Compilation of certificates

Gilles BARTHE Benjamin GRÉGOIRE Tamara REZK
INRIA Sophia-Antipolis Méditerranée, France

Abstract. The purpose of the course is to stress the importance of verification methods for bytecode, and to establish a strong relation between verification at source and bytecode levels. In order to illustrate this view, we shall consider the example of verification condition generation, which underlies many verification tools and plays a central role in Proof Carrying Code infrastructure, and information flow type systems, that are used to enforce confidentiality policies in mobile applications.

Keywords. Information flow typing, program verification, preservation of typing, preservation of proof obligations, proof carrying code

1. Introduction

Reliability and security of executable code is an important concern in many application domains, and in particular mobile code where code consumers run code that originate from untrusted and potentially malicious producers. While many formal techniques and tools have been developed to address this concern, it is particularly striking to notice that many of them operate on source programs, rather than on executable code; the extended static checking tool ESC/Java [6] and the information flow aware language Jif [14] are prominent examples of environments that provide guarantees about source (Java) programs. However, source code verification is not appropriate in the context of mobile code, where code consumers need automatic and efficient verification procedures that can be run locally on executable code and that dispense them from trusting code producers (that are potentially malicious), networks (that may be controlled by an attacker), and compilers (that may be buggy).

Proof Carrying Code (PCC) [17,15,16] is a security architecture for mobile code that targets automated verification of executable code and therefore does not require trust in the code producer, nor in the compiler, nor in the network. In a typical PCC architecture, programs are compiled with a certifying compiler that returns, in addition to executable code, evidence that the code satisfies a desired policy. The evidence is provided in the form of formal objects, that can be used by automatic verification procedures to verify independently that the compiled code is indeed compliant to the policy. Typically, a certifying compiler will generate both program annotations, which specify loop invariants tailored towards the desired policy, as well as proof objects, a.k.a. certificates, that the program is correct w.r.t. its specification. Early work on PCC is based on verification condition generation and exploits the Curry-Howard isomorphism to reduce proof checking to type checking; thus, upon reception of an annotated program with a certificate, the code consumer will automatically extract a set of verification conditions $\phi_1 \dots \phi_n$ using

a verification condition generator and will establish the validity of these conditions using the certificate, which should be a tuple $(M_1, \ldots, M_n)$ of λ-terms such that $M_i : \phi_i$ for $i = 1 \ldots n$.

The idea of certifying compilation is also present in typed low-level languages, where the certificates take the form of type annotations, and certificate checkers are type systems that reject all executable code that does not comply with the consumer policy. Typed assembly languages [19] and the Java Virtual Machine (JVM) [13] provide two well-known examples of typed low-level languages, in which programs come equipped with type information. In the case of the JVM, the type annotations refer to the signature of methods, the type of fields and local variables, *etc*. These type annotations are used by the bytecode verifier, which performs a dataflow analysis to ensure adherence to the JVM safety policy (no arithmetic on references, no stack underflow or overflow, correct initialization of objects before accessing them, *etc*) [12]. Lightweight bytecode verification [18] extends the idea of bytecode verification by requiring that programs also come with additional typing information (concretely the stack type at at junction points) in order to enable a verification procedure that analyzes the program in one pass.

Through their associated verification mechanisms for executable code, infrastructures based on certifying compilers and typed low-level languages suitably address the security concerns for mobile code. Nevertheless, current instances of certifying compilers mostly focus on basic safety policies and do not take advantage of the existing methods for verifying source code. Ideally, one would like to develop expressive verification methods for executable code and to establish their adequacy with respect to verification methods for source programs, so as to be able to transfer evidence from source programs to executable code. The purpose of these notes is to present two exemplary enforcement mechanisms for executable code, and to show how they connect to similar enforcement mechanisms for source code.

The first mechanism aims at ensuring information flow policies for confidentiality: it is a type-based mechanism, compatible with the principles of bytecode verification. We show that the type system is sound, i.e. enforce non-interference of typable programs, and that source programs that are typable in a standard type system for information flow are compiled into programs that are typable in our type system. The benefits of type preservation are two-fold: they guarantee program developers that their programs written in an information flow aware programming language will be compiled into executable code that will be accepted by a security architecture that integrates an information flow bytecode verifier. Conversely, they guarantee code consumers of the existence of practical tools to develop applications that will provably meet the policy enforced by their information flow aware security architecture.

The second mechanism aims at ensuring adherence of programs to a logical specification that establishes their functional correctness or their adherence to a given policy: it is a verification condition generator for programs specified with logical annotations (pre-conditions, post-conditions, etc) compatible with Proof Carrying Code. We show that the verification condition generator is sound, i.e. a program meets its specification, expressed as a pre- and post-condition, provided all verification conditions are valid, and that verification conditions are preserved by compilation. Since verification conditions for source programs and their compilation are syntactically equal (and not merely logically equivalent), one can reuse directly certificates for source programs and bundle them with the compiled program. Preservation of proof obligations provide benefits similar to

operations	$op ::= + \mid - \mid \times \mid /$		
comparisons	$cmp ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$		
expressions	$e ::= x \mid c \mid e\ op\ e$		
tests	$t ::= e\ cmp\ e$		
instructions	$i ::= x := e$		assignment
	$\mid$	$\mathsf{if}(t)\{i\}\{i\}$	conditional
	$\mid$	$\mathsf{while}(t)\{i\}$	loop
	$\mid$	$i; i$	sequence
	$\mid$	skip	skip
	$\mid$	$\mathsf{return}\ e$	return value

where $c \in \mathbb{Z}$ and $x \in \mathcal{X}$.

Figure 1. INSTRUCTION SET FOR BASIC LANGUAGE

preservation of typing and extends the applicability of PCC by offering a means to transfer evidence from source code to executable code and thus to certify complex policies of executable code using established verification infrastructure at source level.

Contents The relation between source verification and verification of executable code is established in the context of a small imperative and assembly languages, and for a non-optimizing compiler, all presented in Section 2. Section 3 is devoted to information flow whereas Section 4 is devoted to verification condition generation. Section 5 discusses the impact of program optimizations on our results, and mentions the main difficulties in extending our results to more realistic settings.

2. Setting

Although the results presented in these notes have been developed for a sequential fragment of the Java Virtual Machine that includes objects, exceptions, and method calls (see Section 5), we base our presentation on a simple imperative language, which is compiled to a stack-based virtual machine.

This section introduces the syntax and the semantics of these simple source and bytecode languages. In addition, we define a non-optimizing compiler, which in the later sections will be shown to preserve information flow typing as well as verification conditions.

Both the source and bytecode languages use named variables taken from a fixed set $\mathcal{X}$, and manipulate memories, i.e. mappings from variables to values. In our setting, values are just integers, thus a memory ρ has type $\mathcal{X} \to \mathbb{Z}$. We denote by $\mathcal{L}$ the set of memories.

2.1. The source language: IMP

Programs Figure 1 defines the basic source language IMP. We let $\mathcal{E}$ be the set of expressions, and $\mathcal{I}$ be the set of instructions. In this language, a program p is simply an instruction followed by a return (i.e. $p = i; \mathsf{return}\ e$).

$$\frac{}{x \overset{\rho}{\hookrightarrow} \rho(x)} \qquad \frac{}{c \overset{\rho}{\hookrightarrow} c} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1\ op\ e_2 \overset{\rho}{\hookrightarrow} v_1\ op\ v_2} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1\ cmp\ e_2 \overset{\rho}{\hookrightarrow} v_1\ cmp\ v_2}$$

$$\frac{e \overset{\rho}{\hookrightarrow} v}{[\rho,\, x := e] \Downarrow_{\mathcal{S}} \rho \oplus \{x \leftarrow v\}} \qquad \frac{}{[\rho,\, \mathsf{skip}] \Downarrow_{\mathcal{S}} \rho} \qquad \frac{[\rho,\, i_1] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, i_2] \Downarrow_{\mathcal{S}} \rho''}{[\rho,\, i_1; i_2] \Downarrow_{\mathcal{S}} \rho''}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [\rho,\, i_t] \Downarrow_{\mathcal{S}} \rho'}{[\rho,\, \mathsf{if}(t)\{i_t\}\{i_f\}] \Downarrow_{\mathcal{S}} \rho'} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false} \quad [\rho,\, i_f] \Downarrow_{\mathcal{S}} \rho'}{[\rho,\, \mathsf{if}(t)\{i_t\}\{i_f\}] \Downarrow_{\mathcal{S}} \rho'}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [\rho,\, i] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho''}{[\rho,\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho''} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false}}{[\rho,\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho}$$

Figure 2. SEMANTICS OF THE BASIC LANGUAGE

instruction $i ::=$	$\mathsf{push}\ c$	push value on top of stack
	$\mid\ \mathsf{binop}\ op$	binary operation on stack
	$\mid\ \mathsf{load}\ x$	load value of x on stack
	$\mid\ \mathsf{store}\ x$	store top of stack in variable x
	$\mid\ \mathsf{goto}\ j$	unconditional jump
	$\mid\ \mathsf{if}\ cmp\ j$	conditional jump
	$\mid\ \mathsf{return}$	return the top value of the stack

where $c \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}$.

Figure 3. INSTRUCTION SET FOR THE BASIC BYTECODE

Operational semantics States consist of an instruction and a memory. Thus, the set $\mathrm{State}_{\mathcal{S}} = \mathcal{I} \times \mathcal{L}$ of states is defined as the set of pairs of the form $[\rho,\, i]$, where i is an instruction.

Figure 2 presents the big step semantics of the basic source language. The first relation $\overset{\rho}{\hookrightarrow} \subseteq (\mathcal{E} \times \mathcal{L}) \times \mathbb{Z}$ defines the evaluation under a memory ρ of an expression e into a value. Abusing notation, we use the same syntax for the evaluation of tests. The second relation $\Downarrow_{\mathcal{S}} \subseteq \mathrm{State}_{\mathcal{S}} \times \mathcal{L}$ defines the big-step semantics of an instruction i as a relation between an initial memory and a final memory. There is no rule for the return, as it only appears at the end of the program. We rather define the semantics of programs directly with the clause:

$$\frac{p = i; \mathsf{return}\ e \quad [\rho_0,\, i] \Downarrow_{\mathcal{S}} \rho \quad e \overset{\rho}{\hookrightarrow} v}{p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v}$$

$$\frac{\dot{p}[k] = \mathsf{push}\ c}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k+1,\ \rho,\ c :: os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{binop}\ op \quad v = v_1\ op\ v_2}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\ \rho,\ v :: os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{load}\ x}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k+1,\ \rho,\ \rho(x) :: os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{store}\ x}{\langle k,\ \rho,\ v :: os\rangle \rightsquigarrow \langle k+1,\ \rho \oplus \{x \leftarrow v\},\ os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{if}\ cmp\ j \quad v_1\ cmp\ v_2}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\ \rho,\ os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{if}\ cmp\ j \quad \neg(v_1\ cmp\ v_2)}{\langle k,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow \langle j,\ \rho,\ os\rangle}$$

$$\frac{\dot{p}[k] = \mathsf{goto}\ j}{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle j,\ \rho,\ os\rangle} \qquad \frac{\dot{p}[k] = \mathsf{return}}{\langle k,\ \rho,\ v :: os\rangle \rightsquigarrow \rho, v}$$

Figure 4. SEMANTICS OF THE BASIC BYTECODE

2.2. The Virtual Machine : VM

Programs A bytecode program $\dot{p}$ is an array of instructions (defined in figure 3). We let $\mathcal{P}$ be the set of program points, i.e. $\mathcal{P} = \{0 \ldots n-1\}$ where n is the length of $\dot{p}$. Instructions act on the operand stack (push and load, binop perform an operation with the two top elements, store saves the top element in a variable) or on the control flow (goto for an unconditional jump and if for a conditional jump). Note that, unlike source programs, return instructions may arise anywhere in the code. We nevertheless assume that the program is well-formed, i.e. that the last instruction of the program is a return.

Operational semantics A bytecode state is a triple $\langle k,\ \rho,\ os\rangle$ where k is a program counter, i.e. an element of $\mathcal{P}$, ρ is a memory, and os the operand stack that contains intermediate values needed by the evaluation of source language expressions. We note $\mathrm{State}_{\mathcal{B}}$ the set of bytecode states.

The small-step semantics of a bytecode program is given by the relation $\rightsquigarrow \subseteq \mathrm{State}_{\mathcal{B}} \times (\mathrm{State}_{\mathcal{B}} + \mathcal{L} \times \mathbb{Z})$ which represents one step of execution. Figure 4 defines this relation.

The reflexive and transitive closure $\rightsquigarrow^* \subseteq \mathrm{State}_{\mathcal{B}} \times \mathrm{State}_{\mathcal{B}}$ of $\rightsquigarrow$ is inductively defined by

$$\frac{}{\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k,\ \rho,\ os\rangle} \qquad \frac{\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k',\ \rho',\ os'\rangle \quad \langle k',\ \rho',\ os'\rangle \rightsquigarrow^* \langle k'',\ \rho'',\ os''\rangle}{\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k'',\ \rho'',\ os''\rangle}$$

Finally, the evaluation of a bytecode program $\dot{p} : \rho_0 \Downarrow \rho, v$, from an initial memory ρ_0 to a final memory ρ and a return value v is defined by

$$\frac{\langle 0,\ \rho_0,\ \emptyset\rangle \rightsquigarrow^* \langle k,\ \rho,\ os\rangle \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow \rho, v}{\dot{p} : \rho_0 \Downarrow \rho, v}$$

Remark that the last transition step is necessary done by a return instruction so the memory is unchanged.

Compilation of expressions

$$
\begin{aligned}
[\![x]\!] &= \mathsf{load}\ x \\
[\![c]\!] &= \mathsf{push}\ c \\
[\![e_1\ op\ e_2]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op
\end{aligned}
$$

Compilation of instructions

$$
\begin{aligned}
k : [\![\mathsf{skip}]\!] &= \\
\\
k : [\![x := e]\!] &= [\![e]\!];\ \mathsf{store}\ x \\
\\
k : [\![i_1; i_2]\!] &= k : [\![i_1]\!]; k_2 : [\![i_2]\!] \\
&\text{where } k_2 = k + |[\![i_1]\!]| \\
\\
k : [\![\mathsf{return}\ e]\!] &= [\![e]\!];\ \mathsf{return} \\
\\
k : [\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ l;\ k_2 : [\![i_2]\!] \\
&\text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
&\qquad\ k_2 = k_1 + |[\![i_1]\!]| + 1 \\
&\qquad\ l = k_2 + |[\![i_2]\!]| \\
\\
k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] &= [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\
&\text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
&\qquad\ k_2 = k_1 + |[\![i]\!]| + 1
\end{aligned}
$$

Figure 5. COMPILATION SCHEME

2.3. The compiler

The compiler from the source language to the bytecode language is defined in Figure 5. Compilation of expression $[\![e]\!]$ generates a bytecode sequence which evaluate e and store/push the result on the top of the operand stack. For the compilation of instructions $k : [\![i]\!]$ the compiler argument k indicates the starting position of the resulting bytecode sequence.

Compilation of an assignment $x := e$ is the compilation of the expression e followed by a store x. At the end of the evaluation of $[\![e]\!]$ the value of e is on the top of the operand stack, then a store x instruction stores this value in the variable x and pop the value from the stack.

The compilation of a conditional $k : [\![\mathsf{if}(e_1\ op\ e_2)\{i_1\}\{i_2\}]\!]$ starts by the sequence corresponding to the evaluation of the two expressions e_2 and e_1. After this sequence the operand stack contains on the top the values of e_1 and e_2, the if $cmp\ k_2$ instruction evaluates the comparison and pop the two value from the stack. If the test is true the evaluation continues at label k_1 corresponding to the beginning of the true branch, if the

test is false the if instruction jumps to label k_2 to the beginning of the false branch. At the end of the true branch a goto instruction jumps to the code of the false branch.

The compilation of a loop $k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!]$ evaluates the two expressions e_2 and e_1 and then performs a conditional jump. If the test is false the evaluation jumps to the code corresponding to the body of the loop, if the test is true the evaluation continue by the evaluation of the loop body and then perform a jump to the label corresponding to the beginning of the evaluation of the test.

Finally the compilation of a program $p = (i; \mathsf{return}\ e)$ is defined by:

$$[\![p]\!] \stackrel{\text{def}}{\equiv} 0 : [\![i]\!];\ [\![e]\!];\ \mathsf{return}$$

2.3.1. Correctness

The compiler is correct in the sense that the semantics of a source program is the same as its compiled version. In other words, for all source program p and initial memory ρ_0, executing p with initial memory ρ_0 terminates with final memory ρ and return value v, iff executing $[\![p]\!]$ with initial memory ρ_0 terminates with final memory ρ and return value v

The correctness proof of compiler is done by exhibiting a strong form of simulation between the evaluation of the source program and the evaluation of the bytecode program. In order to carry the proof, we define a new notion of reduction for bytecode states: $s \rightsquigarrow^n s'$, which stands for s evaluates to s' in exactly n steps of reduction $\rightsquigarrow$. Remark that the relation $s \rightsquigarrow^* s'$ can be defined by $\exists n,\ s \rightsquigarrow^n s'$.

Lemma 2.1 *For all bytecode program $\dot{p}$, expression e, memory ρ and operand stack os such that $l = |[\![e]\!]|$ and $\dot{p}[k..k+l-1] = [\![e]\!]$ the following proposition holds:*

$$\begin{array}{l}\forall\ n\ k'\ os',\\ \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow^n \langle k',\ \rho,\ os'\rangle \wedge k' \geq k + l \Rightarrow\\ \quad\quad \exists\ v\ n' > 0, \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n'} \langle k + l,\ \rho,\ v :: os\rangle \rightsquigarrow^{n-n'} \langle k',\ \rho,\ os'\rangle\end{array}$$

The proof is by induction over e. The base cases are trivial. The case of binary expressions is proved using the induction hypothesis.

Lemma 2.2 (Correctness for expressions) *For all bytecode program $\dot{p}$, expression e, value v, memory ρ and operand stack os such that $l = |[\![e]\!]|$ and $\dot{p}[k..k+l-1] = [\![e]\!]$*

$$e \xhookrightarrow{\rho} v \iff \langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k + l,\ \rho,\ v :: os\rangle$$

The proof is by induction over e. The only difficult case is the $\Longleftarrow$ for binary expressions. We have $e = e_1\ op\ e_2$ and $[\![e]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op$ and

$$\langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k + l,\ \rho,\ v :: os\rangle$$

Using the previous lemma there exists v_1 and v_2 such that $v = v_1 + v_2$ and

$$\begin{array}{l}\langle k,\ \rho,\ os\rangle \rightsquigarrow^*\\ \langle k + |[\![e_2]\!]|,\ \rho,\ v_2 :: os\rangle \rightsquigarrow^*\\ \langle k + |[\![e_2]\!]| + |[\![e_1]\!]|,\ \rho,\ v_1 :: v_2 :: os\rangle \rightsquigarrow^*\\ \langle k + l,\ \rho,\ v :: os\rangle\end{array}$$

We conclude using the induction hypothesis.

Lemma 2.3 *For all bytecode program $\dot{p}$, instruction i which is not a* skip*, memory ρ and operand stack os such that $l = |[\![k]\!]|$ and $\dot{p}[k..k+l-1] = k : [\![i]\!]$ the following proposition holds:*

$$\begin{array}{l}\forall\ n\ k'\ \rho'\ os\ os', \\ \quad \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n} \langle k',\ \rho',\ os'\rangle \wedge k' \geq k + l \Rightarrow \\ \qquad \exists\ \rho''\ n' > 0, \langle k,\ \rho,\ os\rangle \rightsquigarrow^{n'} \langle k + l,\ \rho'',\ os\rangle \rightsquigarrow^{n-n'} \langle k',\ \rho',\ os'\rangle\end{array}$$

The proof is done using a general induction principle on n (i.e the induction hypothesis can be applied to any $n' < n$) and by case analysis on i. Remark that this lemma can be proved with $os = \emptyset$ due to the fact that the compiler maintains the invariant that the operand stack is empty at the beginning and at the end of the evaluation of all instructions. This invariant is used in the proof of the next lemma.

Lemma 2.4 (Correctness for instructions) *For all bytecode program $\dot{p}$, instruction i, memories ρ and ρ' such that $l = |[\![i]\!]|$ and $\dot{p}[k..k + l - 1] = k : [\![i]\!]$*

$$[\rho,\ i] \Downarrow_{\mathcal{S}} \rho' \iff \langle k,\ \rho,\ \emptyset\rangle \rightsquigarrow^{*} \langle k + l,\ \rho',\ \emptyset\rangle$$

The direction $\Longrightarrow$ is done by induction on the evaluation of i (i.e one the derivation of $[\rho,\ i] \Downarrow_{\mathcal{S}} \rho'$) and presents no difficulty. To prove the direction $\Longleftarrow$ we prove that :

$$\forall n,\ \langle k,\ \rho,\ \emptyset\rangle \rightsquigarrow^{n} \langle k + l,\ \rho',\ \emptyset\rangle \Longrightarrow [\rho,\ i] \Downarrow_{\mathcal{S}} \rho'$$

The proof is done using a general induction principle on n and by case analysis on i. The cases of loop and conditional use the previous lemma.

Proposition 2.5 (Correctness of the compiler) *For all source program p, initial memory ρ_0, final memory ρ and return value v,*

$$p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v \iff [\![p]\!] : \rho_0 \Downarrow \rho, v$$

This is a direct application of the above lemmas.

3. Information flow

Confidentiality (also found in the literature as privacy or secrecy) policies aim to guarantee that an adversary cannot access information considered as secret. Thus, confidentiality is not an absolute concept but is rather defined relative to the observational capabilities of the adversary. In these notes, we assume that the adversary cannot observe or modify intermediate memories, and cannot distinguish if a program terminates or not. In addition, we consider that information is classified either as public, and thus visible by adversary, or secret. We refer to [**?**] for further details on information flow.

3.1. Security policy

In this section, we specify formally (termination insensitive) *non-interference* [5,10], a baseline information flow policy, which assumes that an adversary can read the public inputs and outputs of a run, and which ensures that adversaries cannot deduce the value of secret inputs from observing the value of public outputs. In its more general form, non-interference is expressed relative to a lattice of security levels. For the purpose of these notes, we consider a lattice with only two security levels, and let $\mathcal{S} = \{L, H\}$ be the set of security levels, where H (high) and L (low) respectively correspond to secret and public information; one provides a lattice structure by adding the subtyping constraint $L \leq H$.

A policy is a function that classifies all variables as low or high.

Definition 3.1 (Policy) *A policy is a mapping* $\Gamma : \mathcal{X} \to \mathcal{S}$.

In order to state the semantic property of non-interference, we begin by defining when two memories are indistinguishable from the point of view of an adversary.

Definition 3.2 (Indistinguishability of memories) *Two memories* $\rho, \rho' : \mathcal{L}$ *are indistinguishable w.r.t. a policy* Γ*, written* $\rho \sim_\Gamma \rho'$ *(or simply* $\rho \sim \rho'$ *when there is no ambiguity), if* $\rho(x) = \rho'(x)$ *for all* $x \in \mathcal{X}$ *such that* $\Gamma(x) = L$.

One can think about the adversary as a program with access to only low parts of the memory and that is put in sequence with the code that manipulates secret information. Its goal is to distinguish between two different executions starting with indistinguishable memories ρ_1 and ρ_2. This is stated in the following definition.

Definition 3.3 (Non-interfering program) *A bytecode program* $\dot{p}$ *is* non-interfering *w.r.t. a policy* Γ*, if for every* $\rho_1, \rho_1', \rho_2, \rho_2', v_1, v_2$ *such that* $\dot{p} : \rho_1 \Downarrow \rho_1', v_1$ *and* $\dot{p} : \rho_2 \Downarrow \rho_2', v_2$ *and* $\rho_1 \sim \rho_2$*, we have* $\rho_1' \sim \rho_2'$ *and* $v_1 = v_2$.

The definition of non-interference applies both to bytecode programs, as stated above, and source programs. Note moreover that by correctness of the compiler, a source program p is non-interfering iff its compilation $[\![p]\!]$ is non-interfering.

3.2. Examples of insecure programs

This section provides examples of insecure programs that must be rejected by a type system. For each example, we provide the source program and its compilation. In all examples, x_L is a low variable and y_H is a high variable.

Our first example shows a direct flow of information, when the result of some secret information is copied directly into a public variable. Consider the program $x_L := y_H; \mathsf{return}\ 0$ and its compilation in Figure 6(a). The program stores in the variable x_L the value held in the variable y_H, and thus leaks information.

Our second example shows an indirect flow of information, when assignments to low variables within branching instructions that test on secret information leads to information leakages. Consider the program $\mathsf{if}(y_H = 0)\{x_L := 0\}\{x_L := 1\}; \mathsf{return}\ 0$ and its compilation in Figure 6(b). 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 depends on the

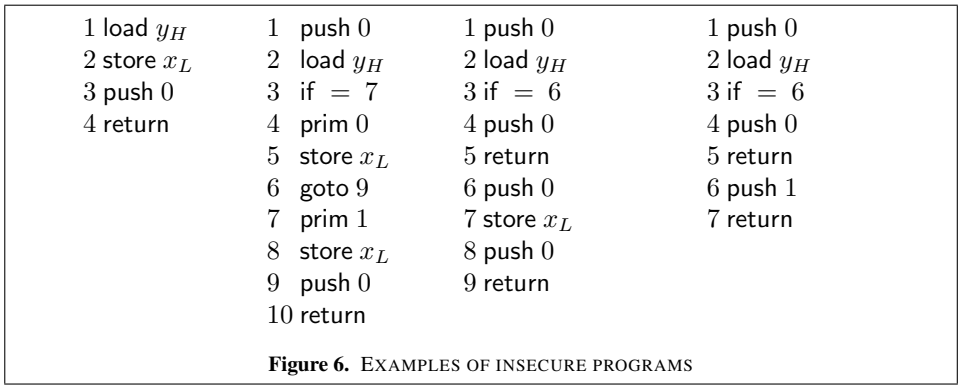

Figure 6. EXAMPLES OF INSECURE PROGRAMS

initial value of y_H. The problem is caused by an assignment to x_L in the scope of an if instruction depending on high variables.

Our third example[1] shows an indirect flow of information, caused by an abrupt termination within branching instructions that test on secret information leads to information leakages. Consider the program $\mathsf{if}(y_H = 0)\{\mathsf{return}\ 0\}\{\mathsf{skip}\}; x_L := 0; \mathsf{return}\ 0$ and its compilation in Figure 6(c); it yields an implicit flow, as the final value of x_L depends on the initial value of y_H. Our fourth example is of a similar nature, but caused by a return whose value depends on a high expression. Consider the program $\mathsf{if}(y_H = 0)\{\mathsf{return}\ 0\}\{\mathsf{return}\ 1\}$ and its compilation in Figure 6(c). Indeed, the final value of x_L is 0 if the initial value of y_H is 0. The problem is caused by a return instruction within the scope of a high if instruction.

3.3. Information flow typing for source code

In this section, we introduce a type system for secure information flow for IMP inspired from [20]. Typing judgments are implicitly parametrized by the security policy Γ of the form:

$$\vdash e : \tau \text{ (expressions)}$$
$$\vdash i : \tau \text{ (statements)}$$

where e is an expression, i is a statement and τ is a security level. The intuitive meaning of $\vdash e : \tau$ is that τ is an upper bound of the security levels of variables that occur in e, whereas the intuitive meaning of $\vdash i : \tau$ is that i is non-interfering and does not assign to variables with security level lower than τ.

Figure 7 and Figure 8 respectively present the typing rules for expressions and instructions. The rule for assignments prevents direct flows, whereas the rule for if statements prevents indirect flows. Note that the typing rule for return is only sound because we do not allow return expressions to appear within statements: indeed, the source code of the program in Figure 6(c), in which a return statement appears in a high branching statement, is insecure.

The type system is sound, in the sense that typable programs are non-interfering.

[1] Both the third and fourth examples are not legal source programs in our syntax. We nevertheless provide the source code for readability.

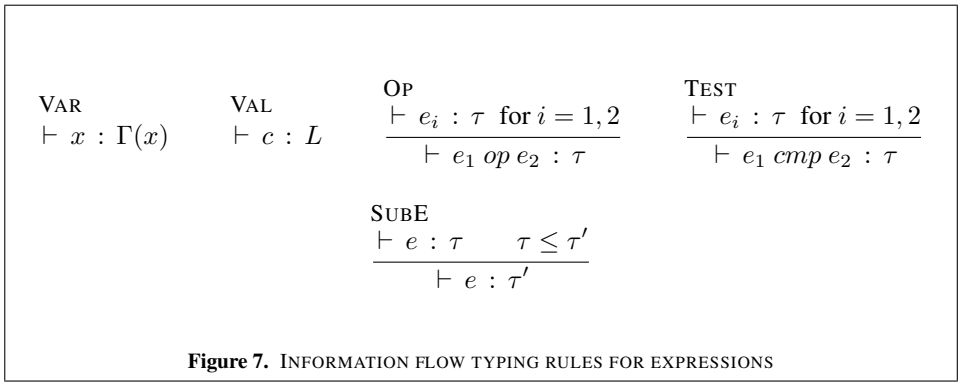

Figure 7. INFORMATION FLOW TYPING RULES FOR EXPRESSIONS

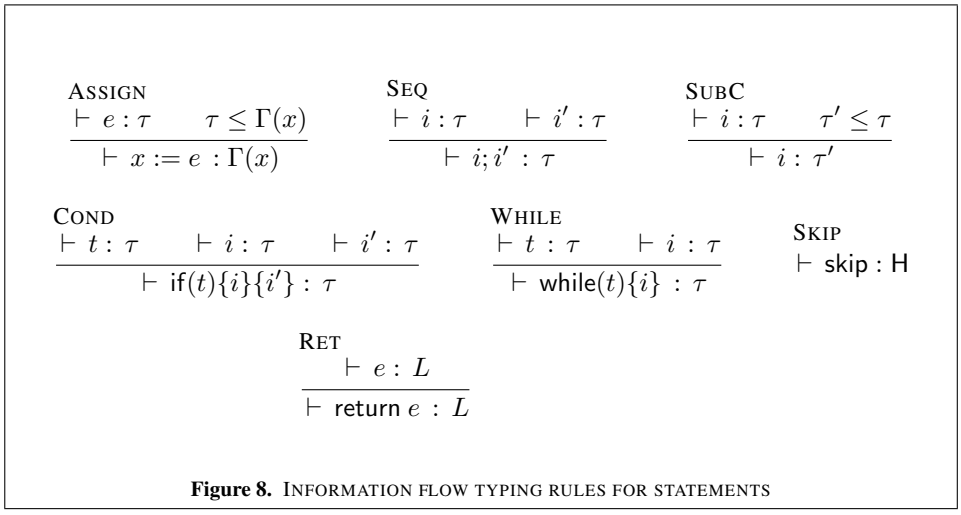

Figure 8. INFORMATION FLOW TYPING RULES FOR STATEMENTS

Proposition 3.4 (Soundness of source type system) *If* $p = i; \mathsf{return}\ e$ *is typable, i.e.* $\vdash i : \tau$ *and* $\mathsf{return}\ e : L$*, then* p *is non-interfering.*

One can prove the above proposition directly, in the style of [20]. An alternative is to derive soundness of the source type system from soundness of the bytecode type system, as we do in the next section.

3.4. Information flow for bytecode

To prevent illicit flows in a non-structured language, one cannot simply enforce local constraints in the typing rules for branching instructions: one must also enforce global constraints that prevent low assignments and updates to occur under high guards (conditional branching that depends on high information). In order to express the global constraints that are necessary to enforce soundness, we rely on some additional information about the program. Concretely, we assume given control dependence regions (cdr) which approximate the scope of branching statements, as well as a security environment, that

attaches to each program point a security level, intuitively the upper bound of all the guards under which the program point executes.

Before explaining the typing rules, we proceed to define the concept of control dependence region, which is closely related to the standard notion used in compilers. The notion of region can be described in terms of a successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$ between program points. Intuitively, j is a successor of i, written $i \mapsto j$, if performing one-step execution from a state whose program point is i may lead to a state whose program point is j. Then, a return point is a program point without successor (corresponding to a return instruction); in the sequel, we write $i \mapsto \mathsf{exit}$ if i is a return point and let $\mathcal{P}_r$ denote the set of return points. Finally, if instructions usually have two successors; when it is the case, the program point of this instruction is referred as a branching point. Formally, the successor relation $\mapsto$ is given by the clauses:

- if $\dot{p}[i] = \mathsf{goto}\ j$, then $i \mapsto j$;
- if $\dot{p}[i] = \mathsf{if}\ cmp\ j$, then $i \mapsto i+1$ and $i \mapsto j$. Since if instructions have two successors, they are thus referred to as branching points;
- if $\dot{p}[i] = \mathsf{return}$, then i has no successors, and we write $i \mapsto \mathsf{exit}$;
- otherwise, $i \mapsto i+1$.

Control dependence regions are characterized by a function that maps a branching program point i to a set of program points $\mathsf{region}(i)$, called the region of i, and by a partial function that maps branching program points to a junction point $\mathsf{jun}(i)$. The intuition behind regions and junction points is that $\mathsf{region}(i)$ includes all program points executing under the guard of i and that $\mathsf{jun}(i)$, if it exists is the sole exit to the region of i; in particular, whenever $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. The properties to be satisfied by control dependence regions, called SOAP properties, are:

Definition 3.5 *A cdr structure* $(\mathsf{region}, \mathsf{jun})$ *satisfies the SOAP (Safe Over APproximation) properties if the following holds:*

SOAP1: *for all program points* i, j, k *such that* $i \mapsto j$ *and* $i \mapsto k$ *and* $j \neq k$*, either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{jun}(i)$*;*

SOAP2: *for all program points* i, j, k*, if* $j \in \mathsf{region}(i)$ *and* $j \mapsto k$*, then either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{jun}(i)$*;*

SOAP3: *for all program points* i, j*, if* $j \in \mathsf{region}(i)$ *and* $j \mapsto \mathsf{exit}$ *then* $\mathsf{jun}(i)$ *is undefined.*

Given a cdr structure $(\mathsf{region}, \mathsf{jun})$, it is straightforward to verify whether or not it satisfies the SOAP properties.

Definition 3.6 *A security environment is a mapping* $se : \mathcal{P} \to \mathcal{S}$.

The bytecode type system is implicitly parametrized by a policy Γ, a cdr structure $(\mathsf{region}, \mathsf{jun})$, and a security environment se. Typing judgments are of the form

$$i \vdash st \Rightarrow st'$$

where i is a program point, and st and st' are stacks of security levels. Intuitively, st and st' keep track of security levels of information on the operand stack during all possible executions.

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

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

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) = L}{i \vdash k :: st \Rightarrow \epsilon}$$

$$\frac{P[i] = \mathsf{if}\ cmp\ j \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \mathrm{lift}_k(st)}$$

Figure 9. TRANSFER RULES FOR VM INSTRUCTIONS

Figure 9 presents a set of typing rules that guarantee non-interference for bytecode programs, where $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$, lift_k is the point-wise extension to stack types of $\lambda l.\ k \sqcup l$. The transfer rule for if requires that the security environment of program points in a high region is high. In conjunction with the transfer rule for load, the transfer rule for if prevents implicit flows and rejects the program of Figure 6(b). Likewise, in conjunction with the transfer rule for push, which requires that the value pushed on top of the operand stack has a security level greater than the security environment at the current program point, and the typing rule for return which requires that $se(i) = L$ and thus avoids return instructions under the guard of high expressions, the transfer rule for return prevents implicit flows and rejects the program of Figure 6(c). Besides, the transfer rule for return requires that the value on top of the operand stack has a low security level, since it will be observed by the adversary. It thus rightfully rejects the program of Figure 6(d).

In addition, the operand stack requires the stack type on the right hand side of $\Rightarrow$ to be lifted by the level of the guard, i.e. the top of the input stack type. It is necessary to perform this lifting operation to avoid illicit flows through operand stack. The following example, which uses a new instruction swap that swaps the top two elements of the operand stack, illustrates why we need to lift the operand stack. This is a contrived example because it does not correspond to any simple source code, but it is nevertheless accepted by a standard bytecode verifier.

1 push 1
2 push 0
3 push 0
4 load y_H
5 if 7
6 swap
7 store x_L
8 push 0
9 return

In this example, the final value of variable x_L depends on the initial value of y_H and so the program is interfering. It is rightfully rejected by our type system, thanks to the lift of the operand stack at program point 5.

One may argue that lifting the entire stack is too restrictive, as it leads the typing system to reject safe programs; indeed, it should be possible, at the cost of added complexity, to refine the type system to avoid lifting the entire stack. Nevertheless, one may equally argue that lifting the stack is unnecessary, because as noted in Section 2 the stack at branching points only has one element in all compiled programs, in which case a more restrictive rule of the form below is sufficient:

$$\frac{P[i] = \mathsf{if}\ cmp\ j \qquad \forall j' \in \mathsf{region}(i).k \leq se(j')}{i \vdash k :: \epsilon \Rightarrow \epsilon}$$

Furthermore, there are known techniques to force that the stack only has one element at branching points.

Typability Typing rules are used to establish a notion of typability. Following Freund and Mitchell [9], typability stipulates the existence of a function, that maps program points to stack types, such that each transition is well-typed.

Definition 3.7 (Typable program) *A program $\dot{p}$ is typable w.r.t. a memory security policy Γ, and cdr structure $(\mathsf{region}, \mathsf{jun})$, and a security environment $se : \mathcal{P} \to \mathcal{S}$ iff there exists a type $S : \mathcal{P} \to \mathcal{S}^\star$ such that:*

- *$S_0 = \varepsilon$ (the operand stack is empty at the initial program point 0);*
- *for all $i \in \mathcal{P}$ and $j \in \mathcal{P} \cup \{\mathsf{exit}\}$, $i \mapsto j$ implies that there exists $st \in \mathcal{S}^\star$ such that $i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$.*

where we write S_i instead of $S(i)$ and $\sqsubseteq$ denotes the point-wise partial order on type stack with respect to the partial order taken on security levels.

The type system is sound, in the sense that typable programs are non-interfering.

Proposition 3.8 *Let $\dot{p}$ be a bytecode program and $(\mathsf{region}, \mathsf{jun})$ a cdr structure that satisfies the SOAP properties. Suppose $\dot{p}$ is typable with respect to* region *and to a memory security policy Γ. Then $\dot{p}$ is non-interfering w.r.t. Γ.*

The proof is based on the SOAP properties, and on two unwinding lemmas showing that execution of typable programs does not reveal secret information. In order to state the unwinding lemmas, one must define an indexed indistinguishability relation between stacks and states.

Definition 3.9 (Indistinguishability of states)

- *A $\mathcal{S}$-stack S is high, written $\mathsf{high}(S)$, if all levels in S are high.*
- *Indistinguishability $os \sim_{S,S'} os'$ between stacks os and os (relative to $\mathcal{S}$-stacks S and S') is defined inductively by the clauses:*

$$\frac{\mathsf{high}(S) \qquad \mathsf{high}(S') \qquad \#os = \#S \qquad \#os' = \#S'}{os \sim_{S,S'} os'}$$

$$\frac{os \sim_{S,S'} os'}{v :: os \sim_{L::S,L::S'} v :: os'} \qquad \frac{os \sim_{S,S'} os'}{v :: os \sim_{H::S,H::S'} v' :: os'}$$

where $\#$ denotes the length of a stack.

- *Indistinguishability between states $\langle i, \rho, os\rangle \sim_{S,S'} \langle i', \rho', os'\rangle$ (relative to stack of security levels S and S') holds iff $os \sim_{S,S'} os'$ and $\rho \sim \rho'$.*

We must also introduce some terminology and notation: we say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j)$ is high for all $j \in \mathsf{region}(i)$. Besides, we let $\mathsf{pc}(s)$ denote the program counter of a state s. Then, the unwinding lemmas can be stated as follows:

- *locally respects:* if $s \sim_{S,T} t$, and $\mathsf{pc}(s) = \mathsf{pc}(t) = i$, and $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$, and $i \vdash T \Rightarrow T'$, then $s' \sim_{S',T'} t'$.
- *step consistent:* if $s \sim_{S,T} t$, and $\mathsf{pc}(s) = i$, and $s \rightsquigarrow s'$ and $i \vdash S \Rightarrow S'$, and $se(i)$ is high, and S is high, then $s' \sim_{S',T} t$.

In order to repeatedly apply the unwinding lemmas, we need additional results about preservation of high contexts.

- *high branching:* if $s \sim_{S,T} t$ with $\mathsf{pc}(s) = \mathsf{pc}(t) = i$ and $\mathsf{pc}(s') \neq \mathsf{pc}(t')$, if $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$ and $i \vdash T \Rightarrow T'$, then S' and T' are high and se is high in region $\mathsf{region}(i)$.
- *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and the security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then S' is high.

The combination of the unwinding lemmas, the high context lemmas, the monotonicity lemmas and the SOAP properties enable to prove that typable programs are noninterfering. The proof proceeds by induction on the length of derivations: assume that we have two executions of a typable program $\dot{p}$, and that s_n and s'_m are final states:

$$s_0 \rightsquigarrow \cdots \rightsquigarrow s_n$$
$$s'_0 \rightsquigarrow \cdots \rightsquigarrow s'_m$$

such that $\mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$ and $s_0 \sim_{S_{\mathsf{pc}(s_0)},S_{\mathsf{pc}(s'_0)}} s'_0$. We want to establish that either the states s_n and s'_m are indistinguishable, i.e. $s_n \sim_{S_{\mathsf{pc}(s_n)},S_{\mathsf{pc}(s'_m)}} s'_m$, or that both stack types $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high. By induction hypothesis, we know that the property holds for all strictly shorter execution paths.

Define $i_0 = \mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$. By the *locally respects* lemma and typability hypothesis, $s_1 \sim_{st,st'} s'_1$ for some stack types st and st' such that $i_0 \vdash S_{i_0} \Rightarrow st$, $st \sqsubseteq S_{\mathsf{pc}(s_1)}$, $i_0 \vdash S_{i_0} \Rightarrow st'$, $st' \sqsubseteq S_{\mathsf{pc}(s'_1)}$.

- If $\mathsf{pc}(s_1) = \mathsf{pc}(s'_1)$ we can apply monotony of indistinguishability (w.r.t. indexes) to establish that $s_1 \sim_{S_{\mathsf{pc}(s_1)},S_{\mathsf{pc}(s'_1)}} s'_1$ and conclude by induction hypothesis.

- If $\mathsf{pc}(s_1) \neq \mathsf{pc}(s'_1)$ we know by the *high branching* lemma that se is high in region $\mathsf{region}(i_0)$ and st and st' are high. Hence both $S_{\mathsf{pc}(s_1)}$ and $S_{\mathsf{pc}(s'_1)}$ are high.
  Using the SOAP properties, one can prove that either $\mathsf{jun}(i_0)$ is undefined and both $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high, or that $\mathsf{jun}(i_0)$ is defined and there exists k, k', $1 \leq k \leq n$ and $1 \leq k' \leq m$ such that $k = k' = \mathsf{jun}(i_0)$ and $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{i_0}} s'_0$ $s_0 \sim_{S_{i_0}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$. Since $s_0 \sim_{S_{i_0}, S_{i_0}} s'_0$ we have by transitivity and symmetry of $\sim$, $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$ with $\mathsf{pc}(s_k) = \mathsf{pc}(s'_{k'})$ and we can conclude by induction hypothesis.

3.5. Preservation of Typability

In this section, we focus on preservation of typability by compilation. Since the bytecode type system uses both a cdr structure $(\mathsf{region}, \mathsf{jun})$ and a security environment se, we must extend the compiler of Section 2.3 so that it generates for each program p a cdr structure $(\mathsf{region}, \mathsf{jun})$ and the security environment se such that $[\![p]\!]$ is typable w.r.t. $(\mathsf{region}, \mathsf{jun})$ and se. The cdr structure of the compiled programs can be defined easily. For example, the region of if statement is given by the clause:

$$\begin{aligned} k : [\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ l;\ k_2 : [\![i_2]\!] \\ \text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\ k_2 = k_1 + |[\![i_1]\!]| + 1 \\ l = k_2 + |[\![i_2]\!]| \\ \mathsf{region}(k_1 - 1) = [k_1, l - 1] \\ \mathsf{jun}(k_1 - 1) = l \\ \mathsf{blev}(k_1 - 1) = \bigcup\{\tau \mid\vdash e_1\ cmp\ e_2 : \tau\} \end{aligned}$$

Note that in addition to the region, we define the branching level $\mathsf{blev}(k_1 - 1)$ of $k_1 - 1$ as the minimal level of its associated test, i.e. $\mathsf{blev}(k_1 - 1)$ low if all variables in e_1 and e_2 are low, and high otherwise.

Likewise, the region of while statement is given by the clause:

$$\begin{aligned} k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] = [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\ \text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\ k_2 = k_1 + |[\![i]\!]| + 1 \\ \mathsf{region}(k_1 - 1) = [k, l - 1] \\ \mathsf{jun}(k_1 - 1) = l \\ \mathsf{blev}(k_1 - 1) = \bigcup\{\tau \mid\vdash e_1\ cmp\ e_2 : \tau\} \end{aligned}$$

The security environment is derived from the cdr structure and the branching level of program points. Formally, we define

$$se(i) = \bigcup\{\mathsf{blev}(j) \mid i \in \mathsf{region}(j)\}$$

with the convention that $\bigcup \emptyset = L$.

Theorem 3.10 (Typability Preservation) *Let p be a typable source program. Then $[\![p]\!]$ is a typable bytecode program w.r.t. the generated cdr structure* $(\mathsf{region}, \mathsf{jun})$ *and the*

generated security environment se*. In addition, the cdr structure* $(\mathsf{region}, \mathsf{jun})$ *satisfies the SOAP properties.*

Using the fact that the compiler preserves the semantics of program, the soundness of the information flow type system for bytecode and preservation of typability, we can derive soundness of the information flow type system for the source language.

Corollary 3.11 (Soundness of source type system) *Let* p *be a typable* IMP *program w.r.t. to a memory security policy* Γ*. Then* p *is non-interfering w.r.t.* Γ*.*

By preservation of typing, $[\![p]\!]$ is typable, and thus non-interfering by soundness of the bytecode type system. By correctness of the compiler, the program p is non-interfering iff its compilation $[\![p]\!]$ is non-interfering, and therefore p is non-interfering.

3.6. Optimizations

Simple optimizations such as constant folding, dead code elimination, and rewriting conditionals whose conditions always evaluate to the same constant can be modeled as source-to-source transformations and can be shown to preserve information-flow typing. Figure 10 provides examples of transformations that preserve typing.[2] Most rules are of the form

$$\frac{P[i] = ins \quad constraints}{P[i] = ins'} \qquad \frac{P[i, i+n] = i\vec{n}s \quad constraints}{P[i, i+n] = i\vec{n}s'}$$

where ins is the original instruction and ins' is the optimized instruction. In some cases however, the rules are of the form

$$\frac{P[i, n+m] = i\vec{n}s \quad constraints}{P[i, n+m'] = i\vec{n}s'}$$

with $m \neq m'$. Therefore such rules do not preserve the number of instructions, and the transformations must recompute the targets of jumps, which is omitted here.

In the rules, we use $\mathcal{F}$ to denote a stack-preserving sequence of instructions, i.e. a sequence of instructions such that the stack is the same at the beginning and the end of $\mathcal{F}$ execution, which we denote as $\mathcal{F} \in \mathsf{StackPres}$ in the rules. We also assume that there are no jumps from an instruction in $\mathcal{F}$ outside $\mathcal{F}$, so that all executions must flow through the immediate successor of $\mathcal{F}$, and that there are no jumps from an instruction outside $\mathcal{F}$ inside $\mathcal{F}$, so that all executions enter $\mathcal{F}$ through its immediate predecessor. In other words, we assume that $ins :: \mathcal{F} :: ins'$ is a program fragment, where ins and ins' are the instructions preceding and following $\mathcal{F}$.

The last rule uses $\mathcal{VAL}(x, i)$ to denote the safe approximation of the value of x at program point i; this approximation can be statically computed through, e.g., symbolic analysis. The optimizations use two new instructions nop and dup, the first one simply jump to the next program point, the second duplicates the top value of the stack and continues the execution to the next program point.

[2] Thanks to Salvador Cavadini for contributing to these examples.

$$\frac{P[i, i+n+2] = i :: \mathcal{F} :: \mathsf{pop} \qquad i \in \{\mathsf{load}\ x, \mathsf{push}\ n\}}{P[i, i+n] = \mathcal{F}}$$

$$\frac{P[i, i+n] = \mathsf{binop}\ op :: \mathcal{F} :: \mathsf{pop}}{P[i, i+n] = \mathsf{pop} :: \mathcal{F} :: \mathsf{pop}}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad x \text{ is dead at } P[i]}{P[i] = \mathsf{pop}}$$

$$\frac{P[i, i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F}}{P[i, i+n] = \mathsf{load}\ x :: \mathcal{F} :: \mathsf{dup}}$$

$$\frac{P[i, i+n] = \mathsf{store}\ x :: \mathcal{F} :: \mathsf{load}\ x \qquad \mathsf{store}\ x \notin \mathcal{F}}{P[i, i+n] = \mathsf{dup} :: \mathsf{store}\ x :: \mathcal{F}}$$

$$\frac{P[i, i+2+n] = \mathsf{store}\ x :: \mathsf{load}\ x :: \mathcal{F} :: \mathsf{store}\ x \qquad \mathsf{store}\ x, \mathsf{load}\ x \notin \mathcal{F}}{P[i, i+n] = \mathcal{F} :: \mathsf{store}\ x}$$

$$\frac{P[i, i+2] = \mathsf{push}\ c_1 :: \mathsf{push}\ c_2 :: \mathsf{binop}\ op}{P[i] = \mathsf{push}\ (c_1\ op\ c_2)}$$

$$\frac{P[i] = \mathsf{load}\ x \qquad \mathcal{VAL}(x, i) = n}{P[i] = \mathsf{push}\ n}$$

In all rules, we assume that $\mathcal{F}$ is stack-preserving.

Figure 10. OPTIMIZING TRANSFORMATION RULES

As noted in [**?**], more aggressive optimizations may break type preservation, even though they are semantics preserving, and therefore security preserving. For example, applying common subexpression elimination to the program

$$x_H := n_1 * n_2; y_L := n_1 * n_2$$

where n_1 and n_2 are constant values, will result in the program

$$x_H := n_1 * n_2; y_L := x_H$$

Assuming that variable x_H is a high variable and y_L is a low variable, the original program is typable, but the optimized program is not, since the typing rule for assignment will detect an explicit flow $y_L := x_H$. For this example, one can recover typability by creating a low auxiliary variable z_L in which to store the result of the computation $n_1 * n_2$, and assign z_L to x_H and y_L, i.e.

$$z_L := n_1 * n_2; x_H := z_L; y_L := z_L$$

$$
\begin{array}{lll}
\text{source logical expressions} & \bar{e} ::= \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \\
\text{source logical tests} & \bar{t} ::= \bar{e}\ cmp\ \bar{e} \\
\text{source propositions} & \phi ::= \bar{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \exists x.\ \phi \mid \forall x.\ \phi
\end{array}
$$

Figure 11. SPECIFICATION LANGUAGE FOR SOURCE PROGRAMS

$$
\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{skip}, \psi) = \psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(x := e, \psi) = \psi\{x \leftarrow e\}, \emptyset}
$$

$$
\frac{\mathsf{wp}_{\mathcal{S}}(i_2, \psi) = \phi_2, \theta_2 \quad \mathsf{wp}_{\mathcal{S}}(i_1, \phi_2) = \phi_1, \theta_1}{\mathsf{wp}_{\mathcal{S}}(i_1; i_2, \psi) = \phi_1, \theta_1 \cup \theta_2}
$$

$$
\frac{\mathsf{wp}_{\mathcal{S}}(i_t, \psi) = \phi_t, \theta_t \quad \mathsf{wp}_{\mathcal{S}}(i_f, \psi) = \phi_f, \theta_f}{\mathsf{wp}_{\mathcal{S}}(\mathsf{if}(t)\{i_t\}\{i_f\}, \psi) = (t \Rightarrow \phi_t) \wedge (\neg t \Rightarrow \phi_t), \theta_t \cup \theta_f}
$$

$$
\frac{\mathsf{wp}_{\mathcal{S}}(i, I) = \phi, \theta}{\mathsf{wp}_{\mathcal{S}}(\mathsf{while}_I(t)\{i\}, \psi) = I, \{I \Rightarrow (t \Rightarrow \phi) \wedge (\neg t \Rightarrow \psi)\} \cup \theta}
$$

Figure 12. WEAKEST PRE-CONDITION FOR SOURCE PROGRAMS

The above examples show that a more systematic study of the impact of program optimizations on information flow typing is required.

4. Verification conditions

Program logics are expressive frameworks that enable reasoning about complex properties as well as program correctness. Early program verification techniques include Hoare logics, and weakest pre-condition calculi, which are concerned with proving program correctness in terms of triples, i.e. statements of the form $\{P\}c\{Q\}$, where P and Q are respectively predicates about the initial and final states of the program c. The intended meaning of a statement $\{P\}c\{Q\}$ is that any terminating run of the program c starting with a state s satisfying the pre-condition P will conclude in a state s' that satisfies the post-condition Q. In these notes, we focus on a related mechanism, called verification condition generation, which differs from the former by operating on annotated programs, and is widely used in program verification environments.

4.1. Source language

The verification condition generator VCgen operates on annotated source programs, i.e. source programs that carry a pre-condition, a post-condition, and an invariant for each

stack expressions	$\bar{os} ::= \mathsf{os} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os}$
bytecode logical expressions	$\bar{e} ::= \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \bar{os}[k]$
bytecode logical tests	$\bar{t} ::= \bar{e}\ cmp\ \bar{e}$
bytecode propositions	$\phi ::= \bar{t} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \exists x.\ \phi \mid \forall x.\ \phi$

where os is a special variable representing the current stack operand stack.

Figure 13. SPECIFICATION LANGUAGE FOR BYTECODE PROGRAMS

loop.

Definition 4.1 (Annotated source program)

- *The set of propositions is defined in Figure* **??***, where $\bar{x}$ is a special variable representing the initial value of the variable x, and* res *is a special value representing the final value of the evaluation of the program.*
- *A pre-condition is a proposition that only refers to the initial values of variables. An invariant is a proposition that refers to the initial and current values of variables (not to the final result). A post-condition is a proposition.*
- *An annotated program is a triple (p, Φ, Ψ), where Φ is a pre-condition, Ψ is a post-condition, and p is a program in which all* while *loops are annotated (we note $\mathsf{while}_I(t)\{s\}$ for a loop annotated with invariant I).*

The VCgen computes a set of verification conditions (VC). Their validity ensure that the program meets its contract, i.e. that every terminating run of a program starting from a state that satisfies the program pre-condition will terminate in a state that satisfies the program post-condition, and that loop invariants hold at the entry and exit of each loops.

Definition 4.2 (Verification conditions for source programs)

- *The weakest pre-condition calculus $\mathsf{wp}_{\mathcal{S}}(i, \psi)$ relative to a instruction i and a post-condition ψ is defined by the rules of Figure* **??***.*
- *The verification conditions of an annotated program (p, Φ, Ψ) with $p = i; \mathsf{return}\ e$ is defined as*

$$\mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi) = \{\Phi \Rightarrow \phi\{\vec{x} \leftarrow \vec{\bar{x}}\}\} \cup \theta$$

where $\mathsf{wp}_{\mathcal{S}}(i, \Psi\{\mathsf{res} \leftarrow e\}) = \phi, \theta$.

4.2. Target language

As for the source language, the verification condition generator operates on annotated bytecode programs, i.e. bytecode that carry a pre-condition, a post-condition and loop invariants. In an implementation, it would be reasonable to store invariants in a separate annotation table, the latter being a partial function form program points to propositions. Here we find it more convenient to store the annotations directly in the instructions.

Definition 4.3 (Annotated bytecode program)

- *The set of bytecode propositions is defined in Figure* **??**.
- *An annotation is a proposition that does not refer to the operand stack. A pre-condition is an annotation that only refers to the initial value of variables. An invariant is an annotation that does not refer to the result of the program. A post-condition is an annotation.*
- *An annotated bytecode instruction is either an bytecode instruction or a bytecode proposition and a bytecode instruction:*

$$\bar{i} ::= i \mid \phi : i$$

- *An annotated program is a triple* $(\dot{p}, \dot{\Phi}, \dot{\Psi})$*, where* $\dot{\Phi}$ *is a pre-condition,* $\dot{\Psi}$ *is a post-condition, and* $\dot{p}$ *is a bytecode program in which some instructions are annotated.*

At the level of the bytecode language, the predicate transformer wp is a partial function that computes, from a partially annotated program, a fully annotated program in which all labels of the program have an explicit pre-condition attached to them. However, wp is only defined on programs that are sufficiently annotated, i.e. through which all loops must pass through an annotated instruction. The notion of sufficiently annotated is characterized by an inductive and decidable definition and does not impose any specific structure on programs.

Definition 4.4 (Well-annotated program) *A annotated program* $\dot{p}$ *is well-annotated if every program point satisfies the inductive predicate* $\mathsf{reachAnnot}_{\dot{p}}$ *defined by the clauses:*

$$\frac{\dot{p}[k] = \phi : i}{k \in \mathsf{reachAnnot}_{\dot{p}}} \qquad \frac{\dot{p}[k] = \mathsf{return}}{k \in \mathsf{reachAnnot}_{\dot{p}}} \qquad \frac{\forall k'.\ k \mapsto k' \Rightarrow k' \in \mathsf{reachAnnot}_{\dot{p}}}{k \in \mathsf{reachAnnot}_{\dot{p}}}$$

Given a well-annotated program, one can generate an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the pre-condition that an initial state before the execution of the corresponding label should satisfy for the function to terminate only in a state satisfying its post-condition.

Definition 4.5 (Verification conditions for bytecode programs) *Let* $(\dot{p}, \dot{\Phi}, \dot{\Psi})$ *be a well-annotated program.*

- *The weakest pre-condition* $\mathsf{wp}_{\mathcal{L}}(k)$ *of a program point* k *and the weakest pre-condition* $\mathsf{wp}_i(k)$ *of its corresponding instruction are defined in Figure* **??**.
- *The verification conditions* $\mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})$ *is defined by the clauses:*

$$\frac{}{\dot{\Phi} \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)\{\vec{x} \leftarrow \vec{\tilde{x}}\}) \in \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})} \qquad \frac{\dot{p}[k] = \phi : i}{\phi \Rightarrow \mathsf{wp}_i(k)) \in \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})}$$

$$
\begin{array}{lcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow c :: \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{push}\ c \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow (\mathsf{os}[0]\ op\ \mathsf{os}[1]) ::\uparrow^2 \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{binop}\ op \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow x :: \mathsf{os}\} & \text{if } \dot{p}[k] = \mathsf{load}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os}, x \leftarrow \uparrow \mathsf{os}, \mathsf{os}[0]\} & \text{if } \dot{p}[k] = \mathsf{store}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } \dot{p}[k] = \mathsf{goto}\ l \\
\mathsf{wp}_i(k) & = & (\mathsf{os}[0]\ cmp\ \mathsf{os}[1] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathsf{os} \leftarrow \uparrow^2 \mathsf{os}\}) & \text{if } \dot{p}[k] = \mathsf{if}\ cmp\ l \\
 & & \wedge\ (\neg(\mathsf{os}[0]\ cmp\ \mathsf{os}[1]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)\{\mathsf{os} \leftarrow \uparrow^2 \mathsf{os}\}) & \\
\mathsf{wp}_i(k) & = & \dot{\Psi}\{\mathsf{res} \leftarrow \mathsf{os}[0]\} & \text{if } \dot{p}[k] = \mathsf{return}
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } \dot{p}[k] = \phi : i \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Figure 14. WEAKEST PRE-CONDITION FOR BYTECODE PROGRAMS

$$
\frac{}{\bar{\rho}, os, \rho \vdash \mathsf{os} \mapsto os} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{e} \mapsto e \quad \bar{\rho}, os, \rho \vdash \bar{os} \mapsto os'}{\bar{\rho}, os, \rho \vdash \bar{e} :: \bar{os} \mapsto v :: os'}
$$

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto v_1 :: \ldots :: v_k :: os'}{\bar{\rho}, os, \rho \vdash \uparrow^k \bar{os} \mapsto os'} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto v_0 :: \ldots :: v_k :: os'}{\bar{\rho}, os, \rho \vdash \bar{os}[k] \mapsto v_k}
$$

$$
\frac{}{\bar{\rho}, os, \rho \vdash \bar{x} \mapsto \bar{\rho}(x)} \qquad \frac{}{\bar{\rho}, os, \rho \vdash x \mapsto \rho(x)} \qquad \frac{}{\bar{\rho}, os, \rho \vdash c \mapsto c}
$$

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1\ op\ \bar{e}_2 \mapsto v_1\ op\ v_2}
$$

Figure 15. INTERPRETATION OF BYTECODE EXPRESSIONS

4.3. Soundness

Bytecode (resp. source) propositions can be interpreted as predicates on bytecode (resp. source) states. In the case of bytecode, the interpretation builds upon a partially defined interpretation of expressions (partiality comes from the fact that some expressions refer to the operand stack and might not be well defined w.r.t. particular states).

Definition 4.6 (Correct program)

- *The evaluation of logical bytecode expressions $\bar{e}$ in an initial memory $\bar{\rho}$, a current operand stack os and a current memory ρ to a value v is defined by the rules of Figure* **??**. *This evaluation is naturally extended to bytecode propositions $\bar{\rho}, os, \rho \vdash P \mapsto \phi_v$, where ϕ_v is a boolean formula, with the following rule for tests:*

$$\frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1 \; cmp \; \bar{e}_2 \mapsto v_1 \; cmp \; v_2}$$

- *An initial memory $\bar{\rho}$, a current operand stack os and a current memory ρ validate a logical bytecode proposition ϕ,$\bar{\rho}, os, \rho \vdash \phi$, if $\bar{\rho}, os, \rho \vdash \phi \mapsto \phi_v$ and ϕ_v is a valid boolean formula.*
- *A well-annotated bytecode program $(\dot{p}, \dot{\Phi}, \dot{\Psi})$ is correct, written $\vdash \mathsf{VCgen}_{\mathcal{B}}(\dot{p}, \dot{\Phi}, \dot{\Psi})$, if all the verification conditions are valid.*

Soundness establishes that the VCgen is a correct backwards abstraction of one step execution.

Lemma 4.7 (One step soundness of VCgen) *For all correct programs $(\dot{p}, \dot{\Phi}, \dot{\Psi})$:*

$$\left.\begin{array}{l} \langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k',\ \rho',\ os'\rangle \\ \bar{\rho}, os, \rho \vdash \mathsf{wp}_i(k) \end{array}\right\} \Rightarrow \bar{\rho}, os', \rho' \vdash \mathsf{wp}_{\mathcal{L}}(k')$$

Furthermore, if the evaluation terminates $\langle k,\ \rho,\ os\rangle \rightsquigarrow \rho, v$ (i.e the instruction at position k is a return*) then $\bar{\rho}, \emptyset, \rho' \vdash \dot{\Psi}\{\mathsf{res} \leftarrow v\}$*

Soundness of the VCgen w.r.t. pre-condition and post-condition follows.

Corollary 4.8 (Soundness of VCgen) *For all correct programs $(\dot{p}, \dot{\Phi}, \dot{\Psi})$, initial memory $\bar{\rho}$, final memory ρ and final value v, if $\dot{p} : \bar{\rho} \Downarrow \rho, v$ and $\bar{\rho}, \emptyset, \emptyset \vdash \dot{\Phi}$ then $\bar{\rho}, \emptyset, \rho \vdash \dot{\Psi}\{\mathsf{res} \leftarrow v\}$*

The proof proceeds as follows. First, we prove by induction on n that

$$\left.\begin{array}{l} \langle k,\ \rho,\ os\rangle \rightsquigarrow^n \langle k',\ \rho',\ os'\rangle \\ \bar{\rho}, os, \rho \vdash \mathsf{wp}_i(k) \end{array}\right\} \Rightarrow \bar{\rho}, os', \rho' \vdash \mathsf{wp}_i(k')$$

If $n = 0$, it is trivial. If $n = 1+m$, we have $\langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k_1,\ \rho_1,\ os_1\rangle \rightsquigarrow^n \langle k',\ \rho',\ os'\rangle$. It is sufficient to prove that $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_i(k_1)$, since then we can conclude the proof using the induction hypothesis. Using the previous lemma, we get $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_{\mathcal{L}}(k_1)$.

We now conclude with a case analysis:

- if the program point k_1 is not annotated then $\mathsf{wp}_{\mathcal{L}}(k_1) = \mathsf{wp}_i(k_1)$, and we are done;
- if the program point k_1 is annotated, say $\dot{p}[k_1] = \phi : i$, then $\mathsf{wp}_{\mathcal{L}}(k_1) = \phi$. Since the program is correct the proposition $\phi \Rightarrow \mathsf{wp}_i(k_1)$ is valid and so $\bar{\rho}, os_1, \rho_1 \vdash \mathsf{wp}_i(k_1)$.

Second, since $\dot{p} : \bar{\rho} \Downarrow \rho, v$ there exists n such that

$$\langle 0,\ \rho_0,\ \emptyset\rangle \rightsquigarrow^n \langle k,\ \rho,\ os\rangle \rightsquigarrow \rho, v$$

By step one above, we have $\rho_0, os, \rho \vdash \mathsf{wp}_i(k)$. Furthermore, $\dot{p}[k] = \mathsf{return}$ so $\mathsf{wp}_i(k) = \dot{\Psi}\{res \leftarrow \mathsf{os}[0]\}$. This concludes the proof.

4.4. Preservation of proof obligations

We now extend our compiler so that it also inserts annotations in bytecode programs, and show that it transforms programs into well-annotated programs, and that furthermore it transforms correct source programs into correct bytecode programs. In fact, we show a stronger property, namely that the proof obligations at source and bytecode level coincide.

The compiler of Section 2.3 is modified to insert invariants in bytecode:

$$\begin{gathered} k : [\![\mathsf{while}_I(e_1\ cmp\ e_2)\{i\}]\!] = I : [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\ \text{where } k_1 = k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\ k_2 = k_1 + |[\![i]\!]| + 1 \end{gathered}$$

As expected, the compiler produces well-annotated programs.

Lemma 4.9 (Well-annotated programs) *For all annotated source program(p, Φ, Ψ), the bytecode program $[\![p]\!]$ is well-annotated.*

In addition, the compiler "commutes" with verification condition generation. Furthermore, the commutation property is of a very strong form, since it claims that proof obligations are syntactically equal.

Proposition 4.10 (Preservation of proof obligations – PPO) *For all annotated source program (p, Φ, Ψ):*

$$\mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi) = \mathsf{VCgen}_{\mathcal{B}}([\![p]\!], \Phi, \Psi)$$

Thus, correctness proofs of source programs can be used directly as proof of bytecode programs without transformation. In particular, the code producer can directly prove the source program and send the proofs and the compiled program to the code consumer without transforming the proofs.

Using the fact that the compiler preserves the semantics of program, the soundness of the verification condition generator for bytecode and PPO, we can derive soundness of the source verification condition generator. (The notion of correct source program is defined in the same way as that of bytecode program).

Corollary 4.11 (Soundness of $\mathsf{VCgen}_{\mathcal{S}}$) *If $\vdash \mathsf{VCgen}_{\mathcal{S}}(p, \Phi, \Psi)$ then for all initial memories ρ_0 satisfying Φ, if $p : \rho_0 \Downarrow_{\mathcal{S}} \rho, v$ then $\rho_0, \rho \vdash \Psi$.*

By correctness of the compiler, $[\![p]\!] : \rho_0 \Downarrow \rho, v$. By preservation of proof obligations, $\vdash \mathsf{VCgen}_{\mathcal{B}}([\![p]\!], \Phi, \Psi)$. By correctness of the bytecode VCgen, $\rho_0, \rho \vdash \Psi$.

4.5. Optimizations

Preservation of proof obligations does not hold in general for program optimizations, as illustrated by the following example:

$r_1 := 1$	$r_1 := 1$
$\{\mathsf{true}\}$	$\{\mathsf{true}\}$
$r_2 := r_1$	$r_2 := 1$
$\{r_1 = r_2\}$	$\{r_1 = r_2\}$

The proof obligations related to the sequence of code containing the assignment $r_2 := r_1$ is $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$ for the original and optimized version respectively. The second proof obligation is unprovable, since this proof obligation is unrelated to the sequence of code containing the assignment $r_1 := 1$.

In order to extend our results to optimizing compilers, we are led to consider certificate translation, whose goal is to transform certificates of original programs into certificates of compiled programs. Given a compiler $[\![\cdot]\!]$, a function $[\![\cdot]\!]_{\mathrm{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ", written $c : P \models \phi$), a certificate translator is a function $[\![\cdot]\!]_{\mathrm{cert}}$ such that for all programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow [\![c]\!]_{\mathrm{cert}} : [\![p]\!] \models [\![\phi]\!]_{\mathrm{spec}}$$

In [1], we show that certificate translators exist for most common program optimizations, including program transformations that perform arithmetic reasoning. For such transformations, one must rely on certifying analyzers that generate automatically certificates of correctness for the analysis, and then appeal to a weaving process to produce a certificate of the optimized program.

Whereas [1] shows the existence of certificate translators on a case-by-case basis, a companion work [2] uses the setting of abstract interpretation [7,8] to provide sufficient conditions for transforming a certificate of a program p into a certificate of a program p', where p' is derived from p by a semantically justified program transformation, such as the optimizations considered in [1].

5. Extensions to sequential Java

The previous sections have considered preservation of information flow typing and preservation of proof obligations for a simple language. In reality, these results have been proved for a sequential Java-like language with objects, exceptions, and method calls. The purpose of this section is to highlight the main issues of this extension. The main difficulties are three-fold:

- *dealing with object-orientation*: Java and JVM constructs induce a number of well-known difficulties for verification. For instance, method signatures (for type systems) or specifications (for logical verification) are required for modular verification. In addition, signatures and specifications must account for all possible termination behaviors; in the case of method specifications, it entails providing exceptional post-conditions as well as normal post-conditions. Furthermore, signatures and specifications must be compatible with method overriding;
- *achieving sufficient precision*: a further difficulty in scaling up our results to a Java-like language is precision. The presence of exceptions and object-orientation yields a significant blow-up in the control flow graph of the program, and, if no care is taken, may lead to overly conservative type-based analyses and to an explosion of verification conditions. In order to achieve an acceptable degree of usability, both the information flow type system and the verification condition generator need to rely on preliminary analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will

perform safety analyses such as class analysis, null pointer analysis, exception analysis, and array out-of-bounds analysis. These analyses drastically improve the quality of the approximation of the control flow graph (see [4] forthe case of null pointer exceptions). In particular, one can define a tighter successor relation $\mapsto$ that leads to more precise cdr information and thus typing in the case of information flow [3], and to more compact verification conditions in the case of functional verification [11];

- *guaranteeing correctness of the verification mechanisms*: the implementation of type-based verifiers and verification condition generators for sequential Java bytecode are complex programs that form the cornerstone of the security architectures that we propose. It is therefore fundamental that their implementation is correct, since flaws in the implementation of a type system or of a verification condition generator can be exploited to launch attacks. We have therefore used the Coq proof assistant [**?**] to certify both verification mechanisms. The verification is based on Bicolano, a formal model of a fragment of the Java Virtual Machine in the Coq proof assistant. In addition to providing strong guarantees about the correctness of the type system and verification condition generator, the formalization serves as a basis for a Foundational Proof Carrying Code architecture. A distinctive feature of our architecture is that both the type system and the verification condition generator are executable inside higher order logic and thus one can use reflection for verifying certificates. As compared to Foundational Proof Carrying Code [**?**], which is deductive in nature, reflective Proof Carrying Code exploits the interplay between deduction and computation to support efficient verification procedures and compact certificates.

6. Conclusion

Popular verification environments such as Jif (for information flow) and ESC/Java (for functional verification) target source code, and thus do not address directly the concerns of mobile code, where code consumers require guarantees on the code they download and execute. The purpose of these notes has been to demonstrate in a simplified setting that one can bring the benefits of source code verification to code consumers by developing adequate verification methods at bytecode level and by relating them suitably to source code verification.

Acknowledgments This work is partially supported by the EU project MOBIUS, and by the French ANR project PARSEC.

References

[1] A. W. Appel. Foundational Proof-Carrying code. In *Proceedings of LICS'01*, pages 247–258. IEEE Computer Society, 2001.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, Seoul, Korea, August 2006. Springer-Verlag.

[3] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[4] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 125 – 140. Springer-Verlag, 2007.

[5] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML — progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[6] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.

[9] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.

[10] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[11] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[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. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[15] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

[16] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3-4):303–334, 2003.

[17] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, January 2003.

[18] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.