A Certified Lightweight Non-interference Java Bytecode Verifier*

Gilles Barthe[1], David Pichardie[2,**], and Tamara Rezk[1]

[1] INRIA Sophia Antipolis, France
[2] IRISA/INRIA Rennes, France

Abstract. Non-interference is a semantical condition on programs that guarantees the absence of illicit information flow throughout their execution, and that can be enforced by appropriate information flow type systems. Much of previous work on type systems for non-interference has focused on calculi or high-level programming languages, and existing type systems for low-level languages typically omit objects, exceptions, and method calls, and/or do not prove formally the soundness of the type system. We define an information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and method calls, and prove that it guarantees non-interference. For increased confidence, we have formalized the proof in the proof assistant Coq; an additional benefit of the formalization is that we have extracted from our proof a certified lightweight bytecode verifier for information flow. Our work provides, to our best knowledge, the first sound and implemented information flow type system for such an expressive fragment of the JVM.

1 Introduction

Starting from the work of Volpano and Smith [21], type systems have become a popular means to enforce information flow policies in programming languages [19]. It is striking to notice that, although mobile code security is one central motivation behind those works, there has been very little effort to study information flow in low-level languages such as Java bytecode. While focusing on source languages is useful to provide developers with assurance that their code does not leak information unduly, users need to be provided with enforcement mechanisms that operate at bytecode level, because Java applets are downloaded as JVM bytecode programs.

Contribution. We define and prove the soundness of an information flow type system for a sequential fragment of the Java Virtual Machine (JVM) with objects, arrays, methods, and exceptions; the type system follows the principles of bytecode verification and thus can be integrated in a standard Java security architecture.

* Work partially supported by IST Project MOBIUS, by the RNTL Castles and by the ACI Sécurité SPOPS.

** Most of this work was performed while at INRIA Sophia Antipolis.

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 125–140, 2007.

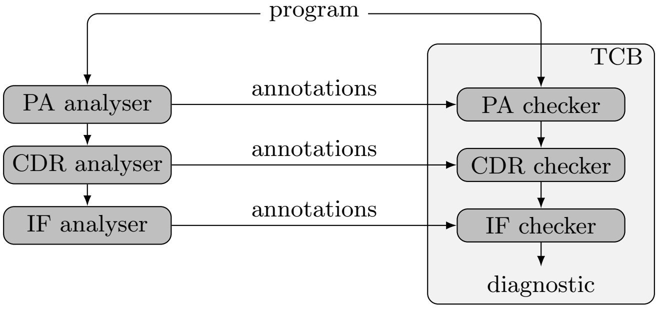


Fig. 1. Information flow analyser and checker

In order to deal with the unstructured nature of bytecode programs, and in particular jumps and exceptions, the analysis is performed in three successive phases, described in the left part of Figure 1:

1. the PA (pre-analyse) analyser computes information that can be used to reduce the control flow graph and to detect branches that will never be taken. The PA analyser performs analyses of null pointers (to predict unthrowable null pointer exceptions), classes (to predict target of throws instructions), array accesses (to predict unthrowable out-of-bounds exceptions), and exceptions (to over-approximate the set of throwable exceptions for each method).
2. the CDR analyser computes control dependence regions (cdr), using the results of the PA analyser to minimise the size of regions. The computations are based on well-known techniques based on post-dominators (see the companion report [5] for details).
3. the IF (Information Flow) analyser uses lightweight bytecode verification techniques, which adapt Kildall's algorithm to compute efficiently for each program point its security environment (i.e. the upper bound of the guards under which it executes) and a stack type that records the security levels of elements of the stack at this program point.

Checking, described on the right part of Figure 1, assumes that programs are annotated with (part of) the results of the PA, CDR, and IF analysers:

1. the PA checker verifies that annotations provided by the PA analyser are correct. Correctness is expressed as as an equivalence between the JVM semantics and an instrumented semantics that manipulate programs annotated with the results of the PA analyser;
2. the CDR checker verifies that regions provided by the CDR analyser verify the safe over-approximation properties (SOAP) of Section 4. Its correctness relies on the correctness of the PA checker;
3. the IF checker verifies type correctness in the style of lightweight bytecode verification. Correctness is proved by showing that typable programs are non-interfering. Its correctness relies on the correctness of the CDR checker and by transitivity on the correctness of the PA checker.

We have formally defined the CDR and IF checkers, and proved their correctness in the Coq proof assistant. The correctness proof assumes that the PA checker is correct; defining and proving the correctness of (parts of) the PA checker in Coq has been done elsewhere [7], and integrating this development in our framework is left for future work.

Related Work. We refer to the survey article of Sabelfeld and Myers [19] for a more complete account of recent developments in language-based security, and only focus on most relevant work.

Java. Jif [15] is an information-flow typed extension of Java that builds upon the decentralised label model to support flexible and expressive information flow policies. Jif offers developers a practical tool for ensuring that applications meet their information flow policies, but lacks a soundness proof. However, Banerjee and Naumann [3,16] have shown the soundness of a simpler information flow type system for a fragment of Java with objects and methods.

Hedin and Sands [12] have observed that most implementations of the Java API invalidate the assumption, common to our work and to [3,15], that references are opaque, i.e. the only observations that an attacker can make about a reference are those about the object to which it points, and exhibited a typable Jif program that unintentionally leaks information through invoking API methods. There are several ways to address this issue, but we leave it for future work.

JVM. The paper improves substantially on our earlier work [6]: the language of this paper is more realistic (it includes methods and arrays and provides an accurate treatment of exceptions), the security policies are more expressive (we adopt arbitrary lattices of security levels instead of two-element lattices), the enforcement mechanism is more accurate (thanks to the PA checker) and simpler (some redundant typing constraints have been removed), and the soundness proof has been machine checked using the proof assistant Coq.

Lanet *et al.* [8] report on a successful use of model-checking techniques to detect illicit information flows in a case study involving Java smart cards. Genaim and Spoto [10] propose another automatic method to check information flow policies for Java bytecode using boolean functions and binary decision diagrams.

Type-preserving compilation. Generalising the results of earlier work with Naumann [4], we have shown that programs typable into an fragment of Jif are compiled into bytecode programs that are accepted by our information flow checker [17]. These results show that (a fragment of) Jif can be used to develop information-flow aware applications that are accepted by our type system. Conversely, they show that applications written in (a fragment of) Jif can be verified automatically at the consumer side by an enhanced bytecode verifier. Zanardini [23] has shown for a fragment of Java including objects and method calls that the compiled counterpart of a source Java program that is accepted by an analyser for abstract non-interference (ANY) [11], also satisfies ANY. This issue has also been studied in the context of typed assembly languages [9,22].

2 Language: Syntax and Semantics

Our information flow type checker is checked correct against Bicolano[1], which formalises the semantics of the JVM in Coq. Bicolano consists of a small step semantics, which captures one-step execution of the JVM and a big step semantics, a small step semantics where method calls are big step (which dispenses from dealing with stack frames and is useful for reasoning); all semantics are proved equivalent in the usual sense. For the purpose of this paper, we have also defined a non-standard semantics on annotated programs, using annotations to eliminate some impossible transitions.

Programs. A program in the JVM is composed of a set of classes. Each class includes a set of fields and a set of methods, including a distinguished method **main** that is the first one to be executed. Each method description includes a method identifier, its code (set of labelled bytecode instructions), a table of exception handlers, and a signature that gives the type of its arguments and of its result.[2] We note $\mathsf{Handler}(i, C) = t$ when there is a handler at program point t for exception of class C thrown at program point i, and $\mathsf{Handler}(i, C) \uparrow$ otherwise. A method identifier may correspond to several methods in the class hierarchy according to overriding of methods. We assume there is a function lookup attached to each program that takes a method identifier and a class name and returns the method to be executed.

Memory Model. The memory model is summarised in Figure 2. During the execution of a method values manipulated by the JVM are either numerical values (taken in a set $\mathcal{N}$), locations (taken in an infinite set $\mathcal{L}$), or simply the *null* constant. Method computation is done on states of the form $\langle h, pc, \rho, s\rangle$ where h is the heap of objects and arrays, pc is the current program point, ρ is the set of local variables and s the operand stack. Heaps are modelled as a partial function $h : \mathcal{L} \rightharpoonup (\mathcal{O} + \mathcal{A})$ from location to objects or arrays. The set $\mathcal{O}$ of objects is modelled as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$, i.e. a class name and a partial function from fields to values. The set $\mathcal{A}$ of arrays is modelled as $\mathcal{N} \times \mathcal{S} \times (\mathcal{N} \rightharpoonup \mathcal{V})$, i.e. each array a handles a length number (noted $a.length$), a security level (noted $at(a)$) and a partial function from index to values (whose accesses are noted $a[i]$). The array security level is a proof artifact useful to keep track of the level attached to every element of an array during allocation. It is straightforward to prove equivalence between executions which manipulate this extra information and those who do not. A set of local variables is a mapping $\rho \in \mathcal{X} \to \mathcal{V}$ from local variables to values. Operand stacks are lists of values. A method execution terminates on *final states.* A final state is either a pair $(v, h) \in \mathcal{V} \times \mathrm{Heap}$ (normal termination), or a pair $(\langle l\rangle, h) \in \mathcal{L} \times \mathrm{Heap}$ (the method execution terminates because of an exception thrown on an object pointed by a location l, but not caught in this method).

[1] `http://mobius.inria.fr/bicolano`

[2] In this abstract, we assume that all methods return a value upon normal termination; however our formalisation also considers void methods.

$\mathcal{N}$: numerical values	$\mathcal{L}$: locations	$\mathcal{X}$: variable names
$\mathcal{C}$: class names	$\mathcal{F}$: field names	$\mathcal{P}$: program points

$\mathcal{V} = \mathcal{N} + \mathcal{L} + \{null\}$	values
$\mathrm{LocalVar} = \mathcal{X} \to \mathcal{V}$	local variables
$\mathrm{OpStack} = \mathcal{V}^*$	operand stacks
$\mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$	objects
$\mathcal{A} = \mathcal{N} \times \mathcal{S} \times (\mathcal{N} \rightharpoonup \mathcal{V})$	arrays
$\mathrm{Heap} = \mathcal{L} \rightharpoonup (\mathcal{O} + \mathcal{A})$	heap
$\mathrm{State} = \mathrm{Heap} \times \mathcal{P} \times \mathrm{LocalVar} \times \mathrm{OpStack}$	states
$\mathrm{FinalState} = (\mathcal{V} + \mathcal{L}) \times \mathrm{Heap}$	final states

Fig. 2. Memory model of the JVM

Operational Semantics. Semantic transitions between consecutive states are modelled by a relation $\leadsto_m^\tau$, parameterised by a tag $\tau \in \{\emptyset\} + \mathcal{C}$ (set noted Tag in the sequel) to describe the nature of the transition ($c \in \mathcal{C}$ for a transition which throws an exception of class c and $\emptyset$ for any other transition). We note $\rho, h \Downarrow_m r, h$ the transitive closure $\langle 1, \rho, \varepsilon, h\rangle(\leadsto_m)^\star r, h$ between an initial state and a final result.

We give in Figure 3 the semantics[3] of some instructions. There are four rules for the virtual call instruction. The first models the case where execution of the callee terminates normally. The location l is used to resolve the virtual call. Thanks to the class of l and the identifier m_{ID}, a method m' is found in the class hierarchy (through the lookup operator). The transitive closure of $\leadsto_m$ is then used to obtain the result of the execution of m'. Execution of m' is initialised with location l for the reserved variable *this* and the elements of the operand stack os_1 for the other variables. The second and the third rules model the cases where execution of the called method terminates by an uncaught exception. In the former rule the thrown exception is caught in method m while in the latter rule it is uncaught and m then terminates abnormally. In both cases, we impose that thrown exception has been statically predicted by the result excAnalysis(m_{ID}) of the exception analysis. The fourth rule corresponds to a null pointer exception thrown because the virtual call was made on a null reference. We note **np** the Java class associated to the null pointer exception. When a native exception **np** is thrown the catching mechanism is model by the function RuntimeExceptionHandling. Each instruction which performs accesses references (like getfield f, putfield f and throw) has similar semantics rules. The fifth rule corresponds to the array store instruction (xastore) where the value v is stored in the array pointed by the location l, at the index number i. The last two rules concern the instruction throw which throws the exception pointed by the reference on top of the stack.

[3] For every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f[x \mapsto v]$ denote the unique function f' s.t. $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$. Further, we let $A^\star$ denote the set of A-stacks for every set A. We use:: to denote the cons and concatenation operations on stacks.

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ \{\mathit{this} \mapsto l, \boldsymbol{x} \mapsto os_1\}, h \Downarrow_{m'} v, h' \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \leadsto_m^{\emptyset} \langle i+1, \rho, v :: os_2, h'\rangle}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \qquad e = \mathsf{class}(h'(l')) \\ \{\mathit{this} \mapsto l, \boldsymbol{x} \mapsto os_1\}, h \Downarrow_{m'} \langle l'\rangle, h' \qquad \mathsf{Handler}_m(i, e) = t \qquad \boxed{e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})} \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \leadsto_m^{e} \langle t, \rho, l' :: \epsilon, h'\rangle}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \qquad e = \mathsf{class}(h'(l')) \\ \{\mathit{this} \mapsto l, \boldsymbol{x} \mapsto os_1\}, h \Downarrow_{m'} \langle l'\rangle, h' \qquad \mathsf{Handler}_m(i, e) \uparrow \qquad \boxed{e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})} \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \leadsto_m^{e} \langle l'\rangle, h'}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad l' = \mathrm{fresh}(h) \qquad \boxed{\mathsf{nullAnalysis}(m, i) \neq \mathit{safe}}}{\langle i, \rho, os_1 :: \mathit{null} :: os_2, h\rangle \leadsto_m^{\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathsf{xastore} \qquad 0 \leq i < h(l).\mathit{length}}{\langle i, \rho, v :: i :: l :: os, h\rangle \leadsto_m^{\emptyset} \langle i+1, \rho, os, h[l \mapsto h(l)[i \mapsto v]]\rangle}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad e = \mathsf{class}(h(l)) \qquad \mathsf{Handler}_m(i, e) = t \qquad \boxed{e \in \mathsf{classAnalysis}(m, i)}}{\langle i, \rho, l :: os, h\rangle \leadsto_m^{e} \langle t, \rho, l :: \epsilon, h\rangle}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad e = \mathsf{class}(h(l)) \qquad \mathsf{Handler}_m(i, e) \uparrow \qquad \boxed{e \in \mathsf{classAnalysis}(m, i)}}{\langle i, \rho, l :: os, h\rangle \leadsto_m^{e} \langle l\rangle, h}$$

with $\mathsf{RuntimeExceptionHandling} : \mathrm{Heap} \times \mathcal{L} \times \mathcal{C} \times \mathcal{PP} \times (\mathcal{X} \rightharpoonup \mathcal{V}) \to \mathrm{State} + (\mathcal{L} \times \mathrm{Heap})$ defined by

$$\mathsf{RuntimeExceptionHandling}(h, l', C, i, \rho) = \begin{cases} \langle t, \rho, l' :: \epsilon, h[l' \mapsto \mathsf{default}(C)]\rangle & \text{if } \mathsf{Handler}_m(i, C) = t \\ \langle l'\rangle, h[l' \mapsto \mathsf{default}(C)] & \text{if } \mathsf{Handler}_m(i, C) \uparrow \end{cases}$$

Fig. 3. Selected semantics rules

In several rules boxed premises represent extra-hypotheses added to the standard JVM semantics thanks to the PA analyser, in the same way that only well-typed states are considered when assuming a program is byte-code verified. It is possible to show that our instrumented semantics coincides with the standard semantics if the PA analysis is safe.

3 Policies

The security policy is expressed at the level of methods and based on the assumption that the attacker can only draw observations on the input/output behaviour of methods. We do not consider the case of executions that hang, nor of "wrong" executions that get stuck—such executions are eliminated by bytecode verification.

The policy is given by a lattice $(\mathcal{S}, \leq, \sqcup, \sqcap)$ of security levels, and:

- a security level k_{obs} that determines the observational capabilities of the attacker. More precisely, the attacker can observe fields, local variables, and return values whose level is at or below k_{obs};
- a global policy $\mathit{ft} : \mathcal{F} \to \mathcal{S}$ that attaches security levels to fields. The global policy is used to determine a notion of equivalence $\sim$ between heaps. Intuitively, two heaps h_1 and h_2 are equivalent if $h_1(l).f = h_2(l).f$ for all locations l and fields f s.t. $\mathit{ft}(f) \leq k_{\mathrm{obs}}$;

- a table of method signatures, that associates to each method identifier[4] and security level (corresponding to the object called) a security signature of the form $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$, where $\boldsymbol{k_v}$ provides the security level of the method local variables, including its arguments[5], k_h is the heap effect of the method, i.e. the lower bound for security levels of fields that are affected during execution of the method, and $\boldsymbol{k_r}$ is a record of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \dots e_n : k_{e_n}\}$, where k_n is the security level of the return value (normal termination) and each e_i is an exception class that might be propagated by the method, associated with a security level k_i.[6] It indicates the level of information than can be learnt by observing if the method terminates by an uncaught exception e_i or by a normal return.

A method is safe w.r.t. a signature $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$ if:

1. two terminating runs of the method with $\sim_{\boldsymbol{k_v}}$-equivalent inputs and equivalent heaps, yield $\sim_{\boldsymbol{k_r}}$-equivalent results and equivalent heaps;
2. the heap effect of the method is greater than k_h, i.e. the method does not perform field updates on fields whose security level is below k_h.

Note that the heap effect does not appear in the statement of non-interference proper but is needed to make a modular analysis. We use the heap effect for virtual calls that occur in a high context in order to enforce that no modification is done on low information during the execution of the called method.

Formally, the observational power of the attacker is defined by various *indistinguishability* relations $\sim^D$ on each different semantic sub-domains D of the JVM memory, see Figure 4; these relations are parameterised by a bijection $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ on (a partial set of) locations in order to model the difference between the allocation history between two states (following Banerjee and Naumann's approach [3]): after a high branching where allocations may occur, objects might be indistinguishable, even if their locations are different during execution. Figure 5 presents the notion of output indistinguishability. In all cases, heaps must be indistinguishable. This definition implies that if indistinguishability outputs are of different nature (like normal value/exception or two exceptions from different classes) the security level of the corresponding exception must be high in the output signature $\boldsymbol{k_r}$. When outputs are of similar nature (two normal values or two exceptions of the same class) they are indistinguishable as soon as the corresponding security level in $\boldsymbol{k_r}$ is low.

Definition 1 (Safe method and program). *A method m is* safe *w.r.t. a policy $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in$*

[4] Associating signatures with method identifier instead of method allows to enforce that overriding of a method preserves its declared security signatures.

[5] I.e. local variables have a fixed security level. Leroy [14] defines a transformation that ensures this property, and shows it enables on-device bytecode verification. Hunt and Sands [13] propose an alternative approach.

[6] In the rest of the paper, we will write $\boldsymbol{k_r}[n]$ instead of k_n and $\boldsymbol{k_r}[e_i]$ instead of k_{e_i}.

relation	definition
$v_1 \sim^{\mathcal{V}}_{\beta} v_2$ where $v_1, v_2 \in \mathcal{V}$	$null \sim^{\mathcal{V}}_{\beta} null \qquad \dfrac{v \in \mathcal{N}}{v \sim^{\mathcal{V}}_{\beta} v} \qquad \dfrac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \sim^{\mathcal{V}}_{\beta} v_2}$
$\rho_1 \sim^{\mathrm{LocalVar}}_{\beta,\boldsymbol{k_v}} \rho_2$ where $\rho_1, \rho_2 \in \mathrm{LocalVar}$	$\forall x \in \mathcal{X},\ \boldsymbol{k_v}(x) \leq k_{\mathrm{obs}} \Rightarrow \rho_1(x) \sim^{\mathcal{V}}_{\beta} \rho_2(x)$
$o_1 \sim^{\mathcal{O}}_{\beta} o_2$ where $o_1, o_2 \in \mathcal{O}$	– $\mathsf{class}(o_1) = \mathsf{class}(o_2)$ – $\forall f \in dom(o_1),\ ft(f) \leq k_{\mathrm{obs}} \Rightarrow o_1(f) \sim^{\mathcal{V}}_{\beta} o_2(f)$
$a_1 \sim^{\mathcal{A}}_{\beta} a_2$ where $a_1, a_2 \in \mathcal{A}$	– $a_1.length = a_2.length$ and $at(a_1) = at(a_2)$ – $\forall i \in [0, a_1.length[,\ at(a_1) \leq k_{\mathrm{obs}} \Rightarrow a_1[i] \sim^{\mathcal{V}}_{\beta} a_2[i]$
$h_1 \sim^{\mathrm{Heap}}_{\beta} h_2$ where $h_1, h_2 \in \mathrm{Heap}$	– β is a bijection between $dom(\beta)$ and $rng(\beta)$ – $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h_2)$ – $\forall l \in dom(\beta),\ h_1(l) \sim^{\mathcal{O}}_{\beta} h_2(\beta(l))$ or $h_1(l) \sim^{\mathcal{A}}_{\beta} h_2(\beta(l))$

Fig. 4. Indistinguishability relations

$$\frac{h_1 \sim_\beta h_2 \quad \boldsymbol{k_r}[n] \leq k_{\mathrm{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{\beta,\boldsymbol{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \quad \boldsymbol{k_r}[\mathsf{class}(h_1(l_1))] \leq k_{\mathrm{obs}} \quad l_1 \sim_\beta l_2}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \quad \boldsymbol{k_r}[\mathsf{class}(h_1(l_1))] \not\leq k_{\mathrm{obs}}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (v_2, h_2)} \qquad \frac{h_1 \sim_\beta h_2 \quad \boldsymbol{k_r}[\mathsf{class}(h_2(l_2))] \not\leq k_{\mathrm{obs}}}{(v_1, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \quad \boldsymbol{k_r}[\mathsf{class}(h_1(l_1))] \not\leq k_{\mathrm{obs}} \quad \boldsymbol{k_r}[\mathsf{class}(h_1(l_1))] \not\leq k_{\mathrm{obs}}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\boldsymbol{k_r}} (\langle l_2 \rangle, h_2)}$$

Fig. 5. Output indistinguishability

$\mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in$ Heap, $r_1, r_2 \in \mathcal{V}+\mathcal{L}$ such that $\rho_1, h_1 \Downarrow_m r_1, h_1'$, $\rho_2, h_2 \Downarrow_m r_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\boldsymbol{k_v},\beta} \rho_2$:

- non-interference *there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$ and $(r_1, h_1) \sim_{\beta',\boldsymbol{k_r}} (r_2, h_2)$;*
- heap effect safety *for each location $l \in dom(h_1)$ and each fields $f \in \mathcal{F}$ such that $k_h \not\leq ft(f)$, $h_1(l).f = h_1'(l).f$.*

A program is safe *with respect to a table of method signature Γ if for all its method m, m is safe with respect to all policies in $\{\ \Gamma_m[k] \mid k \in \mathcal{S}\ \}$.*

4 Verification of Control Dependence Regions

The CDR checker begins by computing the static flow graph of all methods. In order to treat methods accurately, the flow graph of method m is represented by an indexed successor relation $(\mapsto^\tau_m)_{\tau \in \mathrm{Tag}} \subseteq (\mathcal{PP} \times \mathcal{PP}) + \mathcal{PP}$, where Tag is either an exception class (exceptional flow) or $\emptyset$ (normal flow). We write $i \mapsto^\tau_m j$ (resp. $i \mapsto^\tau_m$) if $(i, j) \in \mapsto^\tau_m$ (resp. $i \in \mapsto^\tau_m$). Furthermore, we say that i is a return point if $i \mapsto^\tau$ for some τ and note $i \mapsto_m j$ for $\exists \tau$, $i \mapsto^\tau_m j$.

The CDR checker retrieves the functions provided by the CDR analyser:

$$\mathit{region}_m : \mathcal{PP} \times \mathrm{Tag} \to \wp(\mathcal{PP}) \qquad \mathit{jun}_m : \mathcal{PP} \times \mathrm{Tag} \rightharpoonup \mathcal{PP}$$

and checks the SOAP[7] properties below in order to guarantee the correctness of the information that they provide:

SOAP1: for all program points i, j, k and tag τ such that $i \mapsto_m j$, $i \mapsto_m^\tau k$ and $j \neq k$ (i is hence a branching point), $k \in \mathit{region}_m(i, \tau)$ or $k = \mathit{jun}_m(i, \tau)$;
SOAP2: for all program points i, j, k and tag τ, if $j \in \mathit{region}_m(i, \tau)$ and $j \mapsto_m k$, then either $k \in \mathit{region}_m(i, \tau)$ or $k = \mathit{jun}_m(i, \tau)$;
SOAP3: for all program points i, j and tag τ, if $j \in \mathit{region}(i, \tau)$ (or $i = j$) and j is a return point then $\mathit{jun}_m(i, \tau)$ is undefined;
SOAP4: for all program points i and tags τ_1, τ_2, if $\mathit{jun}_m(i, \tau_1)$ and $\mathit{jun}_m(i, \tau_2)$ are defined and $\mathit{jun}_m(i, \tau_1) \neq \mathit{jun}_m(i, \tau_2)$ then $\mathit{jun}_m(i, \tau_1) \in \mathit{region}_m(i, \tau_2)$ or $\mathit{jun}_m(i, \tau_2) \in \mathit{region}_m(i, \tau_1)$;
SOAP5: for all program points i, j and tag τ, if $j \in \mathit{region}(i, \tau)$ (or $i = j$) and j is a return point then for all tag τ' such that $\mathit{jun}_m(i, \tau')$ is defined, $\mathit{jun}_m(i, \tau') \in \mathit{region}_m(i, \tau)$.

Junction points uniquely delimit ends of regions. SOAP1 expresses that successors of branching points belongs (or ends) the region associated with the same kind as their successor relation. SOAP2 says that a successor of a point in a region is either still in the same region or at this end. SOAP3 forbids junction points for a region which contains (or start with) a return point. SOAP4 and SOAP5 express properties between regions of a same program point but with different tags. SOAP4 says that if two differently tagged regions end in distinct points, the junction point of one must belong to the region of the other. SOAP5 imposes that the junction point of a region must be within every region which contains (or starts with) a return point and is decorated with a different tag.

5 Type System

The information flow type system is defined as a modular (i.e. method-wise) data flow analysis of an abstract transition relation. Typing is defined relative to the table Γ of method signatures (used to handle method calls) and to the global policy *ft*, to the CDR annotations, to a security environment *se* that assigns security levels to program points (used to avoid implicit flows) and to a current method signature *sgn*.

Typing Rules. The typing rules are designed to prevent information leakage through imposing appropriate constraints; Figure 6 presents some selected typing rules which are commented below. Typing rules are of one of the two forms below,

[7] Safe Over Approximation Property.

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \\ k_e = \bigsqcup_{e \in \mathsf{excAnalysis}(m_{\mathrm{ID}})} \boldsymbol{k'_r}[e] \qquad \forall j \in \mathit{region}(i, \emptyset),\ k \sqcup k_e \leq se(j) \end{array}}{\Gamma, \mathit{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} st_1 :: k :: st_2 \Rightarrow \mathsf{lift}_{k \sqcup k_e}\left((\boldsymbol{k'_r}[n] \sqcup se(i)) :: st_2\right)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \forall j \in \mathit{region}(i, e),\ k \sqcup \boldsymbol{k'_r}[e] \leq se(j) \qquad \mathsf{Handler}(i, e) = t \end{array}}{\Gamma, \mathit{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \boldsymbol{k'_r}[e]) :: \varepsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \boldsymbol{k'_a} \xrightarrow{k'_h} \boldsymbol{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad k \leq \boldsymbol{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \quad k \sqcup \boldsymbol{k'_r}[e] \leq \boldsymbol{k_r}[e] \quad \forall j \in \mathit{region}(i, e),\ k \sqcup \boldsymbol{k'_r}[e] \leq se(j) \quad \mathsf{Handler}(i, e) \uparrow \end{array}}{\Gamma, \mathit{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$$

$$\frac{P[i] = \mathsf{xastore} \qquad k_1 \sqcup k_2 \sqcup k_3 \leq k_e \qquad \forall j \in \mathit{region}(i, \emptyset),\ k_e \leq se(j)}{\Gamma, \mathit{region}, se, \boldsymbol{k_a} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: k_3[k_e] :: st \Rightarrow \mathsf{lift}_{k_e}(st)}$$

Fig. 6. Selected typing rules

where the rule on the left is used for normal intra-method execution, and the rule on the right is used for return instructions:

$$\frac{P[i] = \mathit{ins} \quad \mathit{constraints}}{\Gamma, \mathit{ft}, \mathit{region}, se, \mathit{sgn}, i \vdash^{\tau} st \Rightarrow st'} \qquad \frac{P[i] = \mathit{ins} \quad \mathit{constraints}}{\Gamma, \mathit{ft}, \mathit{region}, se, \mathit{sgn}, i \vdash^{\tau} st \Rightarrow}$$

where $st, st' \in \overline{\mathcal{S}}^{\star}$ are stacks of *extended security levels*, *ins* is an instruction found at point i in program P, and τ is a tag. An *extended security level* is either a standard level $k \in \mathcal{S}$ or a pair of level (k, k_e) (noted $k[k_e]$) to type array references. Here k represents the level of the reference while k_e is the level of the elements in the array. Such a distinction is mandatory to be able to have low arrays of high elements. Tags are useful when several rules deal with a same instruction. Depending on the nature of the rule ($st \Rightarrow st'$ or $st \Rightarrow$) and the tag ($\tau = \emptyset$ or $\tau = e \in \mathcal{C}$) we make a non-ambiguous correspondence between semantic and typing rules.

Virtual call. There are several constraints common to all rules for virtual calls. The constraint $k \leq k'_h$ avoids invocation of methods with low heap effect on high target objects, as invoking two different target objects (in two executions) may lead to different method bodies to be executed (due to method lookup) and thus if the method identifier has a low heap effect ($k_h \leq k_{\mathrm{obs}}$), then the low memory may be modified differently in both executions. The constraint $se(i) \leq k'_h$ prevents implicit flows (low assignment in high regions) during execution of the called method. The constraint $k_h \leq k'_h$ prevents the called method to update fields with a level lower that k_h. It allows to avoid invocation of methods with low effect on the heap by a method with high effect. Finally, constraints $k \leq \boldsymbol{k'_a}[0]$

and $\forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \boldsymbol{k'_a}[i+1]$ link argument levels with formal parameter levels.

In the first typing rule, the next stack type is lifted[8] with level $k \sqcup k_e$ to avoid indirect flows because of null a pointer exception on the current object. The level k_e is greater than all levels of the exceptions that may escape from the called method. If abnormal termination of the called method reveals secret information then k_e is high and the next stack type must be high too. The security level of the return value is $(k'_r[n] \sqcup se(i))$. The level $k'_r[n]$ corresponds to the level of the return value in the context of the called method. $se(i)$ prevents implicit flow on the result after the virtual call.

The second and the third typing rule are parameterised by an exception e that may be caught by the called method. In the second rule, this exception is caught in the current method while in the third it is not. In both rules $k \sqcup \boldsymbol{k'_r}[e]$ gives an upper bound on the information that can be gained by observing if the called method reached the point $i+1$. This level is hence used to constrain $\mathit{region}(i, e)$, the top of the stack when e is caught and the security level $\boldsymbol{k_r}[e]$ when it is not.

Arrays. We only give the rule concerning normal execution of the array store instruction. We require the stored value to have a lower level than those of the array content ($k_1 \leq k_e$). The level k_2 of the index should be lower than k_e to prevent attacker to learn information by observing which part of the array has been modified. In a similar way, the level k_3 of the reference should be lower than k_e to avoid modifying two distinct arrays with observable contents. Several exceptions can occur when performing an array store (due to null pointer reference, out-of-bound access or wrong type assignment) so we lift the stack type with the level k_e and impose a similar constraint on the current region.

Typing Method and Program. The definition of typable method is stated to ensure that runs of typable programs (i.e. programs whose methods are typable against their signatures) verify at each step the constraints imposed by the typing rules, provided they are called with parameters that respect the signature of their main method.

Definition 2 (Typable method and program). *A method m is typable w.r.t. a method signature table Γ, a global field policy ft, a signature sgn and a cdr region_m if there exists a security environment $se : \mathcal{PP} \rightarrow \mathcal{S}$ and a function $S : \mathcal{PP} \rightarrow \overline{\mathcal{S}}^{\star}$ such that $S_1 = \varepsilon$ and for all $i, j \in \mathcal{PP}$, $\tau \in \mathrm{Tag}$:*

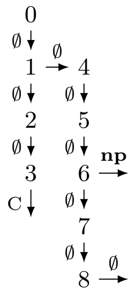
1. *$i \mapsto^{\tau} j$ implies there exists $st \in \overline{\mathcal{S}}^{\star}$ such that $\Gamma, ft, \mathit{region}, se, sgn, i \vdash^{\tau} S_i \Rightarrow st$ and $st \sqsubseteq S_j$;*
2. *$i \mapsto^{\tau}$ implies $\Gamma, ft, \mathit{region}, se, sgn, i \vdash^{\tau} S_i \Rightarrow$*

where $\sqsubseteq$ denotes the point-wise extension of $\leq$ on stack types.

[8] Lifting a stack type with a level k correspond to a map of $\lambda x.k \sqcup x$ on the whole stack. This technique was initially proposed in [6].

```
int m(boolean x,C y) throws C
{
  if (x) {throw new C();}
  else {y.f = 3;};
  return 1;
}
```

```
0 : load x
1 : ifeq 4
2 : new C
3 : throw
4 : load y
5 : push 3
6 : putfield f
7 : const 1
8 : return
```



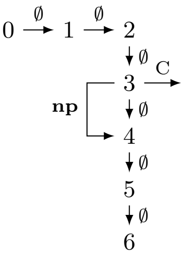
i	$S(i)$	$se(i)$
0	ε	L
1	$L :: \varepsilon$	L
2	ε	L
3	$L :: \varepsilon$	L
4	ε	L
5	$H :: \varepsilon$	L
6	$L :: H :: \varepsilon$	L
7	ε	H
8	$H :: \varepsilon$	H

$$\mathit{region}(1,\emptyset) = \{2,3,4,5,6,7,8\} \quad \mathit{jun}(1,\emptyset) \text{ undef.}$$

$$\mathit{region}(6,\emptyset) = \emptyset \quad \mathit{jun}(6,\emptyset) = 7 \quad \mathit{region}(6,\mathbf{np}) = \{7,8\} \quad \mathit{jun}(6,\mathbf{np}) \text{ undef.}$$

Fig. 7. Typable methods at source and bytecode level

```
0 : load o_L
1 : load y_H
2 : load x_L
3 : invokevirtual m
4 : store z_H
5 : push 1
6 : store t_L
handler : [0, 3], NullPointer → 4
```



i	$S(i)$	$se(i)$
0	ε	L
1	$L :: \varepsilon$	L
2	$L :: L :: \varepsilon$	L
3	$L :: H :: L :: \varepsilon$	L
4	$H :: \varepsilon$	L
5	ε	L
6	$L :: \varepsilon$	L

$$\mathit{region}(3,\emptyset) = \mathit{region}(3,\mathbf{np}) = \emptyset \quad \mathit{jun}(3,\emptyset) = \mathit{jun}(3,\mathbf{np}) = 4$$
$$\mathit{region}(3,C) = \{4,5,6,\ldots\} \quad \mathit{jun}(3,C) \text{ undef.}$$

Fig. 8. Typable fragment with virtual call

A program is typable with respect to a table of method signature Γ, a global field policy ft and a family of cdr $(region_m)_m$ if for all its method m, m is typable with respect to Γ, ft, $region_m$ and all signature in $\{ \Gamma_m[k] \mid k \in \mathcal{S} \}$.

In contrast to [6], types are monovariant, i.e. there is a single stack type per program point. Monovariant analyses are less precise, but remain sufficiently precise for showing type-preserving compilation. Monovariant analyses are more efficient, but harder to prove correct, as several monotonicity results are needed.

Typable Examples. We now give two examples of typable methods. For simplicity, we take as lattice of security levels $\mathcal{S} = \{L, H\}$ with $L \leq H$, where H is the high level for confidential data, and L is the low level for observable data. We note x_k a local variable x whose security level is k.

Figure 7 presents an example of a typable method `m`, giving the corresponding source code and the tagged flow graph. `m` may throw two kinds of exceptions: an exception of class `C` depending on the value of `x`, and an exception of class **np** depending on the values of `x` and `y`. Normal return depends on `y` because execution terminates normally only if it is not *null*. The method `m` is typable with the signature $m : (\mathit{this} : L,\ x : L,\ y : H) \xrightarrow{H} \{n : H,\ C : L,\ \mathbf{np} : H\}$ with the cdr (given only for branching points), the type stacks and the security environment given in Figure 7.

Figure 8 gives another example[9] where fine grain exception handling is necessary for the code to be typable. Here the update $t_L = 1$ at point 6 is accepted if and only if $se(6)$ is low. This fragment is accepted by our type system since, thanks to the fine grain regions, typing rule for virtual call only propagates exception levels $\boldsymbol{k_r}[\mathbf{np}] = H$ in the region $\mathit{region}(3, \mathbf{np})$ (instead of $\mathit{region}(3, C)$).

6 Main Result

We have formalised in Coq several predicates: i) the security condition as SAFE[10]; ii) the correctness of program annotations as PA; iii) the SOAP properties as CDR (given in Section 4); iv) the information flow type checker as IF based on the notion of typable program (Definition 2).

We have machine-checked the following theorem.

Theorem 1. CDR *and* IF *are decidable predicates. Furthermore for every annotated program* P*,*

$$\mathsf{PA}(P) \wedge \mathsf{CDR}(P) \wedge \mathsf{IF}(P) \Longrightarrow \mathsf{SAFE}(P)$$

The first item is proved by formalising boolean-valued functions $\mathsf{check}_{\mathrm{CDR}}$ and $\mathsf{check}_{\mathrm{IF}}$ that characterise the predicates CDR and IF respectively. The function $\mathsf{check}_{\mathrm{CDR}}$ performs a direct verification of the SOAP properties for each method, and the function $\mathsf{check}_{\mathrm{IF}}$ uses lightweight bytecode verification techniques : typability of each method of a program is achieved by traversing the static flow graph and checking for all edges the corresponding typing condition. What is left for future work is to define a decidable predicate $\mathsf{check}_{\mathrm{PA}}$ that entails PA.

The second item is proved in two steps: first, we prove unwinding lemmas and lemmas about security environments. The unwinding lemmas show that one-step execution of typable programs does not reveal secret information. This is formalised using state indistinguishability; indistinguishability between operand stacks is defined relative to stack types S and T, and hence we had to define state indistinguishability relative to stack types. In the sequel, we write $s \sim_{S,T} t$ whenever s and t are equivalent w.r.t. S and T. The unwinding lemmas are of the form (we omit partial bijections and transition tags):

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

[9] To keep the example short here we give compressed version of a compiled code.

[10] Note that SAFE is based on the small-step semantics which acts as reference in Bicolano (without any instrumentation) as defined in Definition 1.

In addition to the unwinding lemmas, we need two lemmas about security environments:

- *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 $s \rightsquigarrow^{\tau} s'$, $t \rightsquigarrow^{\tau'} t'$, $i \vdash^{\tau} S \Rightarrow S'$ and $i \vdash^{\tau'} T \Rightarrow T'$, then S' and T' are high and *se* is high in both region $\mathit{region}(i, \tau)$ and $\mathit{region}(i, \tau')$.
- *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then S' is high.

We then provide a high-level reasoning establishing that a typable program is safe. This part of the proof is not dedicated to a specific fragment of the JVM but applies instead for cdr-based non-interference proofs on low level languages.

7 Remarks on Formal Proofs

The whole Coq development[11] is about 20,000 lines of definitions and proofs; the most important details of the proofs are given in a companion report [5].

The IF checker, and to a lesser extent the CDR checker are complex programs that form the cornerstone of the security architectures that we propose. It is therefore fundamental that their implementation is correct, and therefore their soundness proof should be machine checked. The need for machine-checked proofs is accentuated by the fact that non-interference proofs are particularly involved (w.r.t. say standard type safety proofs discussed in [2]), and that some lemmas as *locally respects* involve two parallel executions leading to an explosion of cases. For example, the JVM virtual call has 5 different transitions (call on a null reference which generates a null pointer exception caught or not, normal termination of the callee, termination by an exception caught or not in the caller context) which required 15 distinct proofs to be exhaustively confronted.

Another motivation for formal proofs is *foundational proof carrying code* or FPCC [1] since the Trusted Computed Base is here relegated to the Coq type checker and the formal definition of non-interference. However, we depart from FPCC in our strategy to prove programs: whereas FPCC uses deductive reasoning to encode proof rules or typing rules, we provide a computational encoding that enables the use of reflective tactics and yields compact certificates. Once we have defined a boolean-valued function $\mathsf{check}_{\mathrm{PA}}$ that entails PA, one can rewrite the main theorem as

$$\mathsf{check}_{\mathrm{PA}}(P) = \mathrm{True} \wedge \mathsf{check}_{\mathrm{CDR}}(P) = \mathrm{True} \wedge \mathsf{check}_{\mathrm{IF}}(P) = \mathrm{True} \Longrightarrow \mathsf{SAFE}(P)$$

Thus the certificate for an annotated program shall be of the form

$$\langle \mathsf{refleq}\ \mathrm{True}, \mathsf{refleq}\ \mathrm{True}, \mathsf{refleq}\ \mathrm{True} \rangle$$

where refleq True is a proof of True = True.

[11] Available on-line at `http://www.irisa.fr/lande/pichardie/iflow`

Agreeingly, much of the certificate is already in the annotations (that are in P), but in comparison with FPCC, we do not have a part of the certificate that encodes deductively the type derivation for P.

Following the approach of *proof carrying proof checkers* [7], it is also possible to extract certified checkers from Coq proofs, which opens up the possibility of safely downloading proof checkers, adding flexibility to the PCC infrastructure.

8 Conclusion

We have developed an information flow type system for a fragment of the JVM that includes objects, methods, exceptions, and arrays, and machine checked its soundness in Coq.

An important goal for future work is to experiment with our type system, by running our verifier on Jif case studies. Unfortunately, most case studies make an intensive use of declassification, which is not provisioned by our type system. Therefore, it seems important to design and machine check type systems that support information release [20]. Another important goal is to extend our results to multi-threaded Java, in order to broaden the scope of applications of our type system; the proposal of Russo and Sabelfeld [18] to control the interactions between threads and the schedulers seems a suitable starting point.

References

1. A.W. Appel and A.P. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *Proceedings of POPL'00*, pages 243–253. ACM Press, 2000.
2. B.E. Aydemir, A. Bohannon, M. Fairbairn, J.N. Foster, B.C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings of TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.
3. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005.
4. G. Barthe, D. Naumann, and T. Rezk. Deriving an Information Flow Checker and Certifying Compiler for Java. In *Symposium on Security and Privacy, 2006*. IEEE Press, 2006.
5. G. Barthe, D. Pichardie, and T. Rezk. Non-interference for low level languages. Technical report, INRIA, 2006. `http://hal.inria.fr/inria-00106182`.
6. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
7. F. Besson, T. Jensen, and D Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3):273-291, 2006.
8. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
9. E. Bonelli, A.B. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In *Proceedings of CASSIS'05*, volume 3956 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2005.

10. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proceedings of VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, 2005.
11. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of POPL'04*, pages 186–197. ACM Press, 2004.
12. D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *Proceedings of CSFW'06*, pages 255–269. IEEE Computer Society Press, 2006.
13. S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *Proceedings of POPL'06*, pages 79–90. ACM Press, 2006.
14. X. Leroy. Bytecode verification on Java smart cards. *Software–practice and experience*, 32(4):319–340, April 2002.
15. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
16. D. Naumann. Verifying a secure information flow analyzer. In *Proceedings of TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, 2005.
17. T. Rezk. *Verification of confidentiality policies for mobile code.* PhD thesis, Université de Nice Sophia-Antipolis, 2006.
18. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of CSFW'06*, 2006.
19. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
20. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press, 2005.
21. 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.
22. D. Yu and N. Islam. A typed assembly language for confidentiality. In P. Sestoft, editor, *Proceedings of ESOP'06*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, 2006.
23. D. Zanardini. *Certified Abstract Non-Interference: Object-Oriented Code Validation for Information Flow Security.* PhD thesis, Università di Verona, April 2006.