Security Types Preserving Compilation
(Extended abstract)

Gilles Barthe\textsuperscript{1}, Amitabh Basu\textsuperscript{2}\textsuperscript{**}, and Tamara Rezk\textsuperscript{1}

\textsuperscript{1} INRIA Sophia-Antipolis, France \{Gilles.Barthe,Tamara.Rezk\}@sophia.inria.fr
\textsuperscript{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 c-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

\textsuperscript{*} Work partially supported by IST Projects Profundis and Veriﬁcard.
\textsuperscript{**} 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. 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 \( \text{Instr} \) of instructions and the set \( \text{Prog}_c \) of compiled programs are defined in Figure 1. We often denote programs by \( P_c ::= [f := i]^* \). Given a program \( P \), we let \( \text{PP} \) be its set of programs points, i.e. the set of pairs \( (f, i) \) with \( f \in \mathcal{F} \), where \( \mathcal{F} \) is a set of procedure names, and \( i \in \text{dom}(P_f) \). Further, we assume programs to satisfy the usual well-formedness conditions, such as code containment: for every program point \( (f, i) \), \( P_f[i] = \text{if } j \Rightarrow j \in \text{dom}(P_f) \), etc.

The operational semantics is given as a transition relation between states. In our setting, values are integers, i.e. \( V = \mathbb{Z} \) and states are triples of the form \( \langle \text{cs}, p, s \rangle \) where \( \text{cs} \in \text{PP}^* \) is a call string whose length is bounded by some
\[ i ::= \text{prim op \ primitive value/operation} \\
| \text{load } x \text{ load value of } x \text{ on stack} \\
| \text{store } x \text{ store top of stack in } x \\
| \text{if } j \text{ conditional jump} \\
| \text{goto } j \text{ unconditional jump} \\
| \text{call } f \text{ procedure call} \\
| \text{return return} \]

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 \( F \) of procedure names, \( x \) ranges over a set \( \mathcal{X} \) of registers, and \( j \) ranges over \( \mathbb{N} \).

\textbf{Fig. 1. Instruction Set}

Previously agreed upon 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 \( \text{max} \) and \( \text{MAX} \); further we assume given for every operation symbol \( op \) a corresponding total binary function \( op \) on integers.

Finally, we write \( \rho \circ \{x \mapsto v\} \) to denote the unique function \( \rho' \) 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.

\textbf{Fig. 2. Operational Semantics of Assembly Language}
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 $S = \{H, L\}$ and is ordered by the clause $L \leq H$; considering a lattice of security levels instead is possible but adds technicalities without adding insight. Indistinguishability on states is defined in terms of indistinguishability relations on register maps, and on operand stacks. The former is a pointwise extension of the obvious indistinguishability 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 $SL = H \lor v = v'$.
- The relation is extended pointwise to maps: for $\rho, \rho' : X \rightarrow V$ and $\Gamma : X \rightarrow S$, we have $\rho \sim_{\Gamma} \rho'$ is defined as

\[
\forall x \in X. \ (\rho \ x) \sim_{(\Gamma \ x)} (\rho' \ x)
\]

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. $\Gamma : X \rightarrow S$, written $NI_{\Gamma}(P)$, if for every $\rho, \rho', \mu, \mu' : X \rightarrow V$ such that $\rho \sim_{\Gamma} \rho'$ and $(\langle \text{main}, 1 \rangle, \rho, \epsilon) \sim^* (\epsilon, \mu, \epsilon)$ and $(\langle \text{main}, 1 \rangle, \rho', \epsilon) \sim^* (\epsilon, \mu', \epsilon)$, 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 $\text{high}_{\alpha s} (s, st)$ 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 \( st, st' \in ST \). Then \( s \sim_{st, st'} s' \) is defined inductively as follows:

\[
\begin{align*}
\text{high}_o(s, st) & \rightarrow \text{high}_o(s', st') \\
\text{high}_o(s, st) & \rightarrow \text{high}_o(s', st') \\
\text{high}_o(s, st) & \rightarrow \text{high}_o(s, st') \\
\text{high}_o(s, st) & \rightarrow \text{high}_o(s', st') \\
\end{align*}
\]

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

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.

\[
\text{if } y \in H \text{ then } x \in L := 0 \text{ else } x \in L := 1
\]

that yield implicit flows through low assignments in a high branching instruction; in this program, the final value of \( x \in L \) depends on \( y \in H \) 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 \( \text{if} \) 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 \( P \)

\[
\mathcal{PP}_d = \{ (f, i) \in \mathcal{PP} \mid p_f[i] = \text{if} \}
\]

and assume given two functions \( \text{reg} : \mathcal{PP}_d \rightarrow \mathcal{O}(\mathcal{PP}) \) that computes the control dependence region of an \( \text{if} \) and \( \text{jun} : \mathcal{PP}_d \rightarrow \mathcal{PP} \) that computes the junction point of the two branches of the \( \text{if} \). Formally, we assume that

for every \( (f', i') \in \mathcal{PP}_d \), if \( (f', i') \in \text{reg}((f, i)) \) then \( \text{reg}((f', i')) \subseteq \text{reg}((f, i)) \)—we latter refer to this property as RIP or region inclusion property—and that for every execution path

\[
\langle (f, i) : \text{cs}, \rho, s \rangle \leadsto \langle \text{cs}_1, \rho_1, s_1 \rangle \leadsto \ldots \leadsto \langle \text{cs}_n, \rho_n, s_n \rangle
\]

one of the following holds:

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

We shall use the function \( \text{reg} \) in the type system, and the assumption about execution paths in the proof of non-interference.

Remark Strictly speaking, the function \( \text{jun} \) needs not be total. However, we can always extend \( \text{jun} \) to a total function with the required property by supposing that the last instruction of the main procedure is a \text{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 $\Gamma : \mathcal{X} \rightarrow \mathcal{S}$ 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:

\[
\begin{align*}
\text{load } y_H \\
\text{store } x_L
\end{align*}
\]

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 $\text{store } x_L$ to the case where the type in the top of the stack type is low, whereas the instruction $\text{load } y_H$ pushes a $H$ in the stack type.

The security environment $se : \mathcal{PP} \rightarrow \mathcal{S}$ 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:

\[
\begin{align*}
1 & \text{ load } y_H & 1 & \text{ prim } 0 \\
2 & \text{ if } 6 & 2 & \text{ store } x_L \\
3 & \text{ prim } 0 & 3 & \text{ load } y_H \\
4 & \text{ store } x_L & 4 & \text{ if } 6 \\
5 & \text{ goto } 8 & 5 & \text{ return} \\
6 & \text{ prim } 1 & 6 & \text{ prim } 1 \\
7 & \text{ store } x_L & 7 & \text{ store } x_L \\
8 & \text{ return} & 8 & \text{ return}
\end{align*}
\]

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

\[
\begin{align*}
\text{cs} = (f, i) :: \text{cs}_0 & \quad P_f[i] = \text{instruction} \\
\Gamma, \text{cs} \vdash \text{st}, \text{se} & \Rightarrow \text{st}', \text{se}'
\end{align*}
\]
where $\Gamma$ 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. $\Gamma, cs \vdash st, se \Rightarrow st_1, se_1$ and $\Gamma, cs \vdash st, se \Rightarrow st_2, se_2$ implies $st_1 = st_2$ and $se_1 = se_2$.

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

- if $P_f[i] = \text{return}$ then $(f, i) :: (f', j) :: cs' \mapsto (f', j+1) :: cs'$ and $(f, i) :: \epsilon \mapsto \epsilon$;
- if $P_f[i] = \text{call} f'$ then $(f, i) :: cs \mapsto (f', 1) :: (f, i) :: cs$;
- if $P_f[i] = \text{goto} j$ then $(f, i) :: cs \mapsto (f, j) :: cs$;
- if $P_f[i] = \text{if } j \text{ then } (f, i) :: cs \mapsto (f, k) :: cs$ for $k \in \{i + 1, j\}$;
- otherwise, $(f, i) :: cs \mapsto (f, i + 1) :: cs$.

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 $CS \rightarrow \mathcal{O}(ST \times SE)$, where the set $ST$ of stack types is defined as the set of $S$-stacks of length smaller than $\text{MAX}$, and the set $SE$ of security environments is defined as $PP \rightarrow S$; given a security type $S$ and a call string $cs$ we let $S_{cs}$ denote $S(cs)$. For the purpose of this paper, we work with judgments of the form $\Gamma, S, cs \vdash P$ where $S$ is a security type; the typing rule is

\[
\forall st, se \in S_{cs}. \forall cs' \in CS. cs \mapsto cs' \Rightarrow \exists st', se' \in S_{cs'}. \Gamma, cs \vdash st, se \Rightarrow st', se' \Gamma, S, cs \vdash P
\]

(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. $\Gamma$, written $\Gamma, S \vdash P$, if $\Gamma, S, cs \vdash p$ for all $cs$ s.t. $(\text{main}, 1) :: \epsilon \mapsto^* cs$, where $\mapsto^*$ denotes the transitive closure of $\mapsto$. As usual, we say that $P$ is typable w.r.t. $\Gamma$, 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 $\text{NI}_{\Gamma}(P)$.

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 $s \cdot cs$ to denote the call string of a state $s$ and $\text{hd}$ to denote the head function. We also write $\text{high } st$ if all elements in the $S$-stack $st$ are high. We also write $\Gamma \vdash cs, st, se \Rightarrow cs', st', se'$ if $\Gamma, cs \vdash st, se \Rightarrow st', se'$ and $cs \mapsto cs'$, and use $\Gamma \vdash \cdot, \cdot, \cdot \Rightarrow^* \cdot, \cdot, \cdot$ to denote its transitive closure.
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 $s_1 :: cs = s_2 :: cs$ and let $(st_1, se), (st_2, se) \in S_1::cs$ be security types s.t. $s_1 \sim_{\Gamma, st_1, st_2} s_2$, and $s_1 \sim_{\Gamma} s_1'$, and $s_2 \sim_{\Gamma} s_2'$.

Then there exist $(st_1', se_1') \in S_1'::cs$ and $(st_2', se_2') \in S_2'::cs$ s.t. $s_1' \sim_{\Gamma, st_1', st_2'} s_2'$, and let $s_1 :: cs = s_2 :: cs$.

Furthermore, one of the following holds:

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

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 $(st, se) \in S_0::cs$ be a security type s.t. $s :: cs = \langle f, i \rangle :: cs'$, $s \sim_{\text{high}} s'$, high $st$, and $se((f, i)) = H$; Then there exists $(st', se') \in S_0'::cs$ s.t. high $st'$, and $s \sim_{\Gamma, st, st'} s'$, and $\Gamma, s :: cs \vdash (st, se) \Rightarrow (st', se')$.

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

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

\[ s_0 \sim s_1 \sim \cdots \sim s_{p_1} \]
\[ s'_0 \sim s'_1 \sim \cdots \sim s'_{p_2} \]
where \( s_0 = \langle \text{main}, 1 \rangle :: \epsilon, \rho, \epsilon \), and \( s'_0 = \langle \text{main}, 1 \rangle :: \epsilon, \rho', \epsilon \), and \( s_{n_1} \cdot \text{cs} = s_{n_2} \cdot \text{cs} = \epsilon \).

By invoking Lemma 1 as long as it applies, we conclude for some maximal \( q \) that \( s_q \cdot \text{cs} = s'_q \cdot \text{cs} \), and that there exists \( (st_q, se), (st'_q, se') \in S_{s_q} \cdot \text{cs} \) such that \( s'_q \sim_{\Gamma, st_q, st'_q} s_q \). Now there are two cases to treat: if \( s_q \cdot \text{cs} = \epsilon \) 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, it holds high \( st_q \) and high \( st'_q \). We now invoke Lemma 2 repeatedly to conclude that there exists \( s_{q_1} \) and \( s'_{q_2} \) and \( (st_{q_1}, se_1) \in S_{s_{q_1}} \cdot \text{cs} \) and \( (st'_{q_2}, se'_2) \in S_{s'_{q_2}} \cdot \text{cs} \) 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 \( s_{q_1} \cdot \text{cs} = \text{jun}((f, i)) :: \text{cs}' \) and \( s_{q_2} \cdot \text{cs} = \text{jun}((f, i)) :: \text{cs}' \) respectively. As \( \Gamma \vdash s_q \cdot \text{cs}, st_q, se \Rightarrow s_{q_1} \cdot \text{cs}, st_{q_1}, se_1 \) and \( \Gamma \vdash s_q \cdot \text{cs}, st'_{q_2}, se' \Rightarrow s'_{q_2} \cdot \text{cs}, st'_{q_2}, se'_2 \) 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 \( \text{proc } f(x) = c; \text{ return} \) 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 \( \text{Expr} \) of expressions, \( \text{Comm} \) of commands, and \( \text{Prog} \) of programs are given by the following syntaxes:

\[
\begin{align*}
e & ::= x \mid n \mid e \ op \ e \\
c & ::= x := e \mid f(e) \mid c; c \mid \text{while } e \text{ do } c \mid \text{if } e \text{ then } c \text{ else } c \\
P & ::= [\text{proc } f(x) = c; \text{ return}]^*
\end{align*}
\]

The operational, big-step semantics of programs is based on judgments of the form \( \langle c, \mu \rangle \Rightarrow \mu' \), where \( c \in \text{Comm} \) and \( \mu, \mu' : \mathcal{X} \rightarrow \mathcal{V} \). 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 \( \Gamma \vdash_S c : \tau \text{ cmd} \). A program \( P \) is typable, written \( \Gamma \vdash_S P \), if \( \Gamma \vdash_S \text{main} : \tau \text{ cmd} \) for some \( \tau \). The typing rules are inspired from [18,
20], and are given in Figure 1; in the last rule, we assume that the procedure \( f \) is defined by \( \text{proc } f \ (x) = P; \text{return} \). 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 \( C_p : \text{Prog} \rightarrow \text{Prog}_c \) is defined in the usual way from a compilation function on expressions \( C_e : \text{Expr} \rightarrow \text{Instr}^e \), and a compilation function on commands \( C_c : \text{Comm} \rightarrow \text{Instr}^c \). 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 \( \Gamma \vdash C_p(P) \).

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 \( C_e \). By abuse of notation, we write \( se = \tau \) if \( se((f, j)) = \tau \) for every \( (f, j) \in PP \).

**Lemma 3.** Assume \( e \) is an expression in \( P \) and \( \Gamma \vdash_S e : \tau \), and \( C_p(P)[i \ldots j] = C_e(e) \). For every \( cs_0 \in CS \) and \( st, se \in ST \) s.t. \( se = \tau \), there exists \( S^e_{cs_0, st, se} : \{(f, k) :: cs_0 \mid i \leq k \leq j + 1\} \rightarrow ST \) — by abuse of notation, we often write \( S^e \) — s.t.:

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

**Proof.** By structural induction on instructions.

Second, we extend the result to commands.

**Lemma 4.** Assume \( c \) is a command in \( P \), and \( \Gamma \vdash_S c : \tau \text{ cmd} \), and \( C_p(P)[i \ldots j] = C_e(c) \). For every \( cs_0 \in CS \) and \( st_0, se_0 \in ST \) s.t. \( se_0 = \tau \), there exists \( S^c_{cs_0, st_0, se_0} : CS \rightarrow \mathcal{O}(ST) \) — by abuse of notation, we often write \( S^c \) — s.t.:

1. for every \( cs, cs' \in \text{dom}(S^c) \) 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. \( st' = \text{lift}_\tau st_0 \), and for every \( cs \in \text{dom}(S^c) \), \( cs' \notin \text{dom}(S^c) \) and \( st, se \in S^c(cs) \) s.t. \( cs \mapsto cs' \), \( \Gamma, cs \vdash st, se \Rightarrow st', se_0 \). We write \( \nu'_{cs_0, st_0, se_0} \) for \( st' \).

**Proof.** By structural induction on instructions.

**Proof (of Theorem 2).** Set \( se_0 = \tau \). By construction, the function \( S^{\text{main}}_{\text{main}, 1} : c, c, se_0 \) is defined for all \( cs \) s.t. \((\text{main}, 1) : c \mapsto S^c \). It is then immediate to conclude.

---

1 For readability, we write \( \vdash \) for \( \vdash_S \).
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 $\text{main} := c_{\text{main}}; \text{return}$ and $\rho : \mathcal{X} \rightarrow \mathcal{V}$. If $\langle c_{\text{main}}, \rho \rangle \Rightarrow (\mu, \epsilon)$ then the compiled program satisfies $\langle (\text{main}, 1), \rho, \epsilon \rangle \Rightarrow^* (\epsilon, \mu, \epsilon)$.

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 $\Gamma : \mathcal{X} \rightarrow \mathcal{S}$ and assume that $\Gamma \vdash_S P$. Then $P$ is non-interferent w.r.t. $\Gamma$ in the sense that for every $\rho, \rho', \mu, \mu' : \mathcal{X} \rightarrow \mathcal{V}$,

$$(\rho \sim_{\Gamma} \rho' \land \langle c_{\text{main}}, \rho \rangle \Rightarrow (\epsilon, \mu, \epsilon) \Rightarrow (\epsilon, \mu', \epsilon)).$$

**Proof.** By Proposition 1, $\langle (\text{main}, 1), \rho, \epsilon \rangle \Rightarrow^* (\epsilon, \mu, \epsilon)$ and $\langle (\text{main}, 1), \rho', \epsilon \rangle \Rightarrow^* (\epsilon, \mu', \epsilon)$. Furthermore $\Gamma \vdash_{\mathcal{C}_p}(P)$ by Theorem 2. Hence $\mathcal{C}_p(P)$ is non-interferent w.r.t. $\Gamma$ 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]. Information flow types preserving compilation has been studied by Zwandewic and Myers in the context of $\lambda$-calculus and CPS translation [?]. Also, Honda and Yoshida [9] consider type-preserving interpretations of higher-order imperative calculi with security types to $\pi$-calculus with security types. A similar result for resource control is being pursued in the MRG project\(^2\).

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.

### References


\(^2\) See [http://www.dcs.ed.ac.uk/home/mrg](http://www.dcs.ed.ac.uk/home/mrg)


Fig. 4. Typing rules for high-level language

\[
\begin{align*}
\text{(Sub)} & \quad \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \\
\text{(VAR)} & \quad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : \tau} \\
\text{(VAL)} & \quad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash n : \tau} \\
\text{(OP)} & \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \circ e' : \tau} \\
\text{(Assign)} & \quad \frac{\Gamma \vdash x := e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau} \\
\text{(Seq)} & \quad \frac{\Gamma \vdash P : \tau \text{ cmd} \quad \Gamma \vdash Q : \tau \text{ cmd}}{\Gamma \vdash P;Q : \tau \text{ cmd}} \\
\text{(While)} & \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash P : \tau \text{ cmd}}{\Gamma \vdash \text{while} e \text{ do} P : \tau \text{ cmd}} \\
\text{(Cond)} & \quad \frac{\Gamma \vdash \text{if} e \text{ then} P \text{ else} Q : \tau \text{ cmd}}{\Gamma \vdash \text{if} e \text{ then} P \text{ else} Q : \tau \text{ cmd}} \\
\text{(App)} & \quad \frac{\Gamma \vdash f(e) : \tau \text{ cmd}}{\Gamma \vdash f(e) : \tau \text{ cmd}}
\end{align*}
\]

Fig. 5. Compilation of expressions and commands

\[
\begin{align*}
C_e(x) &= \text{load } x \\
C_e(n) &= \text{prim } n \\
C_e(e \circ e') &= C_e(e) :: C_e(e') :: \text{prim } \circ \\
C_e(x := e) &= C_e(e) :: \text{store } x \\
C_e(f \ e) &= C_e(e) :: \text{call } f \\
C_e(c_1; c_2) &= C_e(c_1) :: C_e(c_2) \\
C_e(\text{while } e \text{ do } c) &= \text{let } l_1 = C_e(e); l_2 = C_e(c); x = \#l_2; y = \#l_1 \text{ in} \\
& \quad \text{goto (pc + x + 1) :: l_2 :: if (pc - x - y)} \\
C_e(\text{if } e \text{ then } c_1 \text{ else } c_2) &= \text{let } l_x = C_e(e); l_{c_1} = C_e(c_1); l_{c_2} = C_e(c_2); x = \#l_{c_2}; y = \#l_{c_1} \text{ in} \\
& \quad l_x :: \text{if (pc + x + 2) :: l_{c_2} :: goto (pc + y + 1) :: l_{c_1}} \\
C_p(\text{proc } f(x) := c; \text{return}*) &= [f := \text{store } x :: C_e(c) :: \text{return}]^*
\end{align*}
\]