Static Enforcement of Information Flow Policies for a Concurrent JVM-like Language*

Gilles Barthe and Exequiel Rivas

IMDEA Software Institute, Madrid, Spain

Abstract. An essential security goal of mobile code platforms is to protect confidential data against untrusted third-party applications; yet, prevailing mechanisms for ensuring confidentiality of mobile code are limited to sequential programs, whereas existing applications are generally concurrent. To bridge this gap, we develop a sound information-flow type system for a JVM-like, low-level concurrent object-oriented language. The type system builds upon existing solutions for object-oriented languages and concurrency, solving a number of intricate issues in their combination. Moreover, we connect the type system for bytecode programs to a type system for Java programs, extending the results of type-preserving compilation developed in earlier works.

1 Introduction

Non-interference is a baseline information flow policy which provides strong end-to-end security guarantees about programs, ensuring that they do not reveal confidential data throughout execution. Because information flow policies focus on program behavior rather than program origin, they are suitable for mobile code security, where confidential data must be protected against applications originating from untrusted third parties. Hence, there have been efforts to enhance the Java bytecode verifier, one of the two main security mechanisms of the Java Virtual Machine, so that it provides a type-based enforcement mechanism for non-interference. For instance, Barthe, Pichardie and Rezk [4] give a sound information flow type system for a large fragment of the Java Virtual Machine, including objects and arrays, methods, and exceptions. One important limitation of [4] is that it is restricted to sequential programs. Indeed, mobile code applications are generally concurrent, and hence cannot be handled by the type system.

Developing flexible information flow type systems for concurrent languages is notoriously difficult. Indeed, the concurrent execution of two secure programs may be insecure [14], and hence information flow type systems impose stringent restrictions on programs. In [12,13], Russo and Sabefeld introduce the notion of secure scheduler to provide a more flexible (and yet sound) approach to handle concurrency. Building on their idea, Barthe, Rezk, Russo and Sabelfeld [5] give a modular method for extending an information flow type system for a sequential low-level language into an information flow type system for a concurrent low-level language, and instantiate their method to

* Partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS.

R. Bruni and V. Sassone (Eds.): TGC 2011, LNCS 7173, pp. 73–88, 2012.

a minimal stack-based language that can serve as a target for compiling WHILE programs; the crux of the method is to make schedulers aware of the security environment that maps program points to a security level that lowers bound their effect, and to require that schedulers do not leak information through internal timing. Although the method is modular, its application to the type system of [4] raises a significant difficulty, because the soundness of the latter is based on a mix-step semantics, in which method calls are performed in one step. While the mix-step semantics makes soundness proofs considerably simpler, it is inappropriate for reasoning about concurrent programs, since the execution of separate threads might be interleaved. In this paper, we address this problem by enhancing the proof method of [4] so that all reasoning about program executions is performed directly with respect to a small-step semantics. This requires substantial generalizations in the notions of state equivalence, and substantial adaptations in the unwinding lemmas. Moreover, we provide a more realistic treatment of thread creation, so that it matches more closely the operational semantics of the JVM, and recast one technical hypothesis of [5] in terms of control dependence regions, making its verification simpler.

Moreover, we consider the issue of type-preserving compilation: we define an information flow type system for a concurrent fragment of Java and show that non-optimizing compilation preserves typability of programs, thereby generalizing [3,5]. Besides establishing that developers and consumers views on secure information flow (respectively embodied in the Java and JVM type systems) coincide, our result allows us to derive that the source type system is sound, i.e. enforces non-interference. To our best knowledge, this is the first proof of non-interference for a concurrent fragment of Java.

In summary, the main contributions of this article are:

- we provide the first sound information-flow type system for low-level, concurrent, object-oriented language;
- we define an information flow type system for a concurrent fragment of Java, prove type-preserving compilation, and derive soundness of the type system;
- we simplify one technical hypothesis in [5] that was extremely difficult to establish, even for a simple language.

2 A Concurrent JVM Language

We consider a concurrent JVM-like language, but omit some features most notably locks and exceptions.

Programs. Figure 2 gives the definition of programs. Informally, a program P is given by: i) a set $\mathcal{C}$ of classes, including a **main** class; ii) a set $\mathcal{F}$ of fields, and a function $\mathsf{fieldsof} : \mathcal{C} \to \wp(\mathcal{F})$ that yields a partition of $\mathcal{F}$; iii) a set $\mathcal{M}_{id}$ of method identifiers; iv) a set $\mathcal{M}_{vt}$ of virtual method identifiers, including an identifier run reserved for methods that run as new threads; v) a set $\mathcal{M}$ of methods, including a method identifier, a virtual method identifier, an argument table that maps argument positions to local variable names, and a sequence of instructions from Figure 1; vi) a function lookup attached to each program that takes a method identifier and a class name and returns the

$$\mathsf{Instr} ::= \mathsf{binop}\ op \mid \mathsf{push}\ n \mid \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{ifeq}\ j \mid \mathsf{goto}\ j \mid \mathsf{invokevirtual}\ m_{ID} \mid \mathsf{return} \mid \mathsf{start}$$

where op is a binary operation, n is an integer, i and j are natural numbers, and x is an identifier.

Fig. 1. Instruction set

$$\begin{array}{rll}
\mathcal{C} \ni C & & \text{(classes)} \\
\mathcal{F} \ni f & & \text{(fields)} \\
\mathcal{C} \to \wp(\mathcal{F}) \ni \mathsf{fieldsof} & & \text{(class description)} \\
\mathcal{M}_{id} \ni m_{id} & & \text{(method ids.)} \\
\mathcal{M}_{vt} \ni m_{vt} & & \text{(virt. method ids.)} \\
\mathsf{Instr} \ni ins & & \text{(instruction)} \\
\mathsf{List} \ni lis ::= ins :: lis \mid ins & & \text{(instruction listing)} \\
\mathbb{N} \rightharpoonup \mathcal{I} \ni \mathsf{arg} & & \text{(argument table)} \\
\mathcal{M} \ni m ::= \langle m_{vt}, m_{id}, \mathsf{arg}, lis \rangle & & \text{(methods)} \\
\mathcal{M}_{id} \times \mathcal{C} \to \mathcal{M} \ni \mathsf{lookup} & & \text{(class lookup)} \\
\mathcal{P} \ni i ::= \langle n, m_{id} \rangle & & \text{(program points)} \\
\mathsf{Program} \ni P ::= \langle \mathcal{C}, \mathcal{F}, \mathsf{fieldsof}, \mathcal{M}, \mathcal{M}_{id}, \mathcal{M}_{vt}, \mathsf{lookup} \rangle & & \text{(programs)}
\end{array}$$

Fig. 2. Programs

method to be executed. A program point consists of a method m, and an integer i that is smaller than the length of the list of instructions attached to m, and only i is used if the method is understood from context. We let $P_m[i]$ denote the instruction at index i of method m and let $\mathcal{P}$ denote the set of program points. We write arg_m for the argument table corresponding to method m, and assume a set $\mathcal{I}$ of variable identifiers.

States. Figure 2 introduces the semantic domains used to interpret the program behavior. Informally, a state is a set of threads indexed by thread identifiers together with a heap. The local state of a thread is given by a stack of frames, where each frame contains some local state used to execute a method invocation. The heap is modeled as partial maps from locations to objects; objects are themselves modeled as pairs, consisting of a class and of a partial map from fields to values. We assume heaps satisfy standard well-formedness constraints, e.g the domain of p include the fields of C, and o maps fields to values of the correct type. Values can be either integer numbers or locations, i.e. $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{\mathsf{null}\}$, where $\mathcal{L}$ is an (infinite) set of locations used to store the address of an object in the heap and null denotes the null pointer. Following [4], we avoid pointer arithmetic by making a distinction between locations and integer values. A set of accessors that we use to extract information from frames and states is defined.

Thread-local semantics. The operational semantics of JVM programs is defined in two steps: we first define a thread-local semantics that characterize the effect of executing

$$
\begin{array}{rll}
\mathcal{L} \ni \ell & & \text{(locations)} \\
\mathcal{V} \ni v & ::= \mathbb{Z} \cup \mathcal{L} \cup \{\mathsf{null}\} & \text{(values)} \\
\mathcal{V}^* \ni s & ::= v :: s \mid \epsilon & \text{(operand stacks)} \\
\mathcal{F} \rightharpoonup \mathcal{V} \ni p & & \text{(properties)} \\
\mathcal{O} \ni o & ::= \langle C, p \rangle & \text{(objects)} \\
\mathbb{N} \rightharpoonup \mathcal{V} \ni \rho & & \text{(environments)} \\
\mathsf{Frame} \ni f & ::= \langle m_{id}, i, \rho, s \rangle & \text{(frames)} \\
\mathsf{Thread} \ni t & ::= f :: t \mid f & \text{(threads)} \\
\mathcal{T} \rightharpoonup \mathsf{Thread} \ni ts & & \text{(thread set)} \\
\mathsf{Mem} = \mathcal{L} \to \mathcal{O} \ni \mu & & \text{(heaps)} \\
\mathsf{State} \ni s & ::= \langle ts, \mu \rangle & \text{(states)}
\end{array}
$$

$$
\begin{array}{lll}
f.\mathsf{m} \Rightarrow m_{id} & f.\mathsf{s} \Rightarrow s & s.\mathsf{ts} \Rightarrow ts \\
f.\mathsf{pc} \Rightarrow i & t.\mathsf{s} \Rightarrow \mathsf{hd}(t).\mathsf{s} & s.\mathsf{act} \Rightarrow \mathsf{dom}(ts) \\
f.\mathsf{env} \Rightarrow \rho & t.\mathsf{pc} \Rightarrow \mathsf{hd}(t).\mathsf{pc} & s.\mathsf{mem} \Rightarrow \mu
\end{array}
$$

Fig. 3. Semantic domains and accessors

an instruction on a thread. We omit its definition, which is totally standard, and can be found in the full version of the paper. The thread-local semantics is defined as a relation $\rightsquigarrow_{\mathrm{s}} \subseteq \mathsf{Thread} \times \mathsf{Mem} \times \mathsf{Thread}_{\perp} \times \mathsf{Mem}$: we write $\langle t, \mu \rangle \rightsquigarrow_{\mathrm{s}} \langle t', \mu' \rangle$ instead of $\langle t, \mu, t', \mu' \rangle \in \rightsquigarrow_{\mathrm{s}}$ as is standard. Note that termination of a thread corresponds to $t' = \perp$. The semantics is parametrized by a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$ that returns a default object of the corresponding class and it is used to create the new objects.

Global semantics. The semantics of programs is parametrized by a scheduler, that picks a thread to be executed, according to the current state, and a history h. A history is defined as a sequence of thread identifiers; we let $\mathcal{H}$ denote the set of histories, and let ϵ denote the initial history. We model a scheduler as a function $\mathsf{pickt} : \mathsf{State} \times \mathcal{H} \to \mathcal{T}$ that picks the next instruction to be executed. Moreover, we assume given a function $\mathsf{updh} : \mathsf{State} \times \mathcal{H} \to \mathcal{H}$ that updates the history after program execution. Formally, one-step execution is defined as a relation between pairs of states and histories. We write $\langle s, h \rangle \rightsquigarrow \langle s', h' \rangle$ when executing s with history h leads to state s' with history h'; the rules for the semantics are given in Figure 4. The semantics considers three cases: i) the instruction to be executed spawns a new thread; ii) it terminates the execution of a thread (in case the instruction is a return instruction and the thread has a single frame); iii) otherwise, the thread-local effect of the instruction is propagated to the active thread. We assume given a function fresht that takes as input a set of thread identifiers and generates a new thread identifier. Finally, we define the evaluation semantics.

Definition 1 (Evaluation semantics). *The evaluation relation* $\Downarrow \subseteq (\mathsf{State} \times \mathcal{H}) \times \mathsf{Mem}$ *is defined as:* $s, h \Downarrow \mu'$ *iff there is a pair* s', h' *such that* $s, h \rightsquigarrow^* s', h'$ *and* $s'.\mathsf{act} = \emptyset$ *and* $s'.\mathsf{mem} = \mu$. *We write* $P, \mu \Downarrow \mu'$ *as a shorthand for* $s_{init}(\mu), \epsilon \Downarrow \mu'$, *where* $s_{init}(\mu)$ *stands for* $\langle \{\mathsf{fresht}(\emptyset) \mapsto \langle 1_{\mathsf{main.run}}, \rho, \epsilon \rangle\}, \mu \rangle$ *where* ρ *is an empty environment.*

$$\frac{\mathsf{pickt}(s,h) = ctid \quad \mathsf{updh}(s,h) = h' \quad s_{ctid}.\mathsf{pc} = i \quad P_m[i] = \mathsf{start} \quad s_{ctid}.\mathsf{os} = o :: os \quad \mathsf{fresht}(s) = ntid \quad r = \mathsf{lookup}(\mathsf{run}, \mathsf{class}(s.\mathsf{mem}(o))) \quad s_{ctid}.[\mathsf{pc} := i+1] = \sigma'}{s, h \rightsquigarrow s.[\mathsf{lst}(ctid).\mathsf{lst} := \sigma', \mathsf{lst}(ntid) := \langle\langle 1, \{this \mapsto o\}, \epsilon\rangle\rangle], h'} (\text{Spawn})$$

$$\frac{\mathsf{pickt}(s,h) = ctid \quad \mathsf{updh}(s,h) = h' \quad s_{ctid}.\mathsf{pc} = i \quad P_m[i] = \mathsf{return} \quad \langle s_{ctid}.\mathsf{ts}, s.\mathsf{mem}\rangle \rightsquigarrow_s \langle \bot, \mu\rangle}{s, h \rightsquigarrow s.[\mathsf{lst} := \mathsf{lst} \setminus ctid, \mathsf{mem} := \mu], h'} (\text{Kill})$$

$$\frac{\mathsf{pickt}(s,h) = ctid \quad \mathsf{updh}(s,h) = h' \quad s_{ctid}.\mathsf{pc} = i \quad P_m[i] \neq \mathsf{start} \quad \langle s(ctid).\mathsf{ts}, s.\mathsf{mem}\rangle \rightsquigarrow_s \langle \sigma, \mu\rangle}{s, h \rightsquigarrow s.[\mathsf{lst}(ctid) := \sigma, \mathsf{mem} := \mu], h'} (\text{Seq})$$

Fig. 4. Semantics of multithreaded programs

Next, we define non-interference using the evaluation semantics, and characterize the class of schedulers for which our type system enforces non-interference.

3 Policy, Schedulers and Type System

This section presents an information flow type system for the concurrent language described in the previous section. The type system, and the policy it enforces, rely on security annotations on programs. The annotations use security levels, drawn from the set $\mathcal{S} = \{L, H\}$; the set $\mathcal{S}$ is given a lattice structure by declaring $L \sqsubseteq H$. Figure 5 gives a summary of the security annotations for each program. They include: i) a function ft that assigns security levels to fields; ii) a security environment that assigns security levels to program points, and is used to track implicit flows, and; iii) a method signature table, that assigns to each virtual method a security signature; a signature for a virtual method identifier m consists of a function k_v that gives the security levels describing the security level for each of its local variables, a level k_h that gives the effect of the method on the heap (in the sequel we write $\mathsf{eff}(m)$ for k_h), and a security level k_r for the value returned.

3.1 Policy

The definition of non-interference relies on memory indistinguishability. Following [1,4], we allow the allocator to be non-deterministic and use a partial bijection β on locations to accommodate dynamic creation of low objects. More precisely, the partial bijection β is used to track the relation between low objects, since, given two identical states,

$$\begin{array}{rll}
\mathcal{S} \ni s ::= L \mid H & & \text{(security levels)} \\
\mathcal{F} \to \mathcal{S} \ni \mathsf{ft} & & \text{(security level of fields)} \\
\mathcal{I} \rightharpoonup \mathcal{S} \ni \Gamma & & \text{(local variables security levels)} \\
\mathsf{SE} = \mathcal{P} \to \mathcal{S} \ni se & & \text{(security environments)} \\
\mathsf{Sig} \ni sig ::= \Gamma \xrightarrow{s} s & & \text{(signatures)} \\
\mathcal{M}_{vt} \to \mathsf{Sig} \ni \mathsf{mt} & & \text{(method table)}
\end{array}$$

Fig. 5. Security setting

a non-deterministic allocator may allocate objects in different locations. High objects (i.e. objects created in a high state) are considered indistinguishable, and the partial bijection β needs not track them.

Definition 2. *Let β be a partial bijection between locations.*

- *Value indistinguishability is defined by the clauses:*

$$\frac{}{\mathsf{null} \overset{v}{\sim}_{\beta,L} \mathsf{null}} \qquad \frac{}{v \overset{v}{\sim}_{\beta,H} v'} \qquad \frac{v \in \mathbb{Z}}{v \overset{v}{\sim}_{\beta,L} v} \qquad \frac{v, v' \in \mathcal{L} \quad \beta(v) = v'}{v \overset{v}{\sim}_{\beta,L} v'}$$

- *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable, written $o_1 \overset{o}{\sim}_\beta o_2$, iff o_1 and o_2 are of the same class, and $o_1.f \overset{v}{\sim}_{\beta,\mathsf{ft}(f)} o_2.f$ for all fields f in o_1.*
- *Two memories μ_1 and μ_2 are indistinguishable w.r.t. β, written $\mu_1 \overset{\mu}{\sim}_\beta \mu_2$, iff $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(\mu_1)$, $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(\mu_2)$ and for every $l \in \mathsf{dom}(\beta)$, we have $\mu_1(l) \overset{o}{\sim}_\beta \mu_2(\beta(l))$.*

We can now define our policy: following [4,5], we consider termination-insensitive non-interference. This policy is more permissive than bisimulation-based properties, which are overly conservative for most purposes.

Definition 3 (Non-interfering program). *A program P is non-interfering if for all memories μ_1, μ_2, μ_1', μ_2', and partial bijection β, there exists a partial bijection $\beta' \supseteq \beta$ s.t.*

$$\mu_1 \overset{\mu}{\sim}_\beta \mu_2 \text{ and } P, \mu_1 \Downarrow \mu_1' \text{ and } P, \mu_2 \Downarrow \mu_2' \text{ implies } \mu_1' \overset{\mu}{\sim}_{\beta'} \mu_2'$$

3.2 Schedulers

Execution of a concurrent JVM program proceeds by invoking the scheduler to select an active thread, and executing the current instruction for the selected thread. Following [12,13,5], we prevent internal timing leaks by instrumenting the state with security-relevant information, and making the scheduler aware of the security levels at which threads are executing. To this end, we separate thread identifiers in two sets $\mathcal{T}_H$ of high threads and $\mathcal{T}_L$ of low threads; moreover, we assume that $\mathsf{fresht}(\emptyset)$ is a low thread.

The critical property of a secure scheduler is that it always executes critical threads whenever they arise. A low thread becomes critical when it branches over a high value or it performs a method call on a high object. When it happens, we require that the scheduler executes the critical thread, until it terminates the execution of the conditional branch, or it returns from the high method. Formally, we say that a frame f is low, written $\mathsf{L}(f)$, if $se(f.\mathsf{pc}) = L$ and $\mathsf{eff}(f.\mathsf{m}) = L$, and that a thread t is low, written $\mathsf{L}(t)$, if all its frames are. The set of non-critical and critical threads of a state are defined as:

$$s.\mathsf{lowT} = \{tid \in s.\mathsf{act} \cap \mathcal{T}_L \mid \mathsf{L}(s_{tid})\} \qquad s.\mathsf{critT} = (s.\mathsf{act} \cap \mathcal{T}_L) \setminus s.\mathsf{lowT}$$

Note that, in comparison to [5], we omit the definition of always high threads, since such an analysis breaks modularity for languages with method calls.

The other essential property of a secure scheduler is that, whenever it chooses a low thread, its choice only depends on the low part of the history. To formalize the latter

notion, we must taint in histories the instances of thread identifiers that correspond to the execution of critical threads. Then, we define the low part $h_{|L}$ of a history h as the subsequence of h where high thread identifiers and tainted instances of thread identifiers are removed.

Definition 4 (Secure scheduler). *A function* $\mathsf{pickt} : \mathsf{State} \times \mathcal{H} \to \mathcal{T}$ *is a secure scheduler iff for all states* s *and* s' *and for all histories* h *and* h'*:*

1. $\mathsf{pickt}(\langle s, h\rangle) \in s.\mathsf{act}$
2. *if* $s.\mathsf{crit} \neq \emptyset$*, then* $\mathsf{pickt}(\langle s, h\rangle) \in s.\mathsf{crit}$*;*
3. *if* $\mathsf{pickt}(s, h) \in s.\mathsf{lowT}$ *and* $s'.\mathsf{crit} = \emptyset$ *and* $s'.\mathsf{act} \neq \emptyset$*, and* $s.\mathsf{lowT} = s'.\mathsf{lowT}$*, and* $h_{|L} = h'_{|L}$*, then* $\mathsf{pickt}(s, h) = \mathsf{pickt}(s', h')$.

Examples of secure schedulers include (mildly adapted) round robin schedulers [5].

3.3 Control Dependence Regions

The definition of the type system is parametrized by an approximation of control dependence regions. This approximation is used in the typing rule for branching statements to prevent implicit flows.

Control dependence regions are formulated in terms of the successor relation $\mapsto$, which is defined by the clauses: i) $i \mapsto i + 1$ if $P_m[i]$ is of the form $\mathsf{binop}\ op$, $\mathsf{push}\ n$, $\mathsf{load}\ x$, $\mathsf{store}\ x$, $\mathsf{invokevirtual}\ m_{ID}$, start, $\mathsf{ifeq}\ j$, $\mathsf{getfield}\ f$, $\mathsf{putfield}\ f$, or $\mathsf{new}\ C$; ii) $i \mapsto j$ if $P_m[i]$ is of the form $\mathsf{goto}\ j$, or $\mathsf{ifeq}\ j$; iii) $i \mapsto$, i.e. i does not have a successor if $P_m[i]$ is a return instruction. We let $\mapsto^*$ denote the reflexive-transitive closure of $\mapsto$. Moreover, we say that i is a branching point iff there exists program points j and k such that $j \neq k$, $i \mapsto j$ and $i \mapsto k$. We let $\mathcal{P}^\sharp$ be the set of branching points.

In the sequel, we consider that programs come equipped with a CDR structure $\langle \mathsf{region}, \mathsf{jun}\rangle$, where region is a function from branching points to sets of program points and jun is a partial function from branching points to program points, such that for all program points i, j and k:

CDR1 if $i \mapsto j$ then $j \in \mathsf{region}(i)$ or $j = \mathsf{jun}(i)$;
CDR2 if $j \in \mathsf{region}(i)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;
CDR3 if $j \in \mathsf{region}(i)$ and $j \mapsto$ then $\mathsf{jun}(i)$ is undefined;
CDR4 if $j \in \mathsf{region}(k)$ and $i \mapsto j$ then $i \in \mathsf{region}(k)$ or $k = i$;
CDR5 if $\mathsf{jun}(i) \in \mathsf{region}(j)$ then $\mathsf{region}(i) \subseteq \mathsf{region}(j)$;
CDR6 if $\mathsf{region}(i) \cap \mathsf{region}(j) \neq \emptyset$ then $\mathsf{region}(i) \subseteq \mathsf{region}(j)$ or $\mathsf{region}(j) \subseteq \mathsf{region}(i)$;
CDR7 if $i \in \mathsf{region}(j)$, then $j \mapsto^* i$.

In comparison with earlier works, e.g. [4], the last four CDR assumptions are new. These additional assumptions have a limited impact in our development: we briefly comment on two key points. On the one hand, our infrastructure requires that a CDR checker is executed prior to type checking. In order to ensure soundness of our verification method, the CDR checker must therefore be enhanced to verify these additional properties; fortunately, the additional verifications can be performed without increasing the complexity of the algorithm. On the other hand, type-preserving compilation

also requires showing that source programs are compiled into JVM programs and CDR structures that are accepted by the checker. Thus, we must verify for each compiler that the CDR structures it outputs verify the additional assumptions. This is the purpose of Proposition 2.

On the other hand, these additional assumptions allow a direct proof of the existence of a function next that returns the next (relative to the successor relation) junction point that is not inside a high region. Before we proceed with the definition of the next function, we observe that extending CDR checkers so that they verify the above hypotheses, allows us to define a security environment that respects the CDR structure. It is assumed a function ise that each branching point is mapped to the security level of the value over which the branch is done.

Proposition 1 (Next function). *Let* $\langle \mathsf{region}, \mathsf{jun} \rangle$ *be a CDR structure. Moreover, let* $ise : \mathcal{P}^\sharp \to \mathcal{S}$ *and for all* $i \in \mathcal{P}$*, let* $\overline{i} = \{k \in \mathcal{P}^\# : i \in \mathsf{region}(k) \wedge ise(k) = H\}$*. Set the security environment* se *as follows:*

$$se(i) = \begin{cases} H \text{ if } \overline{i} \neq \emptyset \\ L \text{ if } \overline{i} = \emptyset \end{cases}$$

Then there exists a function $\mathsf{next} : \mathcal{P} \rightharpoonup \mathcal{P}$ *such that the following properties hold:*

NePd $\mathsf{dom}(\mathsf{next}) = \{i \in \mathcal{P} : se(i) = H\}$
NeP1 $i, j \in \mathsf{dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = \mathsf{next}(j)$
NeP2 $i \in \mathsf{dom}(\mathsf{next}) \wedge j \notin \mathsf{dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = j$
NeP3 $j, k \in \mathsf{dom}(\mathsf{next}) \wedge i \notin \mathsf{dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = \mathsf{next}(k)$
NeP4 $i, j \in \mathsf{dom}(\mathsf{next}) \wedge k \notin \mathsf{dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = k$

The proof of soundness makes use of the existence of the next function.

3.4 Type System

The type system adopts the principles of bytecode verification. It performs a modular (method-wise) data flow analysis that operates on stack of security levels, known as security stacks, and outputs a security type, i.e. a mapping from program points to security stacks:

$$\begin{aligned} \mathcal{ST} \ni st ::= s :: st \mid \epsilon & \quad \text{(security stacks)} \\ \mathsf{Type} = \mathcal{P} \to \mathcal{ST} \ni S & \quad \text{(security types)} \end{aligned}$$

More specifically, the type system is defined by a set of transfer rules that relate the stack type at a given program point to the stack type of its successors. The transfer rules are given in Figure 6. Their judgments are of the form $\mathsf{mt}, \mathsf{region}, se, i \vdash st \Rightarrow st'$, where st and st' are stack types; we simply write $i \vdash st \Rightarrow st'$ when the other components are understood from the context.

Definition 5 (Typable program). *A program* P *is typable w. type* S*, written* $S \vdash P$*, if*

1. $S(\langle 1, m_{id} \rangle) = \epsilon$*, for all methods* m_{id}*; and*
2. $\mathsf{mt}, \mathsf{region}, i, se \vdash S(i) \Rightarrow st$ *where* $st \sqsubseteq S(j)$*, for all program points* i *and* j *such that* $i \mapsto j$*; and*
3. $\mathsf{mt}, \mathsf{region}, i, se \vdash S(i) \Rightarrow$*, for all program point* i *such that* $P_m[i] = \mathsf{return}$.

$$\frac{P_m[i] = \mathsf{push}\ n}{\mathsf{mt}, \mathsf{region}, i, se \vdash st \Rightarrow se(i) :: st} \qquad \frac{P_m[i] = \mathsf{load}\ x \quad \mathsf{mt}(m) = k_v \xrightarrow{k_h} k_r}{\mathsf{mt}, \mathsf{region}, i, se \vdash st \Rightarrow (se(i) \sqcup k_v(x)) :: st}$$

$$\frac{P_m[i] = \mathsf{goto}\ j}{\mathsf{mt}, \mathsf{region}, i, se \vdash st \Rightarrow st} \qquad \frac{P_m[i] = \mathsf{binop}\ op}{\mathsf{mt}, \mathsf{region}, i, se \vdash k_1 :: k_2 :: st \Rightarrow (se(i) \sqcup k_1 \sqcup k_2) :: st}$$

$$\frac{P_m[i] = \mathsf{store}\ x \quad \mathsf{mt}(m) = k_v \xrightarrow{k_h} k_r \quad se(i) \sqcup k \sqsubseteq k_v(x)}{\mathsf{mt}, \mathsf{region}, i, se \vdash k :: st \Rightarrow st}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{vt} \quad \mathsf{mt}(m_{ID}) = k_v \xrightarrow{k_h} k_r \quad k :: \bar{k} \sqsubseteq k_v.\,\mathsf{arg}_{m_{vt}} \quad k \sqcup se(i) \sqcup \mathsf{eff}(m) \sqsubseteq k_h}{\mathsf{mt}, \mathsf{region}, i, se \vdash \bar{k} :: k :: st \Rightarrow (k_r \sqcup k_h) :: st}$$

$$\frac{P_m[i] = \mathsf{ifeq}\ j \quad \forall j' \in \mathsf{region}(i), k \sqsubseteq se(j')}{\mathsf{mt}, \mathsf{region}, i, se \vdash k :: st \Rightarrow \mathsf{lift}_k(st)} \qquad \frac{P_m[i] = \mathsf{return} \quad se(i) \sqcup k \sqsubseteq k_r \quad se(i) \sqsubseteq \mathsf{eff}(m)}{\mathsf{mt}, \mathsf{region}, i, se \vdash k :: st \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{start} \quad k \sqcup se(i) \sqcup \mathsf{eff}(m) \sqsubseteq \mathsf{eff}(\mathsf{run})}{\mathsf{mt}, \mathsf{region}, i, se \vdash k :: st \Rightarrow st} \qquad \frac{P_m[i] = \mathsf{getfield}\ f}{\mathsf{mt}, \mathsf{region}, i, se \vdash k :: st \Rightarrow (k \sqcup \mathsf{ft}(f) \sqcup se(i)) :: st}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \quad k_1 \sqcup se(i) \sqcup k_2 \sqsubseteq \mathsf{ft}(f) \quad \mathsf{eff}(m) \sqsubseteq \mathsf{ft}(f)}{\mathsf{mt}, \mathsf{region}, i, se \vdash k_1 :: k_2 :: st \Rightarrow st}$$

Fig. 6. Transfer rules

4 Soundness

This section establishes the soundness property for the type system presented in the previous section: we prove that typable programs are non-interfering. The overall structure of the proof is similar to [4,5]: one first proves locally-respects and step-consistent unwinding lemmas, and then one derives a locally-respects unwinding lemma for a notion of visible execution; finally, one proves non-interference by an argument on traces. The former critically relies on an appropriate notion of state indistinguishability, whereas the latter relies on properties of the control dependence regions and the security environment. In both cases, significant extensions to [4,5] are required. In particular, the notion of state equivalence must be extended to accommodate threads, i.e. stacks of frames, rather than a single frame. The intuition is that two threads are indistinguishable if their frames are in one-to-one correspondence[1] until there is a call to a high method in both threads, or one of the threads enters in a high branch. Throughout this section, we assume that P is typable program whose type is S, i.e. $S \vdash P$.

We first define state equivalence. As the high threads cannot modify the low part of the heap, state equivalence only depends on low threads, and the heap.

Definition 6 (State equivalence). *Let β be a partial bijection on locations.*

– *Operand stack equivalence is defined inductively by the clauses:*

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

where $\mathsf{highos}(os, st)$ *iff* $|os| = |st|$ *and* $st(i) = H$ *for all* $1 \leq i \leq |st|$.

[1] This correspondence is expressed below by the auxiliary relation $\overset{\ell}{\sim}$.

– *Frame equivalence is defined by the clause:*

$$f \overset{\mathsf{s}}{\sim}_\beta f' \;\; \textit{iff} \;\; f.\mathsf{os} : S(f.\mathsf{pc}) \overset{\mathsf{os}}{\sim}_\beta f'.\mathsf{os} : S(f'.\mathsf{pc}) \;\textit{and}\; f.\mathsf{m} = f'.\mathsf{m} \;\textit{and}\; f.\mathsf{env} \overset{\mathsf{env}}{\sim}_{\beta, f.\mathsf{m}} f'.\mathsf{env}$$

where $\rho \overset{\mathsf{env}}{\sim}_{\beta,m} \rho'$ *iff* $\rho(x) \overset{v}{\sim}_{\beta,k_v(x)} \rho'(x)$ *and* $\mathsf{mt}(m) = k_v \xrightarrow{k_h} k_r$.

– *Thread equivalence is defined inductively by the clauses:*

$$\frac{}{\epsilon \overset{\ell}{\sim}_\beta \epsilon} \qquad \frac{\mathsf{high}(fs_0) \quad \mathsf{high}(fs'_0) \quad fs \overset{\ell}{\sim}_\beta fs'}{fs_0 :: fs \overset{t}{\sim}_\beta fs'_0 :: fs'}$$

$$\frac{f \overset{\mathsf{s}}{\sim}_\beta f' \quad \mathsf{eff}(f.\mathsf{m}) = L \quad fs \overset{\ell}{\sim}_\beta fs' \quad \mathsf{L}(f) \wedge \mathsf{L}(f') \Rightarrow f.\mathsf{pc} = f'.\mathsf{pc}}{f :: fs \overset{\ell}{\sim}_\beta f' :: fs'}$$

where $\mathsf{high}(fs)$ *means that* $\mathsf{eff}(fs[i].\mathsf{m}) = H$ *for all* $1 \leq i \leq |fs|$.

– *State equivalence is defined by the clause:*

$$s \overset{c}{\sim}_\beta s' \;\; \textit{iff} \;\; s.\mathsf{mem} \overset{\mu}{\sim}_\beta s'.\mathsf{mem} \wedge \forall t \in s.\mathsf{act} \cap \mathcal{T}_L . s_t \overset{t}{\sim}_\beta s'_t$$

The proof of the soundness theorem introduces three new execution relations, including a notion of visible step; the formal definitions are given in Figure 7. An intuition for these execution relations is: $s, h \rightsquigarrow_{\mathrm{vis}} s', h'$ iff neither s or s' have a critical thread, and $s, h \rightsquigarrow s', h'$ or $s, h \rightsquigarrow^* s', h'$ and all intermediate states have a critical thread. For the proofs, it is convenient to divide this relation into two disjoint ones, corresponding to the reasons under which a thread can be made critical. Specifically, we define $s, h \rightsquigarrow_{\mathrm{method}} s', h'$ iff $s, h \rightsquigarrow^* s', h'$ and s, s' and all the intermediate states have a critical thread caused by a method call on a high object; and $s, h \rightsquigarrow_{\mathrm{branch}} s', h'$ iff $s, h \rightsquigarrow^* s', h'$ and s, s' and all the intermediate states have a critical thread caused by a branching over a high value. The predicates $\mathsf{C_m}$ and $\mathsf{C_b}$ determine whether a thread is high because of a branch or a method call, and NCT, $\mathsf{CT_b}$ and $\mathsf{CT_m}$ respectively indicate that the state has no critical thread; or a critical thread because of a branch; or a critical thread because of a method call. VS predicate is used to ensure threads with high identifiers do actually correspond to high threads.

Then, we prove unwinding lemmas for the newly introduced execution relations. The first two have the form of step-consistent lemmas, while the third has the form of a locally-respect lemma. For the sake of readability, we extend the next function to (critical) threads and (critical) states. For a critical thread t, $\mathsf{next}(t)$ is defined as $\mathsf{next}(i)$ where i is the program point that caused the thread to become critical. Formally,

$$\mathsf{next}(t) = \begin{cases} \mathsf{next}(\mathsf{hd}(t).\mathsf{pc}) & \text{if } se(\mathsf{hd}(t).\mathsf{pc}) = H \wedge \mathsf{eff}(\mathsf{hd}(t).\mathsf{m}) = L \wedge \mathsf{L}(\mathsf{tl}(t)) \\ \mathsf{next}(\mathsf{tl}(t)) & \text{otherwise} \end{cases}$$

Since the current thread (the one picked by the scheduler) can be deduced from the context, we further write $\mathsf{next}(s)$ for $\mathsf{next}(s_{ctid})$ where $ctid$ is the current thread.

$$\begin{array}{ll} \mathsf{C_m}(f) \equiv \mathsf{eff}(f.\mathsf{m}) = H \wedge \mathsf{highos}(f.\mathsf{pc}, S(f.\mathsf{pc})) & \mathsf{C_m}(t) \equiv \exists i.\mathsf{C_m}(t[i]) \wedge \forall j < i.\mathsf{L}(t[j]) \\ \mathsf{C_b}(f) \equiv se(f.\mathsf{pc}) = H \wedge \mathsf{eff}(f.\mathsf{m}) = L & \mathsf{C_b}(t) \equiv \exists i.\mathsf{C_b}(t[i]) \wedge \forall j < i.\mathsf{L}(t[j]) \\ \mathsf{CT_b}(s) \equiv s.\mathsf{critT} = \{tid\} \wedge \mathsf{C_b}(s_{tid}) \wedge \mathsf{VS}(s) & \mathsf{VS}(s) \equiv \forall tid \in s.\mathsf{act} \cap \mathcal{T}_H.\mathsf{C_m}(s_{tid}) \\ \mathsf{CT_m}(s) \equiv s.\mathsf{critT} = \{tid\} \wedge \mathsf{C_m}(s_{tid}) \wedge \mathsf{VS}(s) & \mathsf{NCT}(s) \equiv s.\mathsf{critT} = \emptyset \wedge \mathsf{VS}(s) \end{array}$$

$$\frac{\mathsf{NCT}(s) \quad \mathsf{NCT}(s') \quad s, h \rightsquigarrow s', h'}{s, h \rightsquigarrow_{\text{vis}} s', h'}$$

$$\frac{\mathsf{CT_b}(s) \quad \mathsf{CT_b}(s') \quad s, h \rightsquigarrow s', h'}{s, h \rightsquigarrow_{\text{branch}} s', h'} \qquad \frac{\mathsf{CT_m}(s) \quad \mathsf{CT_m}(s') \quad s, h \rightsquigarrow s', h'}{s, h \rightsquigarrow_{\text{method}} s', h'}$$

$$\frac{s, h \rightsquigarrow_{\text{branch}} s', h' \quad s', h' \rightsquigarrow_{\text{branch}} s'', h''}{s, h \rightsquigarrow_{\text{branch}} s'', h''} \qquad \frac{s, h \rightsquigarrow_{\text{method}} s', h' \quad s', h' \rightsquigarrow_{\text{method}} s'', h''}{s, h \rightsquigarrow_{\text{method}} s'', h''}$$

$$\frac{s, h \rightsquigarrow s', h' \quad s', h' \rightsquigarrow_{\text{branch}} s'', h'' \quad s'', h'' \rightsquigarrow s''', h'''}{s, h \rightsquigarrow_{\text{vis}} s''', h'''}$$

$$\frac{s, h \rightsquigarrow s', h' \quad s', h' \rightsquigarrow_{\text{method}} s'', h'' \quad s'', h'' \rightsquigarrow s''', h'''}{s, h \rightsquigarrow_{\text{vis}} s''', h'''}$$

Fig. 7. Auxiliary execution relations

Lemma 1. *If* $s, h_s \rightsquigarrow_{\text{branch}} s', h_{s'}$ *and* $s \overset{c}{\sim}_\beta t$ *then we have* $s' \overset{c}{\sim}_\beta t$, $h_s \overset{h}{\sim} h_{s'}$, *and* $\mathsf{next}(s) = \mathsf{next}(s')$.

Lemma 2. *If* $s, h_s \rightsquigarrow_{\text{method}} s', h_{s'}$ *and* $s \overset{c}{\sim}_\beta t$ *then we have* $s' \overset{c}{\sim}_\beta t$, *and* $h_s \overset{h}{\sim} h_{s'}$.

Lemma 3. *If* $s, h_s \rightsquigarrow_{\text{vis}} s', h_{s'}$ *and* $t, h_t \rightsquigarrow_{\text{vis}} t', h_{t'}$ *and* $s \overset{c}{\sim}_\beta t$ *and* $h_s \overset{h}{\sim} h_t$ *then we have* $s' \overset{c}{\sim}_{\beta'} t'$ *and* $h_{s'} \overset{h}{\sim} h_{t'}$ *for some* β' *such that* $\beta \subseteq \beta'$.

Theorem 1 (Soundness). *If* P *is typable with respect to a CDR structure* $\langle \mathsf{region}, \mathsf{jun} \rangle$, *then* P *is non-interfering.*

Proof (Sketch of proof). Suppose that $\mu_1 \overset{\mu}{\sim}_\beta \mu_2$, $P, \mu_1 \Downarrow \mu_1'$ and $P, \mu_2 \Downarrow \mu_2'$. Note that if $P, \mu_1 \Downarrow \mu_1'$ then $s_{init}(\mu_1), \epsilon \left(\rightsquigarrow_{\text{vis}}\right)^* s_1, h_{s_1}$ with $s_1.\mathsf{lowT} = \emptyset$ and $s_1.\mathsf{mem} = \mu_1'$. Analogously, $s_{init}(\mu_2), \epsilon \left(\rightsquigarrow_{\text{vis}}\right)^* s_2, h_{s_2}$ with $s_2.\mathsf{lowT} = \emptyset$ and $s_2.\mathsf{mem} = \mu_2'$. Moreover, as the $\rightsquigarrow_{\text{vis}}$ relation "steps over" the critical execution steps both executions take the same number of $\rightsquigarrow_{\text{vis}}$ steps. By lemma 3, we conclude.

5 Type-Preserving Compilation

The purpose of this section is to define an information flow type system for a concurrent fragment of Java, and to show that compilation transforms typable programs into typable programs.

5.1 Source Language and Compiler

The expressions and commands of the source language are defined in Figure 8; the overall structure of a program remains identical. The compiler must produce, in addition to

$$
\begin{aligned}
e &::= n \mid x \mid e\ op\ e \mid \mathsf{new}\ C \\
c &::= x = e \mid c\,;c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid e.f = e \mid \\
&\quad\ \mathsf{return}\ e \mid e.\mathsf{start}() \mid x = e.f \mid x = e.m(\bar{e}) \\
p &::= [\mathsf{m}(\bar{e})\{c\,;\mathsf{return}\ e\}]
\end{aligned}
$$

Fig. 8. Source language

the JVM program, a CDR structure. It is convenient for constructing the latter to consider a labeled version of the commands, and to define a mapping $\mathcal{A}$ that automatically give the commands of the source program labels such that there are no two commands in the source that have the same label; the syntax of the labeled commands in Figure 9. The definition of $\mathcal{A}$ can be found in the full version of the paper. The function $\mathcal{A}$ takes an integer, the number from where it starts to label the subcommands, and a command to be labeled, and returns a labeled command. To label a program, we use $\mathcal{A}$ in each method using as integer the length of the previous method compiled plus one and one for the first method to be compiled.

$$
\begin{aligned}
c ::= {} & [x = e]^n \mid c\,;c \mid [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c]^n \mid [\mathsf{while}\ e\ \mathsf{do}\ c]^n \mid [e.f = e]^n \mid \\
& [\mathsf{return}\ e]^n \mid [e.\mathsf{start}()]^n \mid [x = e.f]^n \mid [x = e.m(\bar{e})]^n
\end{aligned}
$$

Fig. 9. Labeled syntax

The compiler functions $[\![-]\!]$ transform expressions and commands of the source language as lists of JVM instructions; their definition is given in Figure 10. As in [3], we define for each branching command of the source language a *main instruction* in the target code. This instruction is a branching program point in the target program, and hence a program point for which we need to define the region and the junction point. Finally, programs are compiled by applying $[\![-]\!]$ to each method body.

5.2 Source Type System

The type system for the source language is defined in Figure 11. As for the JVM, the type system assumes that the program comes equipped with a mapping of fields to security levels, and a security signature for each method. Following the standard approach for source type system, we first type the expressions and then the commands. A method is well-typed if its command is typed with a type that is compatible with the method signature. A program is well-typed if all its methods are well-typed.

$$
\begin{aligned}
[\![x]\!] &= \mathsf{load}\ x \\
[\![n]\!] &= \mathsf{push}\ n \\
[\![e\ op\ e]\!] &= [\![e]\!] :: [\![e']\!] :: \mathsf{binop}\ op \\
[\![\mathsf{new}\ C]\!] &= \mathsf{new}\ C \\
[\![x = e]\!] &= [\![e]\!] :: \mathsf{store}\ x \\
[\![c_1\ ; c_2]\!] &= [\![c_1]\!] :: [\![c_2]\!] \\
[\![e.f = e']\!] &= [\![e']\!] :: [\![e]\!] :: \mathsf{putfield}\ f \\
[\![\mathsf{return}\ e]\!] &= [\![e]\!] :: \mathsf{return} \\
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] &= \mathsf{goto}\ (pc + \#lc + 1) :: lc :: le :: \underline{\mathsf{ifeq}\ (pc - \#lc - \#le)} \\
&\quad \text{where } le = [\![e]\!], lc = [\![c]\!] \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= le :: \underline{\mathsf{ifeq}\ (pc + \#lc_2 + 2)} :: lc_2 :: \mathsf{goto}\ (pc + \#lc_1 + 1) :: lc_1 \\
&\quad \text{where } le = [\![e]\!], lc_1 = [\![c_1]\!], lc_2 = [\![c_2]\!] \\
[\![e.\mathsf{start}()]\!] &= [\![e]\!] :: \mathsf{start} \\
[\![x = e.f]\!] &= [\![e]\!] :: \mathsf{getfield}\ f :: \mathsf{store}\ x \\
[\![x = e.m(\bar{e})]\!] &= [\![\bar{e}]\!] :: [\![e]\!] :: \mathsf{invokevirtual}\ m :: \mathsf{store}\ x
\end{aligned}
$$

Fig. 10. Compilation

5.3 Preservation Proof

As in [3], the preservation proof preserves in three steps: first, we define a function sregion for the source language, which is used later to obtain a CDR structure for the compiled program. Then, we define an intermediate type system for the source language, and prove its equivalence with the original type system; the intermediate type system is defined with help of the sregion function, and facilitates the third and final step, which is the proof of type-preserving compilation itself. We highlight some critical steps below.

We first define the sregion function, which is the equivalent of a CDR structure for the source language: for each branching command $[c]^n$, we let $\mathsf{sregion}(n)$ be the set of labels that appear in c, i.e. that label the subcommands of c.

Proposition 2. *For every program P, $\langle \mathsf{tregion}, \mathsf{tjun} \rangle$ is a CDR structure for the program $[\![P]\!]$, where for every branching instruction $[c]^n$ in the source code, $\mathsf{tregion}(n)$ is defined as the set of instructions obtained by compiling commands $[c']^{n'}$, where $n' \in \mathsf{sregion}(n)$, and $\mathsf{tjun}(n) = \mathsf{max}\ \{i : i \in \mathsf{tregion}(n)\} + 1$.*

The proof involves showing that every branching program point in the target language is the image of a branching program point in the source language, and derive the CDR properties from the definition of the compiler.

We now define a function ise that associates to each branching point in the compiled program the security level of the corresponding guard in the source language. Proposition 1 yields the existence of a function next that satisfies the **NeP** hypotheses w.r.t. the security environment built from ise.

The proof of type-preserving compilation between the source and target type systems is performed first on expressions, then on commands. Finally, we conclude that

$$\frac{k_v \vdash e : k \quad k \sqsubseteq k_v(x)}{k_v \xrightarrow{k_h} k_r \vdash x = e : k_v(x)} \qquad \frac{k_v \vdash e : k \quad k_v \xrightarrow{k_h} k_r \vdash c : k}{k_v \xrightarrow{k_h} k_r \vdash \mathsf{while}\ e\ \mathsf{do}\ c : k} \qquad \frac{k_v \vdash e : k \quad k_v \xrightarrow{k_h} k_r \vdash c : k \quad k_v \xrightarrow{k_h} k_r \vdash c' : k}{k_v \xrightarrow{k_h} k_r \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : k}$$

$$\frac{k_v \vdash e : k \quad k \sqsubseteq k_r \sqcap k_h}{k_v \xrightarrow{k_h} k_r \vdash \mathsf{return}\ e : k} \qquad \frac{k_v \vdash e : k \quad k \sqcup \mathsf{ft}(f) \sqsubseteq k_v(x)}{k_v \xrightarrow{k_h} k_r \vdash x = e.f : k_v(x)} \qquad \frac{k_v \vdash e : k\ k_v \vdash e' : k'\ k \sqcup k' \sqcup k_h \sqsubseteq \mathsf{ft}(f)}{k_v \xrightarrow{k_h} k_r \vdash e.f = e' : \mathsf{ft}(f)}$$

$$\frac{k_v \xrightarrow{k_h} k_r \vdash c : k \quad k' \sqsubseteq k}{k_v \xrightarrow{k_h} k_r \vdash c : k'} \qquad \frac{k_v \vdash e : k \quad k \sqcup k_h \sqsubseteq \mathsf{eff}(\mathsf{run})}{k_v \xrightarrow{k_h} k_r \vdash e.\mathsf{start}() : \mathsf{eff}(\mathsf{run})} \qquad \frac{k_v \xrightarrow{k_h} k_r \vdash c : k \quad k_v \xrightarrow{k_h} k_r \vdash c' : 'k}{k_v \xrightarrow{k_h} k_r \vdash c\,;c' : k \sqcap k'}$$

$$\frac{\forall i. k_v \vdash e[i] : k_v(\bar{e}[i]) \quad k \sqcup k_h \sqsubseteq k'_h \quad k_v \vdash e : k \quad \mathsf{mt}(m) = k'_v \xrightarrow{k'_h} k'_r \quad k_r \sqsubseteq k'_v(x)}{k_v \xrightarrow{k_h} k_r \vdash x = e.m(\bar{e}) : k_v(x) \sqcap \mathsf{eff}(m)}$$

Fig. 11. A high-level type system for the source language

a typable source program is transformed into a JVM program that is typable w.r.t. the same policy as the source program, the CDR structure generated by the compiler, and the security environment built from ise.

Theorem 2. *Suppose we have a* $\vdash SP$*, then* $\vdash [\![SP]\!]$.

6 A Toy Example

To explore and concretise some details of the type system, we introduce a toy example. Three classes are defined: A, B, and C, with fields defined $\mathsf{fieldsof}(\mathsf{A}) = \{\mathsf{varA}\}$, $\mathsf{fieldsof}(\mathsf{B}) = \{\mathsf{varB}\}$, $\mathsf{fieldsof}(\mathsf{C}) = \{\mathsf{varL}, \mathsf{varH}\}$, and ft defined by the rules $\mathsf{ft}(\mathsf{varL}) = L$, and $\mathsf{ft}(\mathsf{varH}) = \mathsf{ft}(\mathsf{varA}) = \mathsf{ft}(\mathsf{varB}) = H$. The methods are defined in Fig. 12.

From the type system, o is high as it is set inside a high-branch. Also, as o is high, from the start typing rule, we conclude that $\mathsf{eff}(\mathsf{run})$ must be high as well. This is consistent with the typing rule for assignment, as the fields varA and varB are high too.

A small change would make the program become insecure and untypable. If we force one of varA or varB to be a low field, then the type system would reject the code as the value of varH could be determined by inspecting varA or varB.

From this example, we can learn that if a method implementing run is high, then all other instances are forced to be high too. While it seems that this could be fixed in part by considering whether start is launched from a high or low region/object, the problem

```
void A.run() {
  this.varA = 0;
}
void B.run() {
  this.varB = 1;
}
void C.run() {
  if (this.varH)
    o = new A();
  else
    o = new B();
  o.start();
  ...
}
```

Fig. 12. Code of example

is actually harder than this: we can arrive to a start instruction from different methods with different security levels.

7 Related Work

The prevalence of Java in mobile code applications makes it a natural target for information flow analysis, and there is a substantial body of literature on enforcement mechanisms for Java or JVM programs. In particular, Myers *et al* [9] have developed the Jif language, an information flow aware extension of (sequential) Java; Jif has been used for developing significant examples of secure applications, including the Civitas voting system. While there is no soundness proof for the whole Jif type system, Banerjee and Naumann [1] have developed a sound information flow type system for a significant fragment of Java. Barthe, Naumann and Rezk [3] extend the type system and soundness proof to a simplified form of exception. Our work adopts many notions and techniques from [1,9] for object-oriented programs, and [4,8] for bytecode languages. All these works are confined to sequential fragments of Java.

More recently, there have been many proposals for a dynamic approach to information flow, as developed by Le Guernic *et al* for a core imperative [7] and concurrent [6] languages. Nair *et al* [10] report on the implementation of the Trishul system, which dynamically tracks information flow in Java applications; however, no formal guarantee is established.

8 Conclusion

Developing sound and flexible information flow type systems for mobile code is an important goal for language-based security. This article leverages previous works on concurrent bytecode languages and on the sequential JVM to propose for the first time a sound information flow type system for a concurrent low-level object-oriented language. Our work, and in particular the notions of program equivalence we use, lay the foundations for designing flexible type systems for the full JVM. In the future, it would be interesting to extend the type system to exceptions (which are treated in [11]), declassification (in the style of [2]), and locks.

References

1. Banerjee, A., Naumann, D.: Stack-based access control for secure information flow. Journal of Functional Programming 15, 131–177 (2005); Special Issue on Language-Based Security
2. Barthe, G., Cavadini, S., Rezk, T.: Tractable enforcement of declassification policies. In: IEEE Computer Security Foundations Symposium (June 2008)
3. Barthe, G., Naumann, D., Rezk, T.: Deriving an information flow checker and certifying compiler for Java. In: Symposium on Security and Privacy. IEEE Press (2006)
4. Barthe, G., Pichardie, D., Rezk, T.: A Certified Lightweight Non-interference Java Bytecode Verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)

5. Barthe, G., Rezk, T., Russo, A., Sabelfeld, A.: Security of multithreaded programs by compilation. ACM Trans. Inf. Syst. Secur. 13(3) (2010)
6. Le Guernic, G.: Automaton-based confidentiality monitoring of concurrent programs. In: CSF, pp. 218–232. IEEE Computer Society (2007)
7. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-Based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2007)
8. Kobayashi, N., Shirane, K.: Type-based information analysis for low-level languages. In: Asian Programming Languages and Systems Symposium, pp. 302–316 (2002)
9. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Principles of Programming Languages, pp. 228–241. ACM Press (1999), Ongoing development at `http://www.cs.cornell.edu/jif/`
10. Nair, S.K., Simpson, P.N.D., Crispo, B., Tanenbaum, A.S.: A virtual machine based information flow control system for policy enforcement. Electr. Notes Theor. Comput. Sci. 197(1), 3–16 (2008)
11. Rezk, T.: Verification of confidentiality policies for mobile code. PhD thesis, Université de Nice Sophia-Antipolis (2006)
12. Russo, A., Sabelfeld, A.: Securing interaction between threads and the scheduler. In: Computer Security Foundations Workshop, pp. 177–189 (2006)
13. Russo, A., Sabelfeld, A.: Security for Multithreaded Programs Under Cooperative Scheduling. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 474–480. Springer, Heidelberg (2007)
14. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In: Principles of Programming Languages, pp. 355–364 (1998)