Security Types Preserving Compilation* (Extended abstract)

Gilles Barthe[1], Amitabh Basu[2**], and Tamara Rezk[1]

[1] INRIA Sophia-Antipolis, France {Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr
[2] IIT Delhi, India csu00099@cse.iitd.ernet.in

Abstract. Initiating from the seminal work of Volpano and Smith, there has been ample 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, [16]). Further, security type systems for low-level languages should appropriately relate to their counterparts for high-level languages; however, we are not aware of any study of type-preserving compilers for type systems for information flow.
In answer to these questions, we introduce a security type system for a low-level language featuring jumps and calls, and show that the type system enforces termination-insensitive non-interference. Then, we introduce a compiler from a high-level imperative programming language to our low-level language, and show that the compiler preserves security types.

1 Introduction

Type systems are popular artefacts to enforce safety properties in the context of mobile and embedded code. While such safety properties fail short of providing appropriate guarantees with respect to security policies to which mobile and embedded code must adhere, recent work has demonstrated that type systems are adequate to enforce statically security policies. These works generally focus on confidentiality and in particular on non-interference [7], which ensures confidentiality through the absence of information leakage.

Initiating from the seminal work of Volpano, Smith and Irvine [20], type systems for non-interference have been thoroughly studied in the literature, see e.g. [16] for a survey. However, most works focus on high-level calculi, including λ-calculus, see e.g. [8], π-calculus, see e.g. [9], and ς-calculus [3], or high-level programming languages, including Java [2, 12] and ML [15].

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 existing works, see e.g. [4, 5], use model-checking and abstract

* Work partially supported by IST Projects Profundis and Verificard.
** This work was performed while the author was visiting INRIA Sophia-Antipolis.

interpretation techniques to detect illegal information flows, but do not provide proofs of non-interference for programs that are accepted by their analysis. Thus the first part of this paper is devoted to the definition of a security type system for a low-level language with jumps and calls, and a proof that the type system enforces termination-insensitive non-interference.

Of course, security type systems for low-level languages should appropriately relate to their counterparts for high-level languages. Indeed, one would expect that compilation preserves security typing. Thus the second part of the paper is devoted to a case study in compilation with security types: we define a high-level imperative language with procedures, and a compiler to the low-level language studied in the first part of the paper. Further, we endorse the language with a type system that guarantees termination-insensitive non-interference, and show that compilation function preserves typing. To our best knowledge, there is no previous similar result about type-preserving compilation for security types. 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 [13].

Contents The remaining of the paper is organized as follows. In Section 2 we define an assembly language that shall serve as the compiler target, endorse it with a security type system, and prove that the type system enforces termination-insensitive non-interference. In Section 3, we introduce a high-level imperative language with procedures and its associated type system. Further, we introduce a compiler that we show to preserve security typing; we also discuss how type-preserving compilation can be used to lift non-interference to the high-level language. We conclude in Section 4, with related work and directions for further research.

2 Assembly Language

2.1 Syntax and Operational Semantics

The assembly language is a small language with jumps and procedures. A *program* P is a set of *procedures* with a distinguished, main, procedure; we let P_f be the procedure associated to an identifier f in P. Each procedure P_f consists of an array of instructions; we let $P_f[i]$ be the i-th instruction in P_f. The set Instr of instructions and the set Prog_c of compiled programs are defined in Figure 1. We often denote programs by $P_c :: [f := i^\star]^\star$. Given a program P, we let $\mathcal{PP}$ be its set of *programs points*, i.e. the set of pairs $\langle f, i\rangle$ with $f \in \mathcal{F}$, where $\mathcal{F}$ is a set of procedure names, and $i \in dom(P_f)$. Further, we assume programs to satisfy the usual well-formedness conditions, such as code containment: for every program point $\langle f, i\rangle$, $P_f[i] = \mathsf{if}\ j \ \Rightarrow\ j \in dom(P_f)$, etc.

The operational semantics is given as a transition relation between states. In our setting, values are integers, i.e. $\mathcal{V} = \mathbb{Z}$ and states are triples of the form

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

where op is either a literal $n \in \mathbb{Z}$, or a primitive operation $+, -, \times, \ldots$, or a comparison operation $<, \leq, =, \ldots$; f ranges over a set $\mathcal{F}$ of procedure names, x ranges over a set $\mathcal{X}$ of registers, and j ranges over $\mathbb{N}$.

Fig. 1. Instruction set

$\langle \mathsf{cs}, \rho, s \rangle$ where $\mathsf{cs} \in \mathcal{PP}^\star$ is a *call string* whose length is bounded by some previously agreed upon value max, $\rho : \mathcal{X} \rightarrow \mathcal{V}$ is a *register map* that assigns values to local variables (note that $\mathcal{X}$ is finite for any fixed program), and s is an *operand stack*, i.e. a stack of values. The operational semantics of the assembly language is given by the rules of Figure 2; all rules are subject to the proviso that the size of the call string and the operand stack remain bounded by some previously agreed upon maximal size max and MAX; further we assume given for every operation symbol op a corresponding total binary function $\underline{op}$ on integers. Finally, we write $\rho \oplus \{x \mapsto v\}$ to denote the unique function ρ' s.t. $\rho'(y) = \rho(y)$ if $y \neq x$ and $\rho'(x) = v$.

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.

2.2 Defining non-interference

Informally, non-interference guarantees that, executing a program P on initial states that are indistinguishable from the point of view of an attacker will not result in observable differences for the attacker. There are however a number of different ways in which this definition can be made precise, depending on the formulation of observability. One notion, that seems well adapted to our context, is termination-insensitive non-interference, which says that given any two states i and i', and assuming that executing P on i and i' respectively yield as final states f and f', indistinguishability between i and i' entails indistinguishability between f and f'. Note that such a definition implicitly assumes that an attacker cannot observe termination; there are stronger notions of confidentiality that consider termination and even execution time as observable, see e.g. [16].

Indistinguishability is a relation between states and is defined w.r.t. security maps that assign security levels to registers and to program points; throughout this paper, we assume that the set of security levels is $\mathcal{S} = \{H, L\}$ and is ordered by the clause $L \leq H$; considering a lattice of security levels instead is possible

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

$$\frac{P_f[i] = \mathsf{prim}\ op \qquad op \in \mathbb{O} \qquad n_1\ \underline{op}\ n_2 = n}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, n_1 :: n_2 :: s\rangle \leadsto \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, n :: s\rangle}
\qquad
\frac{P_f[i] = \mathsf{if}\ j \qquad v = 0}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, v :: s\rangle \leadsto \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, s\rangle}$$

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

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

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

Fig. 2. Operational Semantics of Assembly Language

but adds technicalities without adding insight. Indistinguishability on states is defined in terms of indistinguishiblity relations on register maps, and on operand stacks. The former is a pointwise extension of the obvious indistinguishibility relation on values.

Definition 1 (Values and register maps indistinguishability).

- *Value indistinguishability* $v \sim_{SL} v'$ *(of values* v *and* v' *w.r.t. security level* SL*) is defined as* $SL = H \vee v = v'$.
- *The relation is extended pointwise to maps: for* $\rho, \rho' : \mathcal{X} \to \mathcal{V}$ *and* $\Gamma : \mathcal{X} \to \mathcal{S}$, *we have* $\rho \sim_\Gamma \rho'$ *is defined as*

$$\forall x \in \mathcal{X}.\ (\rho\ x) \sim_{(\Gamma\ x)} (\rho'\ x)$$

At this point, it is already possible to define non-interference for programs.

Definition 2. *A program* P *whose main procedure is* main *is* non-interferent *w.r.t.* $\Gamma : \mathcal{X} \to \mathcal{S}$, *written* $\mathsf{NI}_\Gamma(P)$, *if for every* $\rho, \rho', \mu, \mu' : \mathcal{X} \to \mathcal{V}$ *such that* $\rho \sim_\Gamma \rho'$ *and* $\langle\langle \mathrm{main}, 1\rangle, \rho, \epsilon\rangle \leadsto^\star \langle \epsilon, \mu, \epsilon\rangle$ *and* $\langle\langle \mathrm{main}, 1\rangle, \rho', \epsilon\rangle \leadsto^\star \langle \epsilon, \mu', \epsilon\rangle$, *we have* $\mu \sim_\Gamma \mu'$.

Stack indistinguishability requires a slight generalization of the pointwise order on stacks so as to handle high if with branches of different length(a motivation for this is given in section 2.4). The intuition is that we require operand stacks to be indistinguishable point-wise on some common top part, and then to be high in the bottom part on which they may not coincide. High operand stacks are defined relative to a stack type: formally, let s be an operand stack and st be a stack type; we write $\mathsf{high}_{os}\ (s, st)$ if s and st have the same length n and $st[i] = H$ for every $1 \leq i \leq n$.

Definition 3 (Operand stack indistinguishability). *Let s, s' be operand stacks and $st, st' \in \mathcal{ST}$. Then $s \sim_{st,st'} s'$ is defined inductively as follows:*

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

Definition 4 (State indistinguishability). *State indistinguishability $\sigma \sim_{\Gamma,st,st'} \sigma'$ (of states $\sigma = \langle \mathsf{cs}, \rho, s\rangle$ and $\sigma' = \langle \mathsf{cs}', \rho', s'\rangle$ w.r.t. register type Γ and stack types st and st') is defined as $s \sim_{st,st'} s' \wedge \rho \sim_\Gamma \rho'$.*

2.3 Control Dependence Regions

Type systems such as [18, 20] reject programs that yield implicit flows through low assignments in a high branching instruction, e.g. $\mathsf{if}\ y_H\ \mathsf{then}\ x_L := 0\ \mathsf{else}\ x_L := 1$ that yield implicit flows through low assignments in a high branching instruction; in this program, the final value of x_L depends on y_H and thus the program is insecure.

In order to proceed likewise for our assembly language, we must resort to control dependence regions, which identify for every if instruction program points that execute under its control condition. While such control dependence regions are easy to identify in a structured, source language, their computation is slightly more intricate for unstructured assembly languages, but see e.g. [1] for algorithms that compute such regions. For the purpose of this paper, we need not be precise about the exact computation of such control dependence regions. Instead, we define for every program P

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

and assume given two functions $\mathsf{reg} : \mathcal{PP}_{\mathrm{if}} \to \wp(\mathcal{PP})$ that computes the control dependence region of an if and $\mathsf{jun} : \mathcal{PP}_{\mathrm{if}} \to \mathcal{PP}$ that computes the junction point of the two branches of the if. Formally, we assume that for every $\langle f', i'\rangle \in \mathcal{PP}_{\mathrm{if}}$, if $\langle f', i'\rangle \in \mathsf{reg}(\langle f, i\rangle)$ then $\mathsf{reg}(\langle f', i'\rangle) \subseteq \mathsf{reg}(\langle f, i\rangle)$— we latter refer to this property as RIP or region inclusion property—and that for every execution path

$$\langle\langle f, i\rangle :: \mathsf{cs}, \rho, s\rangle \leadsto \langle \mathsf{cs}_1, \rho_1, s_1\rangle \leadsto \ \ldots\ \leadsto \langle \mathsf{cs}_n, \rho_n, s_n\rangle$$

one of the following holds:

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

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

Remark Strictly speaking, the function jun needs not be total. However, we can always extend jun to a total function with the required property by supposing that the last instruction of the main procedure is a return.

2.4 Type System

The type system is defined through an abstract transition system that manipulates stack types, and a security environment that associates to each program point a security level, and is parameterized by a register type $\Gamma : \mathcal{X} \to \mathcal{S}$ that sets the security level of each register. Before giving its formal definition, we motivate the type system on representative examples.

Example 1. Consider the following piece of program, where x_L is a low variable and y_H is a high variable:

load y_H
store x_L

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 piece of program stores in the variable x_L the value held in the variable y_H, and yields a direct information leakage. Our type system prevents such explicit flows by restricting the transition of an instruction store x_L to the case where the type in the top of the stack type is low, whereas the instruction load y_H pushes a H in the stack type.

The security environment $se : \mathcal{PP} \to \mathcal{S}$ detects illicit flows by recording for each program point pp the highest security level of the control dependence influence under which the program point pp is.

Example 2. Consider the following pieces of program, where x_L is a low variable and y_H is a high variable:

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

These pieces of program yield implicit flows, as a test on a high value yields different values for a low variable. Indeed, at program point 8, the low variable x_L contains the value 0 if y_H contains the value 0 or the value 1 if y_H contains a value different from 0. Our type system prevents such information leakage by imposing that the abstract transition rule for a high if instruction sets to H the security level of all program points in its control dependence region, and by rejecting low assignments and returns—unless the return is the last instruction of the procedure being executed— that are performed at high program points.

The abstract transition system is defined by the typing transfer rules in Figure 3. These rules, which capture the transformation of security types information by instructions, are of the form

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}_0 \quad P_f[i] = \text{instruction}}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'}$$

where Γ is a fixed register type, and st, se determines typing constraints for cs, and st', se' determines typing constraints for the successors of cs. Note the transfer function rules define a partial function, i.e. $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st_1, se_1$ and $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st_2, se_2$ implies $st_1 = st_2$ and $se_1 = se_2$.

The successor relation $\mapsto \subseteq \mathcal{CS} \times \mathcal{CS}$, where $\mathcal{CS}$ is the set of $\mathcal{PP}$-lists of length $< \mathsf{max}$, is defined by the clauses:

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

Type-checking is performed by a dataflow analysis that explores all abstract execution paths; following Brisset and Coglio [6], we opt for a polyvariant analysis. Hence our type system deals with security types of the form $\mathcal{CS} \to \wp(\mathcal{ST} \times \mathcal{SE})$, where the set $\mathcal{ST}$ of stack types is defined as the set of $\mathcal{S}$-stacks of length smaller than MAX, and the set $\mathcal{SE}$ of security environments is defined as $\mathcal{PP} \to \mathcal{S}$; given a security type S and a call string cs we let S_{cs} denote $S(\mathsf{cs})$. For the purpose of this paper, we work with judgments of the form $\Gamma, S, \mathsf{cs} \vdash P$ where S is a security type; the typing rule is

$$\frac{\forall st, se \in S_{\mathsf{cs}}.\ \forall \mathsf{cs}' \in \mathcal{CS}.\ \mathsf{cs} \mapsto \mathsf{cs}' \ \Rightarrow\ \exists st', se' \in S_{\mathsf{cs}'}.\ \Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'}{\Gamma, S, \mathsf{cs} \vdash P}$$

(There are standard algorithms to compute S when it exists, see e.g. [11]). Finally, we say that a program P has type S w.r.t. Γ, written $\Gamma, S \vdash P$, if $\Gamma, S, \mathsf{cs} \vdash p$ for all cs s.t. $\langle \mathrm{main}, 1\rangle :: \epsilon \mapsto^\star \mathsf{cs}$, where $\mapsto^\star$ denotes the transitive closure of $\mapsto$. As usual, we say that P is typable w.r.t. Γ, written $\Gamma \vdash P$, if $\Gamma, S \vdash P$ for some S.

2.5 Soundness

Typable programs are non-interferent.

Theorem 1. *If $\Gamma \vdash P$ then $\mathsf{NI}_\Gamma(P)$.*

The idea of the proof is as follows: first, we prove in Lemma 1 that indistinguishability is preserved under one step of execution, if the program is typable. Second, we prove in Lemma 2 that one step execution in a high-level environment yields a result state that is indistinguishable from the original one. By combining these results together, we conclude.

In the sequel, we use $s \cdot \mathsf{cs}$ to denote the call string of a state s and hd to denote the head function. We also write $\mathsf{high}\ st$ if all elements in the $\mathcal{S}$-stack s are high. We also write $\Gamma \vdash \mathsf{cs}, st, se \Rightarrow \mathsf{cs}', st', se'$ if $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'$ and $\mathsf{cs} \mapsto \mathsf{cs}'$, and use $\Gamma \vdash \cdot,\cdot,\cdot \Rightarrow^\star \cdot,\cdot,\cdot$ to denote its transitive closure.

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{load}\ x}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow (\Gamma(x) \sqcup se(f,i)) :: st, se}$$

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

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{prim}\ n}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow se(f,i) :: st, se}$$

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

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

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{return} \quad se(f,i) = L \vee i+1 \notin dom(P_f)}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] \in \{\mathsf{goto}\ j, \mathsf{call}\ f'\}}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st, se}$$

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

Lemma 1 (One-Step Noninterference in Low-Level Environments). *Suppose $\Gamma, S \vdash P$. Let s_1, s_2, s_1', s_2' be states with $s_1 \cdot \mathsf{cs} = s_2 \cdot \mathsf{cs}$ and let $(st_1, se), (st_2, se) \in S_{s_1 \cdot \mathsf{cs}}$ be security types s.t. $s_1 \sim_{\Gamma, st_1, st_2} s_2$, and $s_1 \rightsquigarrow s_1'$, and $s_2 \rightsquigarrow s_2'$. Then there exist $(st_1', se_1') \in S_{s_1' \cdot \mathsf{cs}}$ and $(st_2', se_2') \in S_{s_2' \cdot \mathsf{cs}}$ s.t. $s_1' \sim_{\Gamma, st_1', st_2'} s_2'$. Furthermore, one of the following holds:*

- *$se_1' = se_2' = se$ and $s_1' \cdot \mathsf{cs} = s_2' \cdot \mathsf{cs}$;*
- *$s_1 \cdot \mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}'$ and $P_f[i] = \mathsf{if}\ j$ and $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = H$.*

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

Lemma 2 (One-Step Noninterference in High-Level Environments). *Suppose $\Gamma, S \vdash P$. Let s, s' be states and $(st, se) \in S_{s \cdot \mathsf{cs}}$ be a security type s.t. $s \cdot \mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}'$, and $s \rightsquigarrow s'$, and one of the following holds:*

- *$\mathsf{high}\ st$, and $se(\langle f, i\rangle) = H$;*
- *$P_f[i] = \mathsf{if}\ j$, and $\mathsf{hd}\ st = H$.*

Then there exists $(st', se') \in S_{s' \cdot \mathsf{cs}}$ s.t. $\mathsf{high}\ st'$, and $s \sim_{\Gamma, st, st'} s'$, and $\Gamma, s \cdot \mathsf{cs} \vdash (st, se) \Rightarrow (st', se')$.

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

Proof (of Theorem 1). Consider the two execution paths

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

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

By invoking Lemma 1 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 $s_q \cdot \mathsf{cs} = \epsilon$ then $n_1 = n_2 = q$ and we are done; otherwise, $s_q \cdot \mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}'$ and $P_f[i] = \mathsf{if}\ j$ and $\mathsf{hd}\ st_q = \mathsf{hd}\ st'_q = H$.

We now invoke Lemma 2 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 $s_{q_1} \cdot \mathsf{cs} = \mathsf{jun}(\langle f, i\rangle) :: \mathsf{cs}'$ and $s_{q_2} \cdot \mathsf{cs} = \mathsf{jun}(\langle f, i\rangle) :: \mathsf{cs}'$ 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 1 again, and repeat the process until reaching the final states of the reduction sequences.

3 Security type preserving compilation

In this section, we define a high-level imperative language, endorse 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.

3.1 Source language

The source language is a simple imperative language with procedures. A procedure is a declaration of the form $\mathsf{proc}\ f(\boldsymbol{x}) = c;\ \mathsf{return}$ 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 by the following syntaxes:

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

The operational, big-step semantics of programs is based on judgments of the form $\langle c, \mu\rangle \Longrightarrow \mu'$, where $c \in \mathsf{Comm}$ and $\mu, \mu' : \mathcal{X} \to \mathcal{V}$. Rules are standard, see e.g. [18, 20], and omitted. The security type system is based on judgments of the form $\Gamma \vdash_S e : \tau$ and $\Gamma \vdash_S c : \tau\ cmd$. A program P is typable, written $\Gamma \vdash_S P$, if $\Gamma \vdash_S \mathrm{main} : \tau\ cmd$ for some τ. The typing rules are inspired from [18, 20], and are given in Figure [1] 4; in the last rule, we assume that the procedure f is defined by $\mathsf{proc}\ f\ (\boldsymbol{x}) = P;\ \mathsf{return}$. The typing rules exclude mutual and self-recursion; however it is possible to overcome this limitation at the price of further technicalities.

3.2 Compilation

The compilation function $\mathcal{C}_p : \mathsf{Prog} \to \mathsf{Prog}_c$ is defined in the usual way from a compilation function on expressions $\mathcal{C}_e : \mathsf{Expr} \to \mathsf{Instr}^\star$, and a compilation function on commands $\mathcal{C}_c : \mathsf{Comm} \to \mathsf{Instr}^\star$. Their formal definitions are given in Figure 5. In order to enhance readability, we use :: both for consing an element to a list and concatenating two lists, and we omit details of calculating pc in the clauses for while and if expressions. We also use $\#l$ to denote the length of a list.

3.3 Preservation of security types

Compilation preserves typing.

Theorem 2. *If $\Gamma \vdash_S P$ then $\Gamma \vdash \mathcal{C}_p(P)$.*

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 $\mathcal{C}_e$. By abuse of notation, we write $se = \tau$ if $se(\langle f, j\rangle) = \tau$ for every $\langle f, j\rangle \in \mathcal{PP}$.

Lemma 3. *Assume e is an expression in P and $\Gamma \vdash_S e : \tau$, and $\mathcal{C}_p(P)_f[i \ldots j] = \mathcal{C}_e(e)$. For every $\mathsf{cs}_0 \in \mathcal{CS}$ and $st, se \in \mathcal{ST}$ s.t. $se = \tau$, there exists $S^e_{\mathsf{cs}_0, st, se} : \{\langle f, k\rangle :: \mathsf{cs}_0 \mid i \leq k \leq j+1\} \to \mathcal{ST}$—by abuse of notation, we often write S^e—s.t.:*

1. *for every $\mathsf{cs}, \mathsf{cs}' \in dom(S^e)$, if $\mathsf{cs} \mapsto \mathsf{cs}'$ then $\Gamma, \mathsf{cs} \vdash S^e(\mathsf{cs}) \Rightarrow S^e(\mathsf{cs}')$;*
2. *$S^e(\langle f, j+1\rangle :: \mathsf{cs}_0) = \tau :: st, se$.*

Proof. By structural induction on instructions.

Second, we extend the result to commands.

Lemma 4. *Assume c is a command in P, and $\Gamma \vdash_S c : \tau\ cmd$, and $\mathcal{C}_p(P)_f[i \ldots j] = \mathcal{C}_c(c)$. For every $\mathsf{cs}_0 \in \mathcal{CS}$ and $st_0, se_0 \in \mathcal{ST}$ s.t. $se_0 = \tau$, there exists $S^c_{\mathsf{cs}_0, st_0, se_0} : \mathcal{CS} \rightharpoonup \wp(\mathcal{ST})$—by abuse of notation, we often write S^c—s.t.:*

1. *for every $\mathsf{cs}, \mathsf{cs}' \in dom(S^c)$ and $st, se \in S^c(\mathsf{cs})$ s.t. $\mathsf{cs} \mapsto \mathsf{cs}'$, there exists $st', se' \in S^c(\mathsf{cs}')$ s.t. $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'$;*

[1] For readability, we write $\vdash$ for $\vdash_S$.

2. *there exists* st' *s.t.* $st' = st_0$ *or* $st' = \mathsf{lift}_\tau\ st_0$, *and for every* $\mathsf{cs} \in dom(S^c)$, $\mathsf{cs}' \notin dom(S^c)$ *and* $st, se \in S^c(\mathsf{cs})$ *s.t.* $\mathsf{cs} \mapsto \mathsf{cs}'$, $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se_0$. *We write* $\phi^c_{\mathsf{cs}_0, st_0, se_0}$ *for* st'.

Proof. By structural induction on instructions.

Proof (of Theorem 2). Set $se_0 = \tau$. By construction, the function $S^{c_{\mathrm{main}}}_{\langle \mathrm{main}, 1\rangle : \epsilon, \epsilon, se_0}$ is defined for all cs s.t. $\langle \mathrm{main}, 1\rangle : \epsilon \mapsto^\star \mathsf{cs}$. It is then immediate to conclude.

3.4 Recovering non-interference for the source language

One can also prove that compilation preserves operational semantics.

Proposition 1 (Preservation of semantics). *Let* p *be a program whose main procedure is* $[\mathrm{main} := c_{\mathrm{main}}; \mathsf{return}]$ *and* $\rho : \mathcal{X} \to \mathcal{V}$. *If* $\langle c_{\mathrm{main}}, \rho\rangle \Longrightarrow \mu$ *then the compiled program satisfies* $\langle \langle \mathrm{main}, 1\rangle, \rho, \epsilon\rangle \leadsto^\star \langle \epsilon, \mu, \epsilon\rangle$.

By combining Proposition 1 and Theorem 2 we are able to recover the non-interference result for typable source programs.

Corollary 1 (Non-interference for source language). *Let* P *be a program, let* $\Gamma : \mathcal{X} \to \mathcal{S}$ *and assume that* $\Gamma \vdash P$. *Then* P *is non-interferent w.r.t.* Γ *in the sense that for every* $\rho, \rho', \mu, \mu' : \mathcal{X} \to \mathcal{V}$,

$$(\rho \sim_\Gamma \rho' \wedge \langle c_{\mathrm{main}}, \rho\rangle \Longrightarrow \mu \wedge \langle c_{\mathrm{main}}, \rho'\rangle \Longrightarrow \mu') \Longrightarrow \mu \sim_\Gamma \mu'$$

Proof. By Proposition 1, $\langle \langle \mathrm{main}, 1\rangle, \rho, \epsilon\rangle \leadsto^\star \langle \epsilon, \mu, \epsilon\rangle$ and $\langle \langle \mathrm{main}, 1\rangle, \rho', \epsilon\rangle \leadsto^\star \langle \epsilon, \mu', \epsilon\rangle$. Furthermore $\Gamma \vdash \mathcal{C}_p(P)$ by Theorem 2. Hence $\mathcal{C}_p(P)$ is non-interferent w.r.t. Γ by Theorem 1, and thus $\mu \sim_\Gamma \mu'$ by definition of non-interference.

4 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.

4.1 Related work

As emphasized in the introduction, static enforcement of non-interference through type systems is a well-researched topic, see e.g. [16] for a survey, and we can only comment on some of the most relevant literature.

Procedures and exceptions Non-interference for procedures and exceptions has first been studied (for high-level languages) by Volpano and Smith [19, 18]. Improved type systems for exceptions have been studied by Myers for Java [12] and by Pottier and Simonet for ML [15, 17].

Low-level languages Lanet *et al.*, see e.g. [5], 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. [4], for a subset of the JVM that includes jumps, subroutines but no exceptions.

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. [10]. Yet we do not know of any result about security types preserving compilation from a high-level language to a low-level language. Such a result is being pursued in the MRG project[2], but the focus is on type systems for resource control. Also, Honda and Yoshida [9] consider type-preserving interpretations of higher-order imperative calculi with security types to π-calculus with security types.

Certifying compilation, as advocated by Proof Carrying Code [13], extends the idea of type preserving compilation by 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, see e.g. [14].

4.2 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 [2] for the former and on recent, unpublished, work by the authors for the latter.
- Generality: the main result of this paper is specialized to one particular compilation function that departs from standard compilers, e.g. by not being optimizing. We would like to isolate a set of constraints that guarantees preservation of typability for security types, and investigate the impact of standard compiler optimizations on security types.
- Integrity: it is of practical interest, and we believe straightforward, 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.

[2] See `http://www.dcs.ed.ac.uk/home/mrg`

References

1. T. Ball. 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*, 2(1–4):1–16, March–December 1993.
2. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.
3. G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, 1999.
4. C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15, 2002.
5. P. Bieber, J. Cazin, V.Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
6. A. Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *Proceedings of FTFJP'02*, 2002.
7. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society, 1982.
8. N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of POPL'98*, pages 365–377. ACM Press, 1998.
9. K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002.
10. C. League, Z. Shao, and V. Trifonov. Precision in Practice: A Type-Preserving Java Compiler. In G. Hedin, editor, *Proceddings of CC'03*, volume 2622 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag, 2003.
11. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
12. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
13. G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
14. G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344, 1998.
15. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
16. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
17. V. Simonet. Fine-grained Information Flow Analysis for a Lambda-Calculus with Sum Types. In *Proceedings of CSFW'02*, pages 223–237, 2002.
18. D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings ofTAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.
19. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of CSFW'97*, pages 156–168. IEEE Press, 1997.
20. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, pages 167–187, December 1996.

$$\text{(SUB)}_e \ \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \qquad \text{(SUB)}_c \ \frac{\Gamma \vdash P : \tau\ cmd \quad \tau' \leq \tau}{\Gamma \vdash P : \tau'\ cmd}$$

$$\text{(VAR)} \ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(VAL)} \ \Gamma \vdash n : \tau$$

$$\text{(OP)} \ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ op\ e' : \tau} \qquad \text{(ASSIGN)} \ \frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau\ cmd}$$

$$\text{(SEQ)} \ \frac{\Gamma \vdash P : \tau\ cmd \quad \Gamma \vdash Q : \tau\ cmd}{\Gamma \vdash P ; Q : \tau\ cmd} \qquad \text{(WHILE)} \ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash P : \tau\ cmd}{\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ P : \tau\ cmd}$$

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

$$\text{(APP)} \ \frac{\Gamma \vdash P : \tau\ cmd \quad \Gamma \vdash \boldsymbol{e} : \tau \quad \Gamma(\boldsymbol{x}) = \tau}{\Gamma \vdash f(\boldsymbol{e}) : \tau\ cmd}$$

Fig. 4. Typing rules for high-level language

$$\begin{aligned}
\mathcal{C}_e(x) &= \mathsf{load}\ x \\
\mathcal{C}_e(n) &= \mathsf{prim}\ n \\
\mathcal{C}_e(e\ op\ e') &= \mathcal{C}_e(e) :: \mathcal{C}_e(e') :: \mathsf{prim}\ op \\
\mathcal{C}_c(x := e) &= \mathcal{C}_e(e) :: \mathsf{store}\ x \\
\mathcal{C}_c(f\ \boldsymbol{e}) &= \mathcal{C}_e(\boldsymbol{e}) :: \mathsf{call}\ f \\
\mathcal{C}_c(c_1;\ c_2) &= \mathcal{C}_e(c_1) :: \mathcal{C}_e(c_2) \\
\mathcal{C}_c(\mathsf{while}\ e\ \mathsf{do}\ c) &= \ let\ l_1 = \mathcal{C}_e(e); l_2 = \mathcal{C}_c(c); x = \#l_2; y = \#l_1\ in \\
&\qquad \mathsf{goto}\ (pc + x + 1) :: l_2 :: l_1 :: \mathsf{if}\ (pc - x - y) \\
\mathcal{C}_c(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) &= \ let\ l_e = \mathcal{C}_e(e); lc_1 = \mathcal{C}_c(c_1); lc_2 = \mathcal{C}_c(c_2); \\
&\qquad x = \#lc_2; y = \#lc_1\ in \\
&\qquad l_e :: \mathsf{if}\ (pc + x + 2) :: lc_2 :: \mathsf{goto}\ (pc + y + 1) :: lc_1 \\
\mathcal{C}_p[\mathsf{proc}\ f(\boldsymbol{x}) := c;\ \mathsf{return}]^\star &= [f := (\mathsf{store}\ \boldsymbol{x} :: \mathcal{C}_c(c) :: \mathsf{return})]^\star
\end{aligned}$$

Fig. 5. Compilation of expressions and commands