\begin{array}{l} s_0 \rightsquigarrow s_1 \rightsquigarrow \ \ldots \ \rightsquigarrow s_{n_1} \\ s'_0 \rightsquigarrow s'_1 \rightsquigarrow \ \ldots \ \rightsquigarrow s'_{n_2} \end{array}$$

where $s_0 = \langle \langle \text{main}, 1 \rangle :: \varepsilon, \rho, \varepsilon \rangle$, and $s'_0 = \langle \langle \text{main}, 1 \rangle :: \varepsilon, \rho', \varepsilon \rangle$, and $s_{n_1} \cdot \mathsf{cs} = s_{n_2} \cdot \mathsf{cs} = \varepsilon$.

By invoking Lemma 14 as long as it applies, we conclude for some maximal q that $s_q \cdot \mathsf{cs} = s'_q \cdot \mathsf{cs}$, and that there exists $(st_q, se), (st'_q, se) \in S_{s_q \cdot \mathsf{cs}}$ such that $s'_q \sim_{\Gamma, st_q, st'_q} s'_q$. Now there are two cases to treat: if $s_q \cdot \mathsf{cs} = \varepsilon$ then $n_1 = n_2 = q$ and we are done; otherwise, the last instruction executed is an if high.

By the typing rule of the if instruction, we have $\mathsf{high}\ st_q$ and $\mathsf{high}\ st'_q$. We now invoke Lemma 15 repeatedly to conclude that there exists s_{q_1} and s'_{q_2} and $(st_{q_1}, se_1) \in S_{s_{q_1} \cdot \mathsf{cs}}$ and $(st'_{q_2}, se'_2) \in S_{s'_{q_2} \cdot \mathsf{cs}}$ such that $s_q \sim_{\Gamma, st_q, st_{q_1}} s_{q_1}$ and $s'_q \sim_{\Gamma, st'_q, st'_{q_2}} s'_{q_2}$. By transitivity, we conclude $s_{q_1} \sim_{\Gamma, st_{q_1}, st'_{q_2}} s'_{q_2}$. Further, we can choose q_1 and q_2 to be the minimal indexes such that $s_{q_1} \cdot \mathsf{cs} = \mathsf{jun}(f, i) :: cs'$ and $s_{q_2} \cdot \mathsf{cs} = \mathsf{jun}(f, i) :: cs'$ respectively. As $\Gamma \vdash s_q \cdot \mathsf{cs}, st_q, se \Rightarrow^\star s_{q_1} \cdot \mathsf{cs}, st_{q_1}, se_1$and $\Gamma \vdash s_q \cdot \mathsf{cs}, st'_q, se' \Rightarrow^\star s'_{q_2} \cdot \mathsf{cs}, st'_{q_2}, se'_2$ and only if statements may modify the security environment, and we assume RIP, we can further conclude that $se_1 = se_2$.

Thus we can apply Lemma 14 again, and repeat the process until reaching the final states of the reduction sequences. □

5. Security type preserving compilation

In this section, we define a high-level imperative language, endow it with a security type system, and introduce a compiler from the source language to the assembly language. Then we show that the compiler preserves security types, and derive as a corollary that the security type system for the source language enforces non-interference.

5.1. Source language

The source language is a simple imperative language with procedures. A procedure is a declaration of the form $\mathsf{proc}\ \ f(\vec{x}) = c;\ \ \mathsf{return}$ where f is a procedure name and c is a command. As with the assembly language, we assume that a program is a list of procedures with a distinguished, *main*, procedure without parameters. Formally, the set Expr of *expressions*, Comm of *commands*, and Prog of *programs* are given

$$(\textsc{Sub})_e \quad \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

$$(\textsc{Sub})_c \quad \frac{\Gamma \vdash P : \tau\, cmd \quad \tau' \leq \tau}{\Gamma \vdash P : \tau'\, cmd}$$

$$(\textsc{Var}) \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$(\textsc{Val}) \quad \Gamma \vdash n : \tau$$

$$(\textsc{Op}) \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ op\ e' : \tau}$$

$$(\textsc{Assign}) \quad \frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau\, cmd}$$

$$(\textsc{Seq}) \quad \frac{\Gamma \vdash P : \tau\, cmd \quad \Gamma \vdash Q : \tau\, cmd}{\Gamma \vdash P; Q : \tau\, cmd}$$

$$(\textsc{While}) \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash P : \tau\, cmd}{\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ P : \tau\, cmd}$$

$$(\textsc{Cond}) \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash P : \tau\, cmd \quad \Gamma \vdash Q : \tau\, cmd}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q : \tau\, cmd}$$

$$(\textsc{App}) \quad \frac{\Gamma \vdash P : \tau\, cmd \quad \Gamma \vdash \vec{e} : \tau \quad \Gamma(\vec{x}) = \tau}{\Gamma \vdash f(\vec{e}) : \tau\, cmd}$$

Fig. 4. Typing rules for high-level language.

by the following syntaxes:

$$\begin{array}{lcl} e & ::= & x \mid n \mid e\ op\ e \\ c & ::= & x := e \mid f(\vec{e}) \mid c; c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \\ P & ::= & [\mathsf{proc}\ f(\vec{x}) = c;\ \mathsf{return}]^\star \end{array}$$

The operational, big-step semantics of programs is based on judgments of the form $\langle c, \mu \rangle \Downarrow \mu'$, where $c \in \mathsf{Comm}$ and $\mu, \mu' : \mathscr{X} \to \mathscr{V}$. Rules are standard, see e.g. [21,3], and omitted. We write $P, \mu \Downarrow \mu'$ iff $\langle c_{\mathsf{main}}, \mu \rangle \Downarrow \mu'$, where c_{main} is the body of the main procedure.

The security type system is based on judgments of the form $\Gamma \vdash e : \tau$ and $\Gamma \vdash c : \tau\ cmd$. A program P is typable, written $\Gamma \vdash P$, if $\Gamma \vdash c_{\mathsf{main}} : \tau\ cmd$ for some τ. The typing rules are inspired from [21,3], and are given in Fig. 4; in the last rule, we assume that the procedure f is defined by $\mathsf{proc}\ f(\vec{x}) = P;\ \mathsf{return}$.

Note that the typing rules exclude mutual and self-recursion; however it is possible to overcome this limitation at the price of further technicalities.

5.2. Compilation

The compilation function $\mathscr{C}_p$ is defined in the usual way from a compilation function on expressions $\mathscr{C}_e : \mathsf{Expr} \to \mathsf{Instr}^\star$, and a compilation function on commands $\mathscr{C}_c : \mathsf{Comm} \to \mathsf{Instr}^\star$. Their formal definitions are given in Fig. 5. In order to enhance readability, we use :: both for concatenating an element

$$
\begin{aligned}
\mathcal{C}_e(x) &= \mathsf{load}\ x \\
\mathcal{C}_e(n) &= \mathsf{prim}\ n \\
\mathcal{C}_e(e\ op\ e') &= \mathcal{C}_e(e) :: \mathcal{C}_e(e') :: \mathsf{prim}\ op \\
\mathcal{C}_c(x := e) &= \mathcal{C}_e(e) :: \mathsf{store}\ x \\
\mathcal{C}_c(f\ \vec{e}) &= \mathcal{C}_e(\vec{e}) :: \mathsf{call}\ f \\
\mathcal{C}_c(c_1;\ c_2) &= \mathcal{C}_e(c_1) :: \mathcal{C}_e(c_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_c(\mathsf{while}\ e\ \mathsf{do}\ c) &= \\
&\mathit{let}\ \ l_1 = \mathcal{C}_e(e); l_2 = \mathcal{C}_c(c); x = \#l_2; y = \#l_1\ \mathit{in} \\
&\qquad \mathsf{goto}\ (pc + x + 1) :: l_2 :: l_1 :: \mathsf{if}\ (pc - x - y) \\
\mathcal{C}_c(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) &= \\
&\mathit{let}\ \ l_e = \mathcal{C}_e(e); lc_1 = \mathcal{C}_c(c_1); lc_2 = \mathcal{C}_c(c_2); \\
] &\qquad x = \#lc_2; y = \#lc_1\ \mathit{in} \\
&\qquad l_e :: \mathsf{if}\ (pc + x + 2) :: lc_2 :: \mathsf{goto}\ (pc + y + 1) :: lc_1
\end{aligned}
$$

$$
\mathcal{C}_p([\mathsf{proc}\ f(\vec{x}) := c;\ \mathsf{return}]^\star) = [f := (\mathsf{store}\ \vec{x} :: \mathcal{C}_c(c) :: \mathsf{return})]^\star
$$

Fig. 5. Compilation of expressions and commands.

to a list and concatenating two lists, and $\#l$ to denote the length of the list l. Furthermore we omit details of calculating pc in the clauses for while and if expressions.

5.3. Preservation of security types

In this section, we assume that regions used by the type system of the assembly language are minimal in the sense that the region of the compilation of an if instruction compiled from a command if e then c_1 else c_2, includes exactly those program points belonging to the compilation of c_1 and c_2. Similarly, the region of the compilation of an if instruction compiled from a command while e do c includes exactly those program points belonging to the compilation of c. We need these minimal regions to show that compilation preserves typing.

Theorem 16. *If $\Gamma \vdash P$ then $\Gamma \vdash \mathscr{C}_p(P)$.*

The proof proceeds in two steps. First, we show how to compute from an expression in the source language and its type, the type of the corresponding compiled code produced by the function $\mathscr{C}_e$. By abuse of notation, we write $se = \tau$ if $se(f, j) = \tau$ for every $\langle f, j\rangle \in \mathscr{P}\mathscr{P}$.

Lemma 17. *Assume e is an expression in P and $\Gamma \vdash e : \tau$, and $\mathscr{C}_p(P)_f[i \ldots j] = \mathscr{C}_e(e)$. For every $cs_0 \in \mathscr{C}\mathscr{S}$ and $st, se \in \mathscr{S}\mathscr{T}$ s.t. $se = \tau$, there exists $S^e_{cs_0,st,se} : \{\langle f, k\rangle :: cs_0 \mid i \leqslant k \leqslant j + 1\} \to \mathscr{S}\mathscr{T}$—by abuse of* *notation, we often write S^e—s.t.:*

1. *for every $cs, cs' \in dom(S^e)$, if $cs \mapsto cs'$ then $\Gamma, cs \vdash S^e(cs) \Rightarrow S^e(cs')$;*
2. *$S^e(\langle f, j+1\rangle :: cs_0) = \tau :: st, se$.*

Proof. By structural induction on instructions. □

Second, we extend the result to commands.

Lemma 18. *Assume c is a command in P, and $\Gamma \vdash c : \tau\, cmd$, and $\mathscr{C}_p(P)_f[i \dots j] = \mathscr{C}_c(c)$. For every $cs_0 \in \mathscr{CS}$ and $st_0, se_0 \in \mathscr{ST}$ s.t. $se_0 = \tau$, there exists $S^c_{cs_0,st_0,se_0} : \mathscr{CS} \rightharpoonup \wp(\mathscr{ST})$—by abuse of notation, we often write S^c—s.t.:*

1. *for every $cs, cs' \in dom(S^c)$ and $st, se \in S^c(cs)$ s.t. $cs \mapsto cs'$, there exists $st', se' \in S^c(cs')$ s.t. $\Gamma, cs \vdash st, se \Rightarrow st', se'$;*
2. *there exists st' s.t. $st' = st_0$ or $st' = \mathsf{lift}_H\ st_0$, and for every $cs \in dom(S^c)$, $cs' \notin dom(S^c)$ and $st, se \in S^c(cs)$ s.t. $cs \mapsto cs'$, $\Gamma, cs \vdash st, se \Rightarrow st', se'$ for $se' = \mathsf{lift}_H(se_0, \mathsf{reg}(f, i))$ for some f, i or $se' = se_0$. We write $\phi^c_{cs_0,st_0,se_0}$ for st' and $\pi^c_{cs_0,st_0,se_0}$ for se'.*

Proof. By structural induction on instructions. □

Proof of Theorem 16. Set $se_0 = \tau$ where $\Gamma \vdash c_{\mathsf{main}} \cdot \tau cmd$ By construction, the function $S^{c_{\mathsf{main}}}_{\langle \mathsf{main}, 1\rangle : \varepsilon, \varepsilon, se_0}$ is defined for all cs s.t. $\langle \mathsf{main}, 1\rangle : \varepsilon \mapsto^\star cs$. It is then immediate to conclude. □

5.4. Recovering non-interference for the source language

One can also prove that compilation preserves operational semantics.

Proposition 19 (*Preservation of semantics*). *For every program P and memories ρ, μ, if $P, \rho \Downarrow \mu$ then $\mathscr{C}_p(P), \rho \Downarrow \mu$.*

Proof. Routine and omitted. □

By combining Proposition 19 and Theorem 16 we are able to recover the non-interference result for typable source programs.

Corollary 20 (*Non-interference for source language*). *Let P be a program, let $\Gamma : \mathscr{X} \to \mathscr{S}$ and assume that $\Gamma \vdash P$. Then P is non-interfering w.r.t. Γ in the sense that for every $\rho, \rho', \mu, \mu' : \mathscr{X} \to \mathscr{V}$ such that $\rho \sim_\Gamma \rho'$ and $P, \rho \Downarrow \mu$, and $P, \rho' \Downarrow \mu'$, we have $\mu \sim_\Gamma \mu'$.*

Proof. By Proposition 19, $\mathscr{C}_p(P), \rho \Downarrow \mu$ and $\mathscr{C}_p(P), \rho' \Downarrow \mu'$. Furthermore $\Gamma \vdash \mathscr{C}_p(P)$ by Theorem 16. Hence $\mathscr{C}_p(P)$ is non-interfering w.r.t. Γ by Theorem 12, and thus $\mu \sim_\Gamma \mu'$ by definition of non-interference. □

Example 21. In this example the typing is done at source code for a program P level using the type system in Fig. 4.

The compilation using $\mathscr{C}_p$ in Fig. 5 is shown, as well as computation of its type, using the typing transfer rules of Fig. 3 and the typing rule of Definition 10.

The source code is [$\mathsf{proc\ main} = c$; *return*] where c is

$$\mathsf{if}\ y_H = 0\ \mathsf{then}\ y_H := x_L\ \mathsf{else}\ y_H := 1\,;\ x_L := 3;$$

In order to type this program, we need to define the environment Γ, classifying variables in the program as low (L) or high (H). Let $\Gamma(x_L) = L$ and $\Gamma(y_H) = H$. The type for program P is $L\ cmd$ (i.e. $\Gamma\vdash_S main : L\ cmd$), meaning that low variables are possibly modified in this program. In fact, variable x_L is modified. The derivation tree for c is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(y_H) = H}{\Gamma \vdash y_H : H} \quad \dfrac{}{\Gamma \vdash 0 : H}}{\Gamma \vdash y_H = 0 : H} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma(x_L) = L}}{\Gamma \vdash x_L : L}}{\Gamma \vdash x_L : H} \quad \dfrac{}{\Gamma(y_H) = H}}{\Gamma \vdash y_H := x_L : H\ cmd} \quad \dfrac{\dfrac{}{\Gamma \vdash 1 : H} \quad \dfrac{}{\Gamma(y_H) = H}}{\Gamma \vdash y_H := 1 : H\ cmd}}{\Gamma \vdash \mathsf{if}\ y_H = 0\ \mathsf{then}\ y_H = x_L\ \mathsf{else}\ y_H := 1 : H\ cmd}}{\Gamma \vdash \mathsf{if}\ y_H = 0\ \mathsf{then}\ y_H = x_L\ \mathsf{else}\ y_H :=1 : L\ cmd} \quad \dfrac{\dfrac{}{\Gamma \vdash 3 : L} \quad \dfrac{}{\Gamma(x_L) = L}}{\Gamma \vdash x_L := 3 : L\ cmd}}{\Gamma \vdash \mathsf{if}\ y_H = 0\ \mathsf{then}\ y_H := x_L\ \mathsf{else}\ y_H := 1\,;\ x_L := 3 :\ L\ cmd}$$

The compilation $\mathscr{C}_p(P)$ is [$main := c'$] where c' is the code shown below, together with the security type according to the typing rule in Definition 10.

	c'	State Types for c'
1	load y_H	$\{\varepsilon, se_L\}$
2	prim 0	$\{H \cdot \varepsilon, se_L\}$
3	prim =	$\{L \cdot H, se_L\}$
4	if 8	$\{H \cdot \varepsilon, se_L\}$
5	load x_L	$\{\varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
6	store y_H	$\{H \cdot \varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
7	goto 10	$\{\varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
8	prim 1	$\{\varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
9	store y_H	$\{H \cdot \varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
10	prim 3	$\{\varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
11	store x_L	$\{L \cdot \varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$
12	return	$\{\varepsilon, se_L \oplus \{main, 5 \mapsto H, main, 6 \mapsto H, main, 7 \mapsto H, main, 8 \mapsto H, main, 9 \mapsto H\}\}$

It is left to the reader to check that the type obtained by using the type system for the assembly language is the same type obtained using the implicit algorithm in the proof of Theorem 16.

6. Conclusion

We have shown how type systems can be used to enforce non-interference in a low-level language with procedures, and that one can define a security types preserving compiler from a high-level imperative language to such a low-level language.

6.1. Related work

As emphasized in the introduction, static enforcement of non-interference through type systems is a well-researched topic, see e.g. [1] for a survey. We only comment on some of the most relevant literature.

Low-level languages. Lanet et al., see e.g. [11], develop a method to detect illicit flows for a sequential fragment of the JVM. In a nutshell, they proceed by specifying in the SMV model checker a symbolic transition semantics of the JVM that manipulates security levels, and by verifying that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their analysis is more flexible than ours, in that it accepts programs such as $y_L := x_H; y_L := 0$. However, they do not provide a proof of non-interference. The approach of Lanet et al. has been refined by Bernardeschi and De Francesco, see e.g. [10], for a subset of the JVM that includes jumps, subroutines but no exceptions. More recently, Bonelli, Compagnoni and Medel [22] have considered non-interference for a simple assembly language, using linear continuations for computing control dependence regions. Their proof technique is similar to ours. Using abstract interpretation techniques, Genaim and Spoto [23] have also developed a sound information flow analysis for a fragment of Java bytecode. Finally, the first and third author [16] have developed a sound information flow type system for a large fragment of the JVM.

Type preserving compilation. Type preserving compilation has been thoroughly studied in the context of typed intermediate languages, most notably for ML and Java, see e.g. [24]. Information flow types preserving compilation has been studied by Zdancewic and Myers in the context of λ-calculus and CPS translation [25]. Also, Honda and Yoshida [5] consider type-preserving interpretations of higher-order imperative calculi with security types to π-calculus with security types.

Proof carrying code and typed assembly languages. Recent work on Proof Carrying Code [14,26] advocates the use of certifying compilation, which is closely related to type preserving compilation in the sense that a certifying compiler aims at producing, from a certificate (i.e. a proof object) that a source program adheres to a property, a certificate that the compiled program adheres to a corresponding property.

Recent work on typed assembly languages [27] aims at endowing assembly languages with a typing system which guarantees such properties as memory safety of programs.

6.2. Future work

Our work constitutes a preliminary investigation in the realm of certifying compilation for security properties, and may be extended in several directions.

- Language expressiveness: we would like to extend the results of this paper to more powerful languages that include objects and/or higher-order functions. We are particularly interested in scaling up our results to the sequential fragment of Java and of the JVM, building up on [7] for the former and on [16] for the latter.
- Integrity: it should be possible, and of practical interest, to adapt our results to integrity. Indeed, weak forms of integrity guarantee that high variables may not be modified by a low writer, and are dual to confidentiality.
- Machine-checked proofs: we would like to machine-check the proof of soundness of the information flow type system of Section 4 and of [16]. Using earlier work on verified bytecode verifiers [28] in Coq [29] (see [30] for similar earlier work in Isabelle [31]), it should be possible to derive a certified bytecode verifier that guarantees secure information flow for a representative (sequential) fragment of the JVM.

Acknowledgements

We are grateful to Florian Kammueller, David Naumann, and to the referees for their comments on the paper.

This work is partially supported by the ACI Sécurité SPOPS, by the RNTL project CASTLES and by the IST project INSPIRED.

Amitabh Basu was supported by an INRIA International Internship during the completion of this work (Summer 2003).

Appendix A. Proof of Lemmas 14 and 15

In the sequel, we use hd and tl to denote the head and tail functions.

Proof of Lemma 14. By a case analysis on the instruction that is executed. We assume that $s_1 = \langle cs, \rho_1, os_1\rangle$ and $s_2 = \langle cs, \rho_2, os_2\rangle$. Further we always choose st_1', se_1' and st_2', se_2' such that $\Gamma, s \cdot \mathsf{cs} \vdash st_1, se_1 \Rightarrow st_1', se_1'$ and such that $\Gamma, s \cdot \mathsf{cs} \vdash st_2, se_2 \Rightarrow st_2', se_2'$.

Case: $P_f[i] = \mathsf{load}\ x$. By the typing transfer rule,

$$st_1' = (\Gamma(x) \sqcup se(f,i)) :: st_1,$$
$$st_2' = (\Gamma(x) \sqcup se(f,i)) :: st_2.$$

By the operational semantics,

$$s_1' = \langle\langle f, i+1\rangle :: cs_1, \rho_1, \rho_1(x) :: os_1\rangle,$$
$$s_2' = \langle\langle f, i+1\rangle :: cs_2, \rho_2, \rho_2(x) :: os_2\rangle.$$

To see $s_1' \sim_{\Gamma, st_1', st_2'} s_2'$, we have to check that $\rho_1(x) \sim_{\Gamma(x)\sqcup se(f,i)} \rho_2(x)$. There are two cases to treat:

- if $se(f,i) \leqslant \Gamma(x)$ then $\Gamma(x) \sqcup se(f,i) = \Gamma(x)$ and we are done by hypothesis.
- if $se(f,i) > \Gamma(x)$ then $\Gamma(x) \sqcup se(f,i) = H$ and $\rho_1(x) \sim_H \rho_2(x)$ holds by definition.

Case: $P_f[i] = \mathsf{store}\ x$. By the typing transfer rule,

$$st_1' = \mathsf{tl}\ st_1,$$
$$st_2' = \mathsf{tl}\ st_2.$$

By the operational semantics,

$$s_1' = \langle\langle f, i+1\rangle :: cs, \rho_1 \oplus \{x \mapsto \mathsf{hd}\ os_1\}, \mathsf{tl}\ os_1\rangle,$$
$$s_2' = \langle\langle f, i+1\rangle :: cs, \rho_2 \oplus \{x \mapsto \mathsf{hd}\ os_2\}, \mathsf{tl}\ os_2\rangle.$$

We must prove:

- $\mathsf{tl}\ os_1 \sim_{st_1', st_2'} \mathsf{tl}\ os_2$, which follows directly from $os_1 \sim_{st_1, st_2} os_2$ and the definition of $\sim$;
- $\mathsf{hd}\ os_1 \sim_{\Gamma(x)} \mathsf{hd}\ os_2$: if $\Gamma(x) = H$ then we are done by definition so assume that $\Gamma(x) = L$. From the typing transfer rule, we know that $\Gamma(x) \geqslant \mathsf{hd}\ st_1$ and $\Gamma(x) \geqslant \mathsf{hd}\ st_2$, hence $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = L$. As $s_1 \sim_{st_1, st_2} s_2$, it follows that $\mathsf{hd}\ os_1 = \mathsf{hd}\ os_2$, and we are done.

Case: $P_f[i] = \mathsf{if}\ j$. By the typing transfer rule,

$$st_1' = \mathsf{lift}_{(\mathsf{hd}\ st_1)}(\mathsf{tl}\ st_1),$$
$$st_2' = \mathsf{lift}_{(\mathsf{hd}\ st_2)}(\mathsf{tl}\ st_2).$$

By the operational semantics,

$$s'_1 = \langle\langle f, j_1\rangle :: cs, \rho_1, \mathsf{tl}\ os_1\rangle,$$
$$s'_2 = \langle\langle f, j_2\rangle :: cs, \rho_2, \mathsf{tl}\ os_2\rangle.$$

We must prove:

- $\mathsf{tl}\ os_1 \sim_{st'_1, st'_2} \mathsf{tl}\ os_2$, which follows from the fact that for every values v, v' and security levels l, l' s.t. $l \leqslant l'$, $v \sim_l v'$ implies $v \sim_{l'} v'$;
- $se = se'_1 = se'_2$ and $j_1 = j_2$, or $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = H$. Assume that the second disjunct does not holds, i.e. $\mathsf{hd}\ st_1 \neq H$ or $\mathsf{hd}\ st_2 \neq H$. Necessarily $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = L$, by definition of $\sim$, and because $\mathsf{hd}\ os_1 = \mathsf{hd}\ os_2$. The result follows.

Cases for $\mathsf{prim}\ op$ and $\mathsf{prim}\ n$ are similar to the case of load and do not present further difficulties. Cases for $\mathsf{goto}\ j$, $\mathsf{call}\ f$ and return are straightforward since states do not change. □

Proof of Lemma 15. *Case*: $P_f[i] = \mathsf{load}\ x$. Assume $s = \langle\langle f, i\rangle :: cs, \rho, os\rangle$. By the typing transfer rule, the operational semantics and the hypothesis $se(f, i) = H$,

$$st' = H :: st,$$
$$s' = \langle\langle f, i+1\rangle :: cs, \rho, \rho(x) :: os\rangle.$$

As $\mathsf{high}\ st$, it follows that $\mathsf{high}\ st'$ and $os \sim_{\Gamma, st, st'} \rho(x) :: os$. The register map remains unchanged, hence $s \sim_{\Gamma, st, st'} s'$.

Case: $P_f[i] = \mathsf{store}\ x$. Assume $s = \langle\langle f, i\rangle :: cs, \rho, v :: os\rangle$. By the typing transfer rule and the operational semantics

$$\Gamma(x) = H,$$
$$st' = \mathsf{tl}\ st,$$
$$s' = \langle\langle f, i+1\rangle :: cs, \rho \oplus \{x \mapsto v\}, os\rangle.$$

As $\mathsf{high}\ st$, it follows that $\mathsf{high}\ st'$ and $v :: os \sim_{st, st'} os$. Furthermore, $\rho \sim_\Gamma \rho \oplus \{x \mapsto v\}$ since $\Gamma(x) = H$, hence we are done.

Case: $P_f[i] = \mathsf{if}\ j$. Assume $s = \langle\langle f, i\rangle : cs, \rho, os\rangle$. By the typing transfer rule and the operational semantics

$$st = k :: st_0,$$
$$st' = \mathsf{lift}_k(st_0),$$
$$se' = \mathsf{lift}_k(se, \mathsf{reg}(f, i)),$$
$$s' = \langle\langle f, l\rangle :: cs, \rho, os\rangle,$$

where $l \in \{i+1, j\}$. Clearly $s \sim_{\Gamma, st, st'} s'$ and $\mathsf{high}\ st'$, as we assume that either $\mathsf{high}\ st$ or $k = H$.

The remaining instructions are similar or straightforward. □

Appendix B. Proof of Theorem 16

Throughout this section, we assume given a fixed program $P \in \mathscr{P}$ and a fixed register type $\Gamma : \mathscr{X} \to \mathscr{S}$.

Proof of Lemma 17. By structural induction on instructions.

Case: $e \equiv x$. By definition, we have that $\mathscr{C}_e(e) = \mathsf{load}\ x$ and $i = j$. Define S^e such that $S^e(\langle f, i\rangle :: cs_0) = st, se$ and such that $S^e(\langle f, i+1\rangle :: cs_0)$ is equal to $(\Gamma(x) \sqcup se(f,i)) :: st, se$. Properties 1 and 2 hold by the transfer rule of load, and because $\Gamma(x) \sqcup se(f,i) = \tau$ as $se(f,i) = \tau$ and $\Gamma(x) \leqslant \tau$ as $\Gamma \vdash e : \tau$.

Case: $e \equiv n$. By definition, $\mathscr{C}_e(e) = \mathsf{prim}\ n$ and $i = j$. Define S^e such that, $S^e(\langle f, i\rangle :: cs_0) = st, se$ and $S^e(\langle f, i+1\rangle :: cs_0) = se(f,i) :: st, se$. Properties 1 and 2 hold by the rule of prim, and because $se(f,i) = \tau$.

Case: $e \equiv e_1\ op\ e_2$. By definition, $\mathscr{C}_e(e) = \mathscr{C}_e(e_1) :: \mathscr{C}_e(e_2) :: \mathsf{prim}\ op$. Now assume that $P_f[i \ldots i_1] = \mathscr{C}_e(e_1)$ and assume that $P_f[i_1+1 \ldots i_2] = \mathscr{C}_e(e_2)$ and $P_f[i_2+1] = \mathsf{prim}\ op$. By induction hypothesis we can construct S^{e_1} for the security type st, se and S^{e_2} for the security type $\tau :: st, se$. Define

$$S^e(\langle f, k\rangle :: cs_0) = \begin{cases} S^{e_1}(\langle f, k\rangle :: cs_0) & \text{if } i \leqslant k \leqslant i_1, \\ S^{e_2}(\langle f, k\rangle :: cs_0) & \text{if } i_1 + 1 \leqslant k \leqslant i_2, \\ \tau \sqcup se(f, i_2) :: st, se & \text{if } k = i_2 + 1. \end{cases}$$

Properties 1 and 2 are derived from the induction hypothesis on S^{e_1} and S^{e_2}. □

In the sequel, we use $\cup$ to denote the union of two partial maps that coincide on the intersection of their domains. By abuse of notation, we write $S^e_{cs_0,st,se}$ for the function $\lambda x \in dom(S^e_{cs_0,st,se}).\ \{S^e_{cs_0,st,se}(x)\}$.

Proof of Lemma 18. Let $\blacktriangleleft$ be the smallest transitive relation containing the subterm relation and the relation $\blacktriangleleft_0$ defined by the clause

$$\frac{[\mathsf{proc}\ f\ \vec{x} := c;\ \mathsf{return}] \text{ is a declaration in } P}{c \blacktriangleleft_0 f\ \vec{e}}.$$

The relation is well-founded, as the type system excludes mutual or self-recursion, so we proceed by well-founded induction on $\blacktriangleleft$.

Case: $c \equiv x := e$. By definition, $\mathscr{C}_c(c) = \mathscr{C}_e(e) :: \mathsf{store}\ x$. Set $S^c = S^e_{cs_0,st_0,se_0}$, where $S^e_{cs_0,st_0,se_0}$ is defined using Lemma 17 (which can be applied because the command in the source language is typable). As $S^e_{cs_0,st_0,se_0}$ satisfies both properties 1 and 2 of Lemma 17, we can conclude.

Case: $c \equiv c_1; c_2$. By definition, $\mathscr{C}_c(c) = \mathscr{C}_c(c_1) :: \mathscr{C}_c(c_2)$. Set $S^c_{cs_0,st_0,se_0} = S^{c_1}_{cs_0,st_0,se_0} \cup S^{c_2}_{cs_0,st'_0,se_0}$, where S^{c_1} and S^{c_2} are defined by induction hypothesis, and $st'_0 = \phi^{c_1}_{cs_0,st_0,se_0}$. The properties follow by induction hypothesis (and by elementary reasoning on the successors in a compiled program).

Case: $c \equiv \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. By definition,

$$\begin{aligned} &\mathscr{C}_c(c) = \mathscr{C}_e(e) :: \mathsf{if}\ (i'+2) :: \mathscr{C}_c(c_1) :: \mathsf{goto}\ (j+1) :: \mathscr{C}_c(c_2), \\ &P_f[i \ldots j'] = \mathscr{C}_e(e), \\ &P_f[j'+2 \ldots i'] = \mathscr{C}_c(c_1), \\ &P_f[i'+2 \ldots j] = \mathscr{C}_c(c_2). \end{aligned}$$

Set

$$\begin{aligned} &S^c_{cs_0,st_0,se_0} \\ &\quad = S^e_{cs_0,st_0,se_0} \cup S^{c_1}_{cs_0,\mathsf{lift}_\tau(st_0),se_0} \cup S^{c_2}_{cs_0,\mathsf{lift}_\tau(st_0),se_0} \cup \{\langle f, i'+1\rangle :: cs_0 \rightharpoonup \mathsf{lift}_\tau(st_0), se_0\}. \end{aligned}$$

The only subtlety is to prove that S^{c_1} and S^{c_2} coincide on $\langle f, j+1\rangle :: cs_0$, which follows by induction hypothesis.

Case: $c \equiv$ while e do c_1. By definition,

$$\mathscr{C}_c(c) = \mathsf{goto}\ (i'+1) :: \mathscr{C}_c(c_1) :: \mathscr{C}_e(e) :: \mathsf{if}\ (i+1),$$
$$P_f[i+1 \ldots i'] = \mathscr{C}_c(c_1),$$
$$P_f[i'+1 \ldots j'] = \mathscr{C}_e(e).$$

Set $S^c_{cs_0,st_0,se_0} = S^e_{cs_0,st_0,se_0} \cup S^{c_1}_{cs_0,\mathsf{lift}_\tau(st_0),se_0} \cup \{\langle f,i\rangle :: cs_0 \rightharpoonup st_0, se_0\}$. The properties follow by induction hypothesis.

Case: $c \equiv f'(e)$ (for simplicity, we assume that f' only has one parameter). By definition $\mathscr{C}_c(c) = \mathscr{C}_e(e) :: \mathsf{call}\ f'$ with $P_f[i \ldots j] = \mathscr{C}_e(e)$. Further assume that f' is defined in P by $\mathsf{proc}\ f'(x) = c';\ \mathsf{return}$. Set

$$S^c_{cs_0,st_0,se_0} = S^e_{cs_0,st_0,se_0} \cup S^{c'}_{\langle f,j+1\rangle::cs,\tau::st_0,se_0}.$$

The only subtlety is to notice that $S^c_{cs_0,st_0,se_0}(\langle f, j+1\rangle :: cs_0)$ is equal to $\{\tau :: st_0, se_0\}$, which follows from Lemma 17.

References

[1] Sabelfeld A, Myers A. Language-based information-flow security. IEEE Journal on Selected Areas in Communications 2003;21:5–19.

[2] Goguen J, Meseguer J. Security policies and security models. In: Proceedings of SOSP'82. Silver Spring, MD: IEEE Computer Society Press; 1982. p. 11–22.

[3] Volpano D, Smith G, Irvine C. A sound type system for secure flow analysis. Journal of Computer Security 1996;167–87.

[4] Heintze N, Riecke J. The SLam calculus: programming with secrecy and integrity. In: Proceedings of POPL'98. New York: ACM Press; 1998. p. 365–77.

[5] Honda K, Yoshida N. A uniform type structure for secure information flow. In: Proceedings of POPL'02. New York: ACM Press; 2002. p. 81–92.

[6] Barthe G, Serpette B. Partial evaluation and non-interference for object calculi. In: Middeldorp A, Sato T, editors. Proceedings of FLOPS'99, Lecture notes in computer science, vol. 1722. Berlin: Springer; 1999. p. 53–67.

[7] Banerjee A, Naumann DA. Secure information flow and pointer confinement in a Java-like language. In: Proceedings of CSFW'02. Silver Spring, MD: IEEE Computer Society Press; 2002.

[8] Myers AC. Jflow: Practical mostly-static information flow control. In: Proceedings of POPL'99. New York: ACM Press; 1999. p. 228–41.

[9] Pottier F, Simonet V. Information flow inference for ML. ACM Transactions on Programming Languages and Systems 2003;25(1):117–58.

[10] Bernardeschi C, De Francesco N. Combining abstract interpretation and model checking for analysing security properties of Java Bytecode. In: Cortesi A, editor. Proceedings of VMCAI'02, Lecture notes in computer science, vol. 2294. Berlin: Springer; 2002. p. 1–15.

[11] Bieber P, Cazin J, Wiels V, Zanon G, Girard P, Lanet J-L. Checking secure interactions of smart card applets: extended version. Journal of Computer Security 2002;10:369–98.

[12] Leroy X. Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning 2003;30(3–4): 235–69.

[13] Abadi M. Secrecy by typing in security protocols. Journal of the ACM 1999;46(5):749–86.

[14] Necula GC. Proof-carrying code. In: Proceedings of POPL'97. New York: ACM Press; 1997. p. 106–19.

[15] Ball T. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. ACM Letters on Programming Languages and Systems 1993;2(1–4):1–16.

[16] Barthe A, Rezk T. Non-interference for a JVM-like language. In: Fähndrich M, editor. Proceedings of TLDI'05. New York: ACM Press; 2005, pp. 103–112.

[17] with J, Palsberg AW, Appel. Modern compiler implementation in Java. 2nd ed., Cambridge: Cambridge University Press; 2002.

[18] Jacobs B, Pieters W, Warnier M. Statically checking confidentiality via dynamic labels, to appear, 2005.
[19] Coglio A. Simple verification technique for complex Java bytecode subroutines. Concurrency and Computation: Practice and Experience 2004;16(7):647–70.
[20] Henrio L, Serpette B. A parameterized polyvariant bytecode verifier. In: Filliatre J-C, editor. Proceedings of JFLA'03, 2003.
[21] Volpano D, Smith G. A type-based approach to program security. In: Bidoit M, Dauchet M, editors. Proceedings of TAPSOFT'97, Lecture notes in computer science, vol. 1214. Berlin: Springer; 1997. p. 607–21.
[22] Bonelli E, Compagnoni A, Medel R. SIFTAL: A typed assembly language for secure information flow analysis. In: Sabelfeld A, editor. Informal Proceedings of FCS'05; 2004.
[23] Genaim S, Spoto F. Information flow analysis for Java bytecode. In Cousot R, editor. Proceedings of VMCAI'05, Lecture notes in computer science, vol. 3385. Berlin: Springer; 2005. p. 346–62.
[24] League C, Shao Z, Trifonov V. Precision in practice: a type-preserving Java compiler. In Hedin G, editor. Proceedings of CC'03, Lecture notes in computer science, vol. 2622. Berlin: Springer; 2003. p. 106–20.
[25] Zdancewic S, Myers A. Secure information flow and CPS. In: Sands D, editor. Proceedings of ESOP'01, Lecture notes in computer science, vol. 2028. Berlin: Springer; 2001. p. 46–61.
[26] Necula GC, Lee P. The design and implementation of a certifying compiler. In: Proceedings of PLDI'98, 1998. p. 333–44.
[27] Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language. In: Proceedings of POPL'98. New York: ACM Press; 1998. p. 85–97.
[28] Barthe G, Dufay G. A tool-assisted framework for certified bytecode verification. In: Proceedings of FASE'04, Lecture notes in computer science, vol. 2984. Berlin: Springer; 2004. p. 99–113.
[29] Coq Development Team. The coq proof assistant user's guide. Version 8.0, January 2004.
[30] Klein G, Nipkow T. Verified bytecode verifiers. Theoretical Computer Science 2002;298(3):583–626.
[31] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL: a proof assistant for higher-order logic, Lecture notes in computer science, vol. 2283. Berlin: Springer; 2002.

Gilles Barthe is senior researcher at INRIA Sophia-Antipolis. He holds a Ph.D. in Mathematics from the University of Manchester, and an habilitation a diriger les Recherches in Computer Science from the University of Nice. His research interests include programming languages, security, and type theory.

Tamara Rezk graduated in Computer Science from the Universidad Nacional de Córdoba, in Argentina. She is a Ph.D. candidate at INRIA Sophia Antipolis (University of Nice), since October 2003, in France. Her research interests include computer security, formal methods and programming languages.