Secure Information Flow for a sequential Java Virtual Machine*

Gilles Barthe and Tamara Rezk

INRIA Sophia Antipolis, France
{Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr

Abstract. The Java Platform combines static and dynamic mechanisms to enforce innocuity of applications, but these mechanisms fail to guarantee that confidential information will not leak to unauthorized principals. In order to provide stronger guarantees with respect to confidentiality, it is thus necessary to rely on alternative mechanisms, such as static analyses or type systems that enforce non-interference, a high-level security property that guarantees the absence of illicit information flows during a program execution. While Banerjee and Naumann have recently investigated the use of such type systems for the Java language, there lacks a similar study for the Java Virtual Machine (JVM), and more generally for "low-level languages [that] have not received much attention in studies of secure information flow" (Sabelfeld and Myers, [17]). We propose an information flow type system for a non-trivial fragment of the JVM that includes classes, methods, exceptions and subroutines, and show that it enforces non-interference. Our type system is compatible with the principles of bytecode verification, and could thus be used as the basis of an enhanced bytecode verifier.

1 Introduction

Background The Java and JavaCard platforms feature language-based mechanisms that constrain the computational behavior of applets running on the virtual machine. For example, both platforms include a bytecode verifier that performs a static analysis on programs to guarantee properties such as code containment, type correctness, or initialization. Further, the Java platform relies on a stack inspection mechanism that enforces access control dynamically, whereas the JavaCard platform features a firewall mechanism for the same purpose. However, these mechanisms only provide a partial protection against malicious code. In particular, they fail to provide sufficient guarantees w.r.t. confidentiality, as illustrated in [16] for the JavaCard firewall, and in [1] for Java stack inspection.

Type systems [17] provide an elegant means to guarantee that programs do not leak confidential information, in a sense that can be made precise using the notion of non-interference introduced by Goguen and Meseguer [12]. Initiating from early work by Volpano, Smith and Irvine [20], there have been many

* Work partially supported by IST Projects Profundis and Verificard, and by the ACI Sécurité SPOPS.

works to propose type systems that guarantee confidentiality, both for models of computation and for high-level programming languages. However, "low-level languages have not received much attention in studies of secure information flow" [17]. In particular, there has been no previous attempt to develop type systems for secure information flow in the context of the JVM. In fact, existing works for the JVM use abstract interpretation and model-checking techniques to detect illicit flows, but do not prove non-interference for programs that are accepted by their analysis. Further, their verification mechanism operates on complete programs, and thus departs from standard bytecode verification which operates method-wise.

Our contribution This paper is devoted to the study of secure information flow in a small, but representative, fragment of the JVM that includes classes, methods, exceptions and subroutines. We extend JVM programs with security levels that reflect the status of the information being manipulated, and provide a bytecode verifier which only accepts non-interferent programs that do not yield illegal information flows. Our notion of non-interference accounts for direct flows, and for indirect flows through control flows and run-time exceptions, but does not account for information flows through the observation of termination or execution time, or through system exceptions.

Related work Sabelfeld and Myers [17] provide an excellent, up-to-date, review of the literature on non-interference, and set a number of challenges for further research. We refer to their survey for a more complete overview of related work and only focus on very closely related work.

Non-interference for object-oriented languages is considered by Barthe and Serpette [6] in the context of a simple object calculus, and by Banerjee and Naumann for a non-trivial fragment of Java [2–4]; Strecker [19] proposes a machine-checked proof of the results of Banerjee and Naumann. The JIF prototype, developed by Myers [15], provides an information flow type system for a very rich fragment of Java; however, the complexity of the type system is such that non-interference for (a reasonable fragment of) JIF has not been studied formally. Finally, Darvas, Hähnle, and Sands [10] provide an encoding of secure information flow in dynamic logic, and use the KeY system, and its underlying theorem prover, to verify whether a JavaCard program is non-interferent.

Lanet *et al.*, see e.g. [8], develop a method to detect illicit flows for a sequential fragment of the JVM. In a nutshell, they proceed by specifying in the SMV model checker a symbolic transition semantics of the JVM that manipulates security levels, and by verifying that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their approach has been refined by Bernardeschi and De Francesco, see e.g. [7], for a subset of the JVM that includes jumps, subroutines but no method invocation, exceptions, or instructions for heap manipulation. However, these works do not provide a proof of non-interference. Finally, Barthe, Basu and Rezk [5] provide a type system that guarantees non-interference for a simple assembly language, and define a type-preserving compiler from Volpano and Smith's type system to this assembly language.

Contents The remaining of the paper is organized as follows. In Section 2, we introduce the subset of the JVM under study, and provide its operational semantics. Section 3 is devoted to the definition of non-interference, and Section 4 defines the type system that enforces non-interference. Section 5 provides a formal proof that typable programs are non-interferent. We conclude in Section 6 with directions for further research.

2 The Java Virtual Machine: syntax and semantics

This section introduces the syntax and operational semantics of the fragment of the JVM under study.

2.1 Programs

We focus on a sequential frament of the JVM that encompasses many of its interesting features, including classes, methods, exceptions and subroutines, but does not deal with visibility modifiers, arrays and interfaces, and object initialization (we assume that when a heap object is created, all its fields are automatically initializated with some fixed default value). The set of JVM values is defined as $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is a set of locations. The instruction set we consider is shown in Figure 1.

We use a structured representation of programs, in the form of an environment Δ that includes a set $\mathcal{C}$ of class names with a distinguished class **Object** and a distinguished subclass **Throwable** of **Object**, as well as a set $\mathcal{F}$ of fields. Further, we assume that the environment includes a function $\mathsf{super} : \mathcal{C} \setminus \{\textbf{Object}\} \rightarrow \mathcal{C}$ which provides for every class different from **Object** its (unique) superclass; a function $\mathsf{fields} : \mathcal{C} \rightarrow \wp(\mathcal{F})$ which associates to every class its set of fields, including the fields inherited from its superclasses [1], and a function $\mathsf{methods} : \mathcal{C} \rightarrow \wp(\mathcal{M})$ that provides for every class its set of methods, including the methods inherited from its superclasses. For each method m belonging to a class C, we also require the following information:

- the maximum allowed size for the operand stack max_m;
- its set $\mathcal{X}_m$ of local variables. We assume that $\mathcal{X}_m$ contains a special local variable $this$;
- its (non-empty) instruction list P_m. It is often convenient to treat P_m as a map from the set $\mathcal{PP}_m = \{1, \ldots, n\}$ of program points, where n is the length of P_m, to instructions. We let $P_m[i]$ be the ith instruction in P_m;
- its exception table, denoted by E_m, which defines a set of exception handlers protecting points of the method's body. An exception handler $\langle i, j, k\rangle$ transfers control to address k if an exception is thrown in the interval $[i, j)$.

[1] We assume fields inherited from superclasses of a class C have names which are disjoint from the non-inherited fields names of C.

$instr$ ::=	prim op	primitive operation
|	push v	push value on top of stack
|	pop	pop value from top of stack
|	load x	load value of x on stack
|	store x	store top of stack in x
|	ifeq j	conditional jump
|	goto j	unconditional jump
|	new C	create new object in the heap
|	checkcast C	check compatibility of classes
|	instof C	compatibility of classes
|	invoke m_C	method m virtual invocation
|	getfield f_C	load value of field f on stack
|	putfield f_C	store top of stack in field f
|	throw	exception handling
|	jsr j	subroutine call
|	ret x	return from subroutine
|	return	return from method

where:

- op is either a primitive operation $+, \times$, or a comparison operation $<, \leq, =$;
- $v \in \mathcal{V}$, $m \in \mathcal{M}$, $f \in \mathcal{F}$, $C \in \mathcal{C}$, $x \in \bigcup_{m \in \mathcal{M}} \mathcal{X}_m$, $j \in \bigcup_{m \in \mathcal{M}} \mathcal{PP}_m$.

Convention: in JVML programs, method and field references are used in the instructions such as putfield, getfield or invoke in order to determine the field or method name, its type and the class in which it was declared. Overloading is resolved at compile time. We use in the instructions f_C to refer to field f declared in class C and m_C to refer to method m declared in class C. We omit the subscript C when it is not needed.

Fig. 1. Instruction set

We assume that programs satisfy standard JVM constraints, e.g. that for any two handlers $\langle i, j, k\rangle$ and $\langle i', j', k'\rangle$, either $[i, j) \subseteq [i', j')$, or $[i', j') \subseteq [i, j)$, or $[i', j') \cap [i, j) = \emptyset$. Such assumptions are used e.g. to define partial function $\mathsf{CorrectHandler}_m : \mathcal{PP}_m \rightharpoonup E_m$. Moreover, we write i handled in m if $\mathsf{CorrectHandler}_m(i)$ is defined, and i not handled in m otherwise.

Further, we impose inessential restrictions on programs, that help simplifying the presentation: in particular, we assume that all methods return a value and that local variables are only used to pass the parameters and target object on method invocation. Following [11], our semantics generates a single kind of exception namely a **Throwable** object. Finally, we also assume that $\mathsf{fields}(\mathbf{Object}) = \mathsf{fields}(\mathbf{Throwable}) = \emptyset$.

2.2 Memory model

In our memory model, the set state of JVM states is defined as the set of pairs $\langle fs, h\rangle$, where fs is a stack, and h is a heap. We assume that the heap is unbounded—it amounts to ignoring system exceptions such as **OutOfMemory** exceptions. The stack represents a dynamic chain of method calls and is modeled as a list of frames. A frame is a tuple of the form $\langle i, \rho, s, m\rangle$, where: i) m is the method that is currently executed; ii) $i \in \mathcal{PP}_m$ is the program counter that points to the next instruction to be executed; iii) $\rho : \mathcal{X}_m \to \mathcal{V}$; iv) s is an operand stack, i.e. a $\mathcal{V}$-stack with maximum length max_m. The heap is modeled as a partial function $h : \mathcal{L} \rightharpoonup \mathcal{O}$ from locations to the set $\mathcal{O}$ of constructions of the form $\langle\!\langle l_i = v_i \rangle\!\rangle_\sigma^{i \in I}$. Such a construction corresponds to an object of class σ, and with fields l_i; we use $\langle\!\langle\rangle\!\rangle_{\mathbf{Throwable}}$ for an object of class **Throwable**. Finally we let $\mathcal{H}$ be the set of heaps.

2.3 Operational semantics

The operational semantics formalizes one-step execution of the JVM, and relies on an allocator function $\mathsf{fresh} : \mathcal{H} \times \mathcal{C} \to \mathcal{L}$, such that $\mathsf{fresh}(h, C) \in \mathcal{L} \setminus dom(h)$. It is convenient for proofs to assume that the allocator is well-typed with respect to a typing function $\mathsf{loctype} : \mathcal{H} \times \mathcal{L} \to \mathcal{C}$, i.e. $\mathsf{loctype}(h, \mathsf{fresh}(h, C)) = C$ for every heap h and location l.

The operational semantics is given by a relation $\leadsto$ between states; its definition is given in Figure 2, , where we write $f \oplus \{x \mapsto v\}$ to denote the unique function f' s.t. $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$. $C \leq D$ is defined iff $C = D$ or D is a super class for C. We let $\leadsto^\star$ denote the transitive closure of $\leadsto$. The definition of $\leadsto$ appeals to an auxiliary relation $\leadsto_{exc}$ that specifies exception handling; this relation is defined in Figure 3 between flagged states, where a flagged state is either a state, or a pair of the form $\langle o \rangle s$, with s a state and o a reference to a throwable object. We use $\leadsto_{exc}^\star$ to denote the transitive closure of $\leadsto_{exc}$. Note that we assume that the register *this* remains unmodified during the execution of a method.

The operational semantics may be used to define an evaluation semantics $\Downarrow \subseteq \mathsf{state} \times (\mathcal{V} + \{\mathbf{Abnormal}\})$. The rules are (observe that our operational semantics does not allow to return from a frame if it is the only frame in the stack):

$$\frac{P_m[i] = \mathsf{return} \qquad s \leadsto^\star \langle\langle i, \rho, v :: os, m\rangle :: \epsilon, h\rangle}{s \Downarrow v}$$

$$\frac{s \leadsto^\star s' \leadsto_{exc} \langle o \rangle s'' \leadsto_{exc}^\star \langle o' \rangle \langle\langle i, \rho, os, m\rangle :: \epsilon, h\rangle \qquad \mathsf{CorrectHandler}_m(i) \uparrow}{s \Downarrow \mathbf{Abnormal}}$$

3 Non-Interference

The purpose of this section is to define the notion of non-interference adopted in this paper, and to illustrate its power on a number of examples. Non-interference is defined relative to annotated programs (or in our case extended program environments) which carry security information about methods, fields, etc.

$$\frac{P_m[i] = \mathsf{push}\ n}{\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, n :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{prim}\ op \qquad op \in \mathbb{O} \qquad n_1\ \underline{op}\ n_2 = n}{\langle\langle i, \rho, n_1 :: n_2 :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, n :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{pop}}{\langle\langle i, \rho, v :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{load}\ x}{\langle\langle i, \rho, s, m\rangle :: fs, h\rangle \leadsto \langle\langle i+1, \rho, \rho(x) :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{store}\ x}{\langle\langle i, \rho, v :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho \oplus \{x \mapsto v\}, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{ifeq}\ j}{\langle\langle i, \rho, 0 :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{ifeq}\ j \qquad n \neq 0}{\langle\langle i, \rho, n :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle j, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{goto}\ j}{\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle j, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{new}\ C' \qquad o = \mathrm{fresh}(h, C')}{\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle(i+1, \rho, o :: s, m\rangle :: \mathsf{fs}, h \oplus \{o \mapsto \{f \mapsto v_0 | f \in \mathsf{fields(C)}\}\}\rangle}$$

$$\frac{P_m[i] = \mathsf{checkcast}\ C' \qquad o \in dom(h) \qquad C' \geq \mathrm{loctype}(o, h)}{\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{instof}\ C' \qquad o \in dom(h) \qquad C' \geq \mathrm{loctype}(o, h)}{\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, 1 :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{instof}\ C' \qquad o \in dom(h) \qquad C' \not\geq \mathrm{loctype}(o, h)}{\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, 0 :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{invoke}\ m'_{C'} \qquad o \in dom(h) \qquad \mathrm{loctype}(o, h) \leq C'}{\langle\langle i, \rho, s_1 :: o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle 1, \{this \mapsto o, \boldsymbol{x} \mapsto s_1\}, \epsilon, m'\rangle :: \langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f_{C'} \qquad o \in dom(h) \qquad h(o)(f) = v \qquad \mathrm{loctype}(o, h) \leq C'}{\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, v :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f_{C'} \qquad o \in dom(h) \qquad \mathrm{loctype}(o, h) \leq C'}{\langle\langle i, \rho, v :: o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle i+1, \rho, s, m\rangle :: \mathsf{fs}, h \oplus \{o \mapsto h.o \oplus \{f \mapsto v\}\}\rangle}$$

$$\frac{P_m[i] = \mathsf{return}}{\langle\langle i, \rho, v :: s, m\rangle :: \langle i', \rho', s', m'\rangle :: \mathsf{fs}, h\rangle \leadsto \langle(i'+1, \rho', v :: s', m') :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{jsr}\ j}{\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle(j, \rho, (i+1) :: s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{P_m[i] = \mathsf{ret}\ x}{\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto \langle\langle \rho(x), \rho, s, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{s \leadsto^{\star}_{exc} s' \qquad s, s' \in \mathsf{state}}{s \leadsto s'}$$

Fig. 2. Operational Semantics

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad o = null \qquad o' = \mathrm{fresh}(h, \mathrm{Throwable})}{\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto_{exc} \langle o'\rangle\langle\langle i, \rho, o :: s, m\rangle :: \mathsf{fs}, h \oplus \{o' \mapsto \langle\langle\rangle\rangle_{\mathbf{Throwable}}\}\rangle}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad o = null \qquad o' = \mathrm{fresh}(h, \mathrm{Throwable})}{\langle\langle i, \rho, v :: o :: s, m\rangle :: \mathsf{fs}, h\rangle \leadsto_{exc} \langle o'\rangle\langle\langle i, \rho, v :: o :: s, m\rangle :: \mathsf{fs}, h \oplus \{o' \mapsto \langle\langle\rangle\rangle_{\mathbf{Throwable}}\}\rangle}$$

$$\frac{P_m[i] = \mathsf{athrow} \qquad \mathrm{loctype}(o, h) \leq \mathsf{Throwable}}{\langle\langle i, \rho, o.s, m\rangle :: \mathsf{fs}, h\rangle \leadsto_{exc} \langle o\rangle\langle i, \rho, o :: \epsilon, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{\mathsf{CorrectHandler}_m(i) = \langle b, e, t\rangle}{\langle o\rangle\langle\langle i, \rho, s, m\rangle :: \mathsf{fs}, h\rangle \leadsto_{exc} \langle\langle t, \rho, o :: \epsilon, m\rangle :: \mathsf{fs}, h\rangle}$$

$$\frac{\mathsf{CorrectHandler}_m(i)\ \uparrow}{\langle o\rangle\langle i, \rho, s, m\rangle :: \langle \mathsf{fs}, h\rangle \leadsto_{exc} \langle o\rangle\langle \mathsf{fs}, h\rangle}$$

Fig. 3. Operational Semantics: exception handling

3.1 Annotated programs

Program environments are extended with security information for fields, methods and local variables. The security information is given in terms of a set $\mathcal{S}$ of security levels; as usual, we only consider two security levels in $\mathcal{S}$: one high level for confidential data, denoted with H, and one low level for observable data, denoted with L, and endorse the set of levels with the partial order $L \leq H$. We use the k as a typical element for security levels. Further, we introduce a set $\mathcal{SS}ig$ of security signatures whose elements are of the form $\boldsymbol{k_1} \xrightarrow{k} k_2$, with $\boldsymbol{k_1}$ a vector of security levels in $\mathcal{S}$, and $k, k_2 \in \mathcal{S}$. Assume that program environments are extended with functions $\mathsf{sftype} : \mathcal{F} \to \mathcal{S}$ that associates a security level to each field, and $\mathsf{smtype} : \mathcal{M} \times \mathcal{S} \rightharpoonup \mathcal{SS}ig$. We write $\mathsf{smtype}_m(k)$ instead of $\mathsf{smtype}(m, k)$. If $m \in \mathcal{M}$ and $\mathsf{smtype}_m(k_1) = \boldsymbol{k_2} \xrightarrow{k} k_3$, then invoking m on a target object security level $\leq k_1$: i) requires that parameters have security levels $\leq \boldsymbol{k_2}$; ii) returns a value with security level $\geq k_3$, if the invocation completes normally; iii) will not create a new object in the heap with a security level $\leq k$, and will not modified in the heap fields with security levels $\leq k$. It is sometimes convenient to write $\mathsf{effect}_m(k_1)$ for k, and to consider the security mapping $\Gamma_m^k : \mathcal{X}_m \to m$ induced by $\mathsf{smtype}_m(k)$.

3.2 A formal definition of non-interference

Non-interference is a high-level security property that provides strong guarantees with respect to confidentiality by requiring that the observable outputs of a computation do not reveal any information about the confidential input that may have been processed during the computation. It may be instantiated to a number of precise mathematical definitions that reflect the characteristics of the execution environment, the model of a potential attacker, and the notion of observable data and/or event. A particularly simple and well understood definition of non-interference, and the focus of this paper, is termination-insensitive non-interference, in which termination is not considered as observable. We refer the reader to [17] for further information on other notions of non-interference, and focus on the precise definition of (termination-insensitive) non-interference in the context of the JVM. Indeed, defining indistinguishability for JVM states is rather intricate. Just to mention two problems, it should be possible for indistinguishability to relate i) operand stacks of different size, to account for high branching instructions; ii) heaps that are not indistinguishable point-wise, as the allocator may provide different locations for indistinguishable heaps. Our solution builds upon [5] for operand stacks, and relies on a partial bijection between low locations for heaps, as adopted e.g. in [2]. Hence indistinguishability is defined relatively to an extended program environment Δ, and a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$. Our starting point is value indistinguishability.

Definition 1 (Value Indistinguishability). *$v \sim_{\beta,l} v'$ is defined by the clauses:*

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

Next we lift indistinguishability to local variable maps and heaps.

Definition 2 (Local variable map and heap indistinguishability).

- *Two local variable maps ρ and ρ' are indistinguishable w.r.t. $\Gamma : \mathcal{X} \to \mathcal{L}$, written $\rho \sim_{\beta,\Gamma} \rho'$, if $(\rho\ x) \sim_{\beta,(\Gamma\ x)} (\rho'\ x)$ for every $x \in \mathcal{X}$.*
- *Two heaps h_1 and h_2 are indistinguishable (w.r.t. β), written $h_1 \sim_\beta h_2$, if $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h_2)$ and for every $l \in dom(\beta)$, if $h_1(l) = \langle\!\langle l_i = v_i \rangle\!\rangle_\sigma^{i \in I}$ then $h_2(\beta(l)) = \langle\!\langle l_i = v'_i \rangle\!\rangle_\sigma^{i \in I}$ and $v_i \sim_{\beta,\mathsf{sftype}(l_i)} v'_i$ for every $i \in I$.*

At this point, it is already possible to define non-interference of programs. In the definition, we assume that "main" is the main method of Δ.

Definition 3. *Δ is* secure*, written $\mathsf{NI}(\Delta)$, if for all $v, v' \in \mathcal{V} \cup \{\mathbf{Abnormal}\}$, and for all initial states $s = \langle\langle 1, \rho, \epsilon, \mathrm{main}\rangle, h\rangle$ and $s' = \langle\langle 1, \rho', \epsilon, \mathrm{main}\rangle, h'\rangle$, and for all partial bijections $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, we have*

$$[h \sim_\beta h' \wedge \rho \sim_{\beta,\Gamma^L_{\mathrm{main}}} \rho' \wedge s \Downarrow v \wedge s' \Downarrow v'] \text{ implies } v = v'$$

We now turn to operand stack indistinguishability. First, one must allow operand stacks of different size to be indistinguishable in order to handle high branching instructions. The intuition is that we require operand stacks to be indistinguishable point-wise on some common top part, and then to be high in the bottom part on which they may not coincide, see Figure 4. High operand stacks are defined relative to a stack type: formally, let s be an operand stack and st be a stack type; we write $\mathsf{high}_{os}\ (s, st)$ if s and st have the same length n and $st[i] = H$ for every $1 \leq i \leq n$.

Definition 4 (Operand stack indistinguishability). *Let s, s' be operand stacks and $st, st' \in \mathcal{ST}$. Then $s \sim_{\beta,st,st'} s'$ is defined inductively as follows:*

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

Finally, we define state indistinguishability. In order to handle methods with high effect, one must allow states with stacks of different size to be indistinguishable. Informally, two states are indistinguishable if their heaps are, and if their stacks are split into a high top part, and a low bottom part, with their low bottom part coinciding point-wise, see Figure 4. The definition is slightly complicated by the fact that one must determine for each frame the security level of the target object. We do so by observing whether *this* occurs in the domain of the partial bijection β indexing the indistinguishability relation. Formally, we rely on a function $\mathsf{level}^{\beta}_{loc} : \mathcal{L} \to \mathcal{S}$ defined by the clauses $\mathsf{level}^{\beta}_{loc}(l) = H$ if $l \notin dom(\beta)$, and as L otherwise. For a frame $f = \langle i, \rho, s, m\rangle$, we define $\mathsf{level}^{\beta}(\rho) = \mathsf{level}^{\beta}_{loc}(\rho(this))$ and $\mathsf{high}_{fr}(f)$ as $\mathsf{level}^{\beta}(\rho) = H$; we also define $\mathsf{Leffect}^{\beta}(f)$ as $\mathsf{effect}_m(\mathsf{level}_{\beta}(\rho)) = L$. Further, let lf be a stack of frames and ST be a stack of stack types. We write $\mathsf{high}_{st}\ (lf, ST)$ if lf and ST have the same length n and $\mathsf{high}_{fr}(lf[i])$ for every $1 \le i \le n$.

Definition 5. *Let $\langle fs, h\rangle, \langle fs', h'\rangle \in \mathsf{state}$, and $ST, ST' \in \mathcal{ST}$.*

- *$\langle fs, h\rangle \overset{aux}{\sim}_{\beta,ST,ST'} \langle fs', h'\rangle$ is defined inductively by the clauses:*
  - *if $h \sim_{\beta} h'$ then $\langle \epsilon, h\rangle \overset{aux}{\sim}_{\beta,\epsilon,\epsilon} \langle \epsilon, h'\rangle$;*
  - *if $\mathsf{Leffect}^{\beta}(f)$ and $\mathsf{loctype}(\rho(this), h) = \mathsf{loctype}(\rho'(this), h')$, then*

$$\frac{\langle fs, h\rangle \overset{aux}{\sim}_{\beta,ST,ST'} \langle fs', h'\rangle \qquad \rho \sim_{\beta,\Gamma_m^{\mathsf{level}_{\beta}(\rho)}} \rho' \qquad s \sim_{\beta,st,st'} s'}{\langle\langle i, \rho, s, m\rangle :: fs, h\rangle \overset{aux}{\sim}_{\beta,st::ST,st'::ST'} \langle\langle i, \rho', s', m\rangle :: fs', h'\rangle}$$

- *$\langle fs, h\rangle \sim_{\beta,ST,ST'} \langle fs', h'\rangle$ is defined inductively by the clause:*[2]

$$\frac{\langle fs, h\rangle \overset{aux}{\sim}_{\beta,ST,ST'} \langle fs', h'\rangle \qquad \mathsf{high}_{st}\ (fs_0, ST_0) \qquad \mathsf{high}_{st}\ (fs'_0, ST'_0)}{\langle fs_0 ++ fs, h\rangle \sim_{\beta,ST_0++ST,ST'_0++ST'} \langle fs'_0 ++ fs', h'\rangle}$$

3.3 Examples of secure and insecure programs

We give representative examples of programs that leak information through direct flows, and indirect due to dynamic binding of methods, and exceptions. For readability, we tag variables and fields with their security level.

[2] We use ++ to denote concatenation.

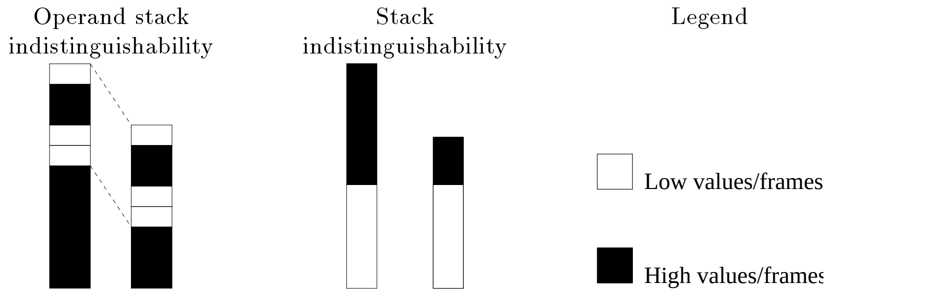

Fig. 4. Operand stack and stack indistinguishability

Example 1. Consider the method m with $\mathsf{smtype}_m(L) = H \xrightarrow{H} L$ and with body:

1 load y_H
2 load y_H
3 putfield f_L
4 push 1
5 return

There is an illegal flow of information from a high variable to a low field. In addition, the assignment at line 3 modifies the heap in a low field, which violates the declaration of the method type.

Example 2. Consider a class C that is a super class of the class D, a method m declared in D, and a method m' declared in D and overridden in C:

I Code of m in D	II Code of m' in C	III Code of m' in D
1 load y_H	1 load $this$	1 load y_H
2 if 6	2 return	2 push 1
3 new C		3 putfield f_L
4 push 0		4 load $this$
5 goto 8		5 return
6 load $this$		
7 load $this$		
8 invoke m'		
9 return		

where $\mathsf{smtype}_m(L) = H \xrightarrow{L} L$ and $\mathsf{smtype}_{m'}(H) = H \xrightarrow{L} H$. At run time, code II or code III is executed depending on the value of high variable y_H. Information about y_H may be inferred by observing the value of field f_L that is changed if code III is executed. Furthermore, information is leaked also through the return value (declared as low) of method m.

Example 3. Consider a method m with exception handler $\langle 5,5,9\rangle$, a method m'' with exception handler $\langle 3,3,5\rangle$, and a method m' without any handler.

Code of m	Code of m'	Code of m''
1 load y_H	1 load y_H	1 load y_H
2 load y_H	2 load $this$	2 load $this$
3 load z_H	3 putfield f_H	3 putfield f_H
4 ifeq 5	4 push 1	4 goto 7
5 invoke m'	5 return	5 push 0
6 goto 12		6 putfield f''_L
7 invoke m''		7 push 1
8 goto 12		8 return
9 push 1		
10 putfield f'_L		
11 push 1		
12 return		

In this example, local variable y_H is passed as parameter to method m'. If the value of y_H at run time is *null* then an exception is thrown and handled in m. The exception handler specifies an assignment to a low field f'_L. An attacker, who can observe the value of f'_L, will detect if variable y_H has value *null* or not according to the value of f'_L after execution of method m. The attacker can also infer information about z_H since the exception may be handled by the handler of method m'', assigning low field f''_L, depending on the value of z_H.

4 Type System

In this section, we define an information flow type system that guarantees non-interference for typable programs. As usual, the type system is described by an abstract transition relation that operates on annotated programs, i.e. programs with extra security annotations, and that is parameterized by an abstract control dependence region that computes the range of branching instruction.

4.1 Static semantics and non-interference

The definition of the information flow type system involves an abstract execution relation $\Rrightarrow$ that describes the result, i.e. abstract state, of executing the instruction at program point p on an abstract state s, and a transfer, or successor, function $\mapsto$ that computes the program points to which control may be transfered after executing the instruction at program point p on an abstract state s. In presence of subroutines, the transfer function cannot be defined as a relation between program points, and abstract states must provide information as to which program point to return upon executing a ret x instruction, i.e. leaving a subroutine. Thus, our information flow type system relies on a address type system. The latter relies on abstract states, which consist of a stack of address types, and of a map that assigns to every register an address type, where the set $\mathcal{R}$ of *address types* is defined as $\{\mathbf{Ret}\ j \mid j \in \mathbb{N}^+\} \cup \{\delta\}$.

Definition 6. *Let m be a method with maximal operand stack size max_m and program points $\mathcal{PP}_m$.*

1. *The set $\mathcal{SR}_m$ of m*-stack address types *is defined as the set of $\mathcal{R}$-stack of length $\leq \mathsf{max}_m$.*
2. *The set $\mathsf{astates}_m$ of m*-address states *is defined as $(\mathcal{PP}_m \to \mathcal{R}) \times \mathcal{SR}_m$.*

The abstract transition relation is defined by clauses of the form

$$\Delta, C, m, i \Vdash \langle \rho, s \rangle \Rrightarrow \langle \rho', s' \rangle \qquad\qquad \Delta, C, m, i \Vdash \langle \rho, s \rangle \Rrightarrow$$

The definition is standard, and omitted—it is straightforward to adapt the presentation from [11] to our setting. The successor relation $\mapsto_m \subseteq \mathsf{astate}_m \times \mathcal{PP}_m \times \mathcal{PP}_m$ of a method m is defined by the clauses (we write $\langle\langle \rho, s \rangle, i\rangle \mapsto_m j$ instead of $\langle\langle \rho, s \rangle, i, j\rangle \in \mapsto_m$):

- if $p_m[i] = \mathsf{goto}\ j$, or $p_m[i] = \mathsf{jsr}\ j$, or both $p_m[i] = \mathsf{ret}\ x$ and $\rho(x) = \mathsf{Ret}\ j$, or both $p_m[i] = \mathsf{throw}$ and $\mathsf{CorrectHandler}_h(i) = \langle b, e, j\rangle$, then $\langle\langle \rho, s \rangle, i\rangle \mapsto_m j$;
- if $p_m[i] = \mathsf{ifeq}\ j$, then $\langle\langle \rho, s \rangle, i\rangle \mapsto_m i+1$ and $\langle\langle \rho, s \rangle, i\rangle \mapsto_m j$;
- if $p_m[i] = \mathsf{putfield}\ f$, or $p_m[i] = \mathsf{getfield}\ f$ then $\langle\langle \rho, s \rangle, i\rangle \mapsto_m i+1$; if furthermore $\mathsf{CorrectHandler}_m(i) = \langle b, e, t\rangle$, then also $\langle\langle \rho, s \rangle, i\rangle \mapsto_m t$;
- if $p_m[i] = \mathsf{return}$, or both $p_m[i] = \mathsf{throw}$ and $\mathsf{CorrectHandler}_m(i) \uparrow$, then $\langle\langle \rho, s \rangle, i\rangle$ has no successors;
- otherwise, $\langle\langle \rho, s \rangle, i\rangle \mapsto_m i+1$.

Further, we write $\langle\langle \rho, s \rangle, i\rangle \Rrightarrow_{\Delta,C,m} \langle\langle \rho', s' \rangle, i'\rangle$ if $\Delta, C, m, i \vdash \langle \rho, s\rangle \Rrightarrow \langle \rho', s'\rangle$ and $\langle \rho, s \rangle, i\rangle \mapsto_m i'$, and let $\Rrightarrow^+_{\Delta,C,m}$ denote the transitive closure of $\Rrightarrow_{\Delta,C,m}$.

Type-checking is performed by a polyvariant analysis as in [9, 13]. Hence our typing rule for methods is of the form $S \Vdash \Delta, C, m$ where Δ is a program environment, C is a class name, m is a method name, and $T \in \mathcal{PP}_m \to \wp(\mathsf{astate}_m)$. The typing rule for methods is

$$\frac{\begin{array}{c}\forall i, j \in \mathcal{PP}_m.\ \forall a \in T_i.\ \langle a, i\rangle \mapsto_m j \text{ implies } \exists a' \in T_j.\ \Delta, C, m, i \Vdash a \Rrightarrow a' \\ \forall i \in \mathcal{PP}_m.\ \forall a \in T_i.\ \langle a, i\rangle \not\mapsto_m \text{ implies } \Delta, C, m, i \Vdash a \Rrightarrow\end{array}}{T \Vdash \Delta, C, m}$$

A program Δ is typable if all its methods are, and we write $\vdash \Delta$.

4.2 Control dependence regions

Our type system is parameterized by control dependence regions. Traditionally, such control dependence regions identify for every conditional branching instruction the set of program points that execute under their condition. In presence of subroutines, it is however not possible to compute control dependence regions from program points, since the control-flow is determined by the return address of subroutines, and the latter is stored in abstract states. Hence our definition of control dependence regions is relative to an abstract state, and a program point. Furthermore, control dependence regions are usually understood as minimal, in the sense that they include the exact set of program points that depend on the condition. For our purposes however, we need not require minimality, and simply require our functions to provide a safe over-approximation of control dependence regions. Formally, we define for every method m its set of conditional control points as

$$\mathcal{PP}^{\sharp}_m = \{i \in \mathcal{PP}_m \mid P_m[i] = \mathsf{ifeq}\ j \vee P_m[i] = \mathsf{putfield}\ f \vee P_m[i] = \mathsf{getfield}\ f\}$$

and assume given functions

$$\begin{aligned} \mathsf{region}_m &: \mathsf{astate}_m \times \mathcal{PP}^{\sharp}_m \to \wp(\mathcal{PP}_m) \\ \mathsf{jun}_m &: \mathsf{astate}_m \times \mathcal{PP}^{\sharp}_m \rightharpoonup \mathcal{PP}_m \end{aligned}$$

that respectively compute the control dependence region and the junction point, and the abstract junction point of an instruction. Furthermore, we assume the functions to satisfy some minimal properties, which are required in the proof of non-interference.

Property 1 (Safe Over Approximation Property (SOAP)). Let $i \in \mathcal{PP}^{\sharp}_m$.

- If $\langle a, i\rangle \mapsto i'$ then $i' \in \mathsf{region}_m(a, i)$;
- If $\langle a, i\rangle \Rrightarrow^{+} \langle a', i'\rangle \Rrightarrow \langle a'', i''\rangle$, and $i' \in \mathsf{region}_m(a, i)$, then $i'' \in \mathsf{region}_m(a, i)$ or $i'' = \mathsf{jun}_m(a, i)$;
- If $\langle a, i\rangle \Rrightarrow^{+} \langle a', i'\rangle$, and $i' \in \mathcal{PP}^{\sharp}_m \cap \mathsf{region}_m(a, i)$ then $\mathsf{region}_m(a', i') \subseteq \mathsf{region}_m(a, i)$;
- If $\mathsf{jun}_m(a, i)$ is defined, then $\mathsf{jun}_m(a, i) \notin \mathsf{region}_m(a, i)$.

Note that it is always possible to find functions region_m and jun_m that satisfy the SOAP property, simply by defining the region of an abstract state as the whole set of all program points of the method. While such a choice is possible, it is not judicious since the precision of the type system directly depends on the precision of these functions.

4.3 Abstract transition relation

The type system is defined through an abstract transition system, and is parameterized by the region analysis. The transition system manipulates pairs $\langle st, se\rangle$ where: st is a stack type that records the security level of all values in the operand stack, and se is a security environment which provides for each program point an upper bound for the security level of the regions in which it is included.

Definition 7. *Let m be a method with maximal operand stack size max_m and program points $\mathcal{PP}_m$.*

1. *The set $\mathcal{ST}_m$ of m-*stack (security) types *is defined as the set of $\mathcal{S}$-stacks of length $\leq \mathsf{max}_m$.*
2. *The set tstate_m of m-*(security) type states *is defined as $(\mathcal{PP}_m \to \mathcal{S}) \times \mathcal{ST}_m$.*

The abstract transition system is defined by the typing transfer rules in Figure 5. These rules, which capture the transformation of security types information by instructions, are of the form

$$\frac{constraints}{\Delta, C, m, a, i, k \vdash st, se \Rightarrow st', se'} \qquad \frac{constraints}{\Delta, C, m, a, i, k \vdash st, se \Rightarrow}$$

where Δ is a program environment, and m is a method name, and $a \in \mathsf{astate}_m$, and st, se is a type state that determines typing constraints for $\mathcal{PP}_m$, and st', se' is a type state that determines typing constraints for the successors of instructions in the code of method m, and k is the security level of the target object. The transfer rules prevent:

- assigning values to low variables, and modifying low fields in high regions;
- calling a low effect method from a high effect method;
- raising within a high region an exception that escapes this region;
- raising in a high level method an exception that the method does not handle.

4.4 Type inference and type checking

As for the return address type system of Subsection 4.1, we use a polyvariant analysis. Thus our typing rule for methods is of the form $T, S \vdash \Delta, C, m$ where Δ is a program environment, C is a class name, m is a method name, and $T \in \mathcal{PP}_m \to \wp(\mathsf{astate}_m)$, and $S \in (\mathsf{astate}_m \times \mathcal{PP}_m) \to \wp(\mathsf{tstate}_m)$. The typing rule for methods is $T, S \vdash \Delta, C, m$ provided that for $k \in \mathcal{S}$ the following holds:

- $\forall i, j \in \mathcal{PP}_m.\ \forall a \in T_i.\ \forall (st, se) \in S_{\langle a,i\rangle}.\ \langle a, i\rangle \mapsto_m j$ implies $\exists a' \in T_j.\ \exists (st', se') \in S_{\langle a',j\rangle}.\ \Delta, C, m, a, i, k \vdash (st, se) \Rightarrow (st', se')$;
- $\forall i \in \mathcal{PP}_m.\ \forall a \in T_i.\ \forall (st, se) \in S_{\langle a,i\rangle}.\ se(i) \geq \mathsf{effect}_m(k)$;
- $\forall i \in \mathcal{PP}_m.\ \forall a \in T_i.\ \langle a, i\rangle \not\mapsto_m$ implies $\Delta, C, m, a, i, k \vdash a \Rightarrow$.

We say a program environment Δ is typable, written $\vdash \Delta$, when all its methods are typable with some T and S, according to the rules given above. (Note that there are algorithms to compute T and S when it exists, see e.g. [14]).

5 Soundness

Typable programs are non-interferent.

Theorem 1. *If Δ is typable then* $\mathsf{NI}(\Delta)$.

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

Below we present the two lemmas, and prove that they entail non-interference; due to space constraints, the proofs of the lemmas are ommitted.—they will appear in the full version of the paper. Before giving the lemmas, we introduce some notation. Let $s, s_1, s_2 \in \mathsf{state}$. Then:

$$\frac{P_m[i] = \mathsf{push}\ n}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow se(i) :: st, se}$$

$$\frac{P_m[i] = \mathsf{prim}\ op}{\Delta, C, m, a, i, \bar{k} \vdash k_1 :: k_2 :: st, se \Rightarrow k_1 \sqcup k_2 \sqcup se(i) :: st, se}$$

$$\frac{P_m[i] \in \{\mathsf{pop}, \mathsf{checkcast}\ C'\}}{\Delta, C, m, a, i, \bar{k} \vdash k_1 :: st, se \Rightarrow st, se}$$

$$\frac{P_m[i] = \mathsf{load}\ x}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow (\Gamma(x) \sqcup se(i)) :: st, se}$$

$$\frac{P_m[i] = \mathsf{store}\ x \quad \Gamma(x) \geq se(i)}{\Delta, C, m, a, i, \bar{k} \vdash \Gamma(x) :: st, se \Rightarrow st, se}$$

$$\frac{P_m[i] = \mathsf{ifeq}\ j}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow \mathsf{lift}_k(st), \mathsf{lift}_k(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{goto}\ j}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow st, se}$$

$$\frac{P_m[i] = \mathsf{jsr}\ j}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow se(i) :: st, se}$$

$$\frac{P_m[i] = \mathsf{return} \qquad \mathsf{smtype}_m(\bar{k}) = \boldsymbol{k} \mapsto k' \qquad k \leq k' \qquad se(i) = L \sqcup \mathsf{effect}_m(\bar{k})}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{new}\ C'}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow se(i) :: st, se}$$

$$\frac{P_m[i] = \mathsf{instof}\ C'}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow se(i) \sqcup k :: st, se}$$

$$\frac{P_m[i] = \mathsf{invoke}\ m' \qquad \mathsf{smtype}_{m'}(k') = \boldsymbol{k_1} \xrightarrow{k_2} k'_1 \qquad k \leq k' \qquad \boldsymbol{k_1} \geq st' \qquad k \leq se(i) \leq k_2}{\Delta, C, m, a, i, \bar{k} \vdash st' :: k :: st, se \Rightarrow (k'_1 \sqcup se(i)) :: st, se}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad \mathsf{sftype}(f) \geq k_1 \qquad (se(i) \sqcup k_2) = L \qquad i \text{ not handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k_1 :: k_2 :: st, se \Rightarrow \mathsf{lift}_{k_2}(st), \mathsf{lift}_{k_2}(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad \mathsf{sftype}(f) \geq k_1 \sqcup se(i) \sqcup k_2 \qquad i \text{ handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k_1 :: k_2 :: st, se \Rightarrow \mathsf{lift}_{k_2}(st), \mathsf{lift}_{k_2}(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad \mathsf{sftype}(f) \geq k_1 \sqcup se(i) \sqcup k_2 \qquad i \text{ handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k_1 :: k_2 :: st, se \Rightarrow k_2 :: \epsilon, \mathsf{lift}_{k_2}(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad (se(i) \sqcup k) = L \qquad i \text{ not handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow \mathsf{lift}_k((\mathsf{sftype}(f) \sqcup se(i)) :: st), \mathsf{lift}_k(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad i \text{ handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow \mathsf{lift}_k((\mathsf{sftype}(f) \sqcup se(i)) :: st), \mathsf{lift}_k(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad i \text{ handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow k :: \epsilon, \mathsf{lift}_k(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{ret}\ x \qquad se(i) = L \sqcup \mathsf{effect}_m(\bar{k})}{\Delta, C, m, a, i, \bar{k} \vdash st, se \Rightarrow \mathsf{lift}_{\Gamma(x)}(st), \mathsf{lift}_{\bar{k}}(se, \mathsf{region}_m(i, a))}$$

$$\frac{P_m[i] = \mathsf{athrow} \qquad (se(i) \sqcup k) = L \qquad i \text{ not handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow k :: \epsilon, se}$$

$$\frac{P_m[i] = \mathsf{athrow} \qquad i \text{ handled in } m}{\Delta, C, m, a, i, \bar{k} \vdash k :: st, se \Rightarrow k :: \epsilon, se}$$

where $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$

- lift_k is the point-wise extension to stack types of $\lambda l.\ k \sqcup l$;
- $\mathsf{lift}_k(se, R)$ is the point-wise extension to all program points in R of $\lambda l.\ k \sqcup l$.

Fig. 5. Transfer rules for instructions

- $s \cdot \mathsf{pc}$, $s \cdot \mathsf{locals}$, $s \cdot \mathsf{os}$, $s \cdot \mathsf{method}$ and $s \cdot \mathsf{heap}$, are the obvious projectors;
- $\mathsf{samemethod}(s_1, s_2)$ if $s_1 \cdot \mathsf{method} = s_2 \cdot \mathsf{method}$ and

$$\mathsf{loctype}(s_1 \cdot \mathsf{locals}(this), s_1 \cdot \mathsf{heap}) = \mathsf{loctype}(s_2 \cdot \mathsf{locals}(this), s_2 \cdot \mathsf{heap})$$

In the sequel we assume that Δ is typable, i.e. that for each class C and method m in C there exists a return address type $T^{C,m}$ and security type $S^{C,m}$ s.t. $T^{C,m}, S^{C,m} \Vdash \Delta, C, m$. For readability, we drop C in the sequel. Also, we omit subscripts in jun and region.

Lemma 1 (One-Step Noninterference in Low-Level Environments). *Suppose* $\mathsf{samemethod}(s_1, s_2)$ *and* $s_1 \cdot \mathsf{method} = m$. *Suppose* $s_1 \cdot \mathsf{pc} = s_2 \cdot \mathsf{pc} = i$ *and* $se(i) = L$. *Let* $a \in \mathsf{astate}_m$, *and* $(st_1, se), (st_2, se) \in S^m_{a, s_1 \cdot pc}$. *Assume that* $s_1 \leadsto s'_1$, *and* $s_2 \leadsto s'_2$. *If* $s_1 \sim_{\beta, st_1, st_2} s_2$, *then* $s'_1 \sim_{\beta', st'_1, st'_2} s'_2$, *for some* st'_1, st'_2, *and* β' *s.t.* $\beta \subseteq \beta'$, *and* $\mathsf{samemethod}(s'_1, s'_2)$. *Furthermore, one of the following holds:*

1. $s_1 \cdot \mathsf{pc} = s_2 \cdot \mathsf{pc}$, *and there exists* se' *s.t.* $(st'_1, se'), (st'_2, se') \in S^{s'_1 \cdot \mathsf{method}}_{a', s'_1 \cdot \mathsf{pc}}$ *and* $se'(s'_1 \cdot \mathsf{pc}) = L$;
2. $\mathsf{samemethod}(s_1, s'_1)$, *and* $\mathsf{high}_{os}(s'_1 \cdot \mathsf{os}, st'_1)$, *and* $\mathsf{high}_{os}(s'_2 \cdot \mathsf{os}, st'_2)$, *and there exists* se'' *s.t.* $(st'_1, se'') \in S^{s'_1 \cdot \mathsf{method}}_{a_1, s'_1 \cdot \mathsf{pc}}$, *and* $(st'_2, se'') \in S^{s'_2 \cdot \mathsf{method}}_{a_2, s'_2 \cdot \mathsf{pc}}$, *and* $s'_1 \cdot \mathsf{pc} \in \mathsf{region}(a, s_1 \cdot \mathsf{pc})$, *and* $s'_2 \cdot \mathsf{pc} \in \mathsf{region}(a, s_2 \cdot \mathsf{pc})$, *and* $\forall j \in \mathsf{region}(a, i).\ se''(j) = H$ *and* $se'_1(\mathsf{jun}(a, i)) = L$.

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

Lemma 2 (One-Step Noninterference in High-Level Environments). *Let* s *be reachable from some initial state for* $s \cdot \mathsf{method}$. *Suppose* $s \leadsto s'$, *and* $\mathsf{high}_{os}(s \cdot \mathsf{os}, st)$. *Let* $\langle st, se \rangle \in S^{s \cdot \mathsf{method}}_{a, s \cdot pc}$, *and assume* $se(s \cdot \mathsf{pc}) = H$. *Then there exists* $(st', se') \in S^{s' \cdot \mathsf{method}}_{a', s' \cdot \mathsf{pc}}$ *s.t.* $\mathsf{high}_{os}(s' \cdot \mathsf{os}, st')$, *and* $s \sim_{\mathsf{id}, st, st'} s'$. *Furthermore, one of the following holds:*

- $se'(s' \cdot \mathsf{pc}) = H$;
- *there exists* $a \in \mathsf{astate}_{s \cdot \mathsf{method}}$ *and* $j \in \mathcal{PP}_{s \cdot \mathsf{method}}$ *s.t.* $s \cdot \mathsf{pc} \in \mathsf{region}(a, j)$, *and* $s' \cdot \mathsf{pc} = \mathsf{jun}(a, j)$ *and* $se = se'$.

Lemma 3. *Suppose* $s_1 \sim_{\beta, st_1, st'_1} s'_1$, $s_1 \sim_{id, st_1, st_2} s_2$ *and* $s'_1 \sim_{id, st'_1, st'_2} s'_1$, *then* $s_2 \sim_{\beta, st_2, st'_2} s'_2$.

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

$$\begin{array}{l} s_1 \leadsto s_2 \leadsto \ \ldots\ \leadsto s_{n_1} \\ s'_1 \leadsto s'_2 \leadsto \ \ldots\ \leadsto s'_{n_2} \end{array}$$

where

- $s_1 = \langle\langle 1, \rho, \epsilon, \mathrm{main}\rangle :: \epsilon, h_1\rangle$, and $s'_1 = \langle\langle 1, \rho', \epsilon, \mathrm{main}\rangle :: \epsilon, h'_1\rangle$;
- $\rho \sim_{\beta, \Gamma^L_{\mathrm{main}}} \rho'$ and $h_1 \sim_\beta h'_1$;
- $s_{n_1} \cdot \mathsf{method} = s_{n_2} \cdot \mathsf{method} = \mathrm{main}$;

- $s_{n_1} = \langle f :: \epsilon, h_{n_1}\rangle$, and $s'_{n_2} = \langle f' :: \epsilon, h'_{n_2}\rangle$, where f and f' are frames;
- $\nexists s.\ s_{n_1} \leadsto s$ and $\nexists s.\ s'_{n_2} \leadsto s$.

By invoking Lemma 1 as long as it applies, we conclude for some maximal q that $s_{q+1} \sim_{\beta', st'_1 st'_2} s'_{q+1}$ and $\mathsf{samemethod}(s_{q+1}, s'_{q+1})$, for some st'_1, st'_2, and β' s.t. $\beta \subseteq \beta'$. By maximality of q, there are two possible cases to consider:

- $q + 1 = n_1 = n_2$. Again there are two cases to consider:
  - $s_{n_1} \leadsto_{exc} s$. Then $\mathsf{CorrectHandler}_{main}(s_{q+1}) \uparrow$ and the instruction at $P_{main}[s_{q+1} \cdot \mathsf{pc}]$ throws an exception. By the typing rules, the exception is thrown due to a low value on the operand stack of s_{q+1}. First assume that the exception is not propagated from another method. As $s_{q+1} \sim_{\beta', st'_1 st'_2} s'_{q+1}$, this value is indistinguishable from the value in the operand stack of s'_{q+1}. Further, $s_{q+1} \cdot \mathsf{pc} = s'_{q+1} \cdot \mathsf{pc}$ and hence the instructions to be executed in both states coincide. It follows that $s_{n_2} \leadsto_{exc} s'$ and that $s_1 \Downarrow \mathsf{Abnormal}$ and $s'_1 \Downarrow \mathsf{Abnormal}$.
    If the exception is propagated from another method m', the typing rule makes the analysis above applicable to m', and we can conclude as above.
  - $s_{n_1} \not\leadsto_{exc} s$. The reasoning is similar to the previous case.
- $\mathsf{samemethod}(s_1, s_{q+1})$, and $\mathsf{high}(s_{q+1}, st'_1)$ and $\mathsf{high}(s'_{q+1}, st'_2)$ and $(st'_1, se'') \in S^{s_{q+1}\cdot\mathsf{method}}_{a_1, s_{q+1}\cdot\mathsf{pc}}$ and $(st'_2, se'') \in S^{s'_{q+1}\cdot\mathsf{method}}_{a_2, s'_{q+1}\cdot\mathsf{pc}}$ and $s_{q+1}\cdot\mathsf{pc}, s'_{q+1}\cdot\mathsf{pc} \in \mathsf{region}(a_q, s_q)$, and $\forall i \in \mathsf{region}(a_q, s_q).\ se''(i) = H$, and $se''(\mathsf{jun}(a_q, s_q \cdot \mathsf{pc})) = L$.
  We now invoke Lemma 2 as long as it applies, to conclude that there exists some maximal q_1 and q_2 s.t. Lemma 2 cannot be applied any longer on s_{q+q_1} and s_{q+q_2}. By maximality of q_1, there are two cases to consider:
  - $se''(s_{q+q_1} \cdot \mathsf{pc}) = H$. There are again two cases to consider:
    - if $P_{\mathrm{main}}[s_{q+q_1} \cdot \mathsf{pc}] = \mathtt{return}$, which contradicts the hypothesis that the program is typable (by the typing rule of `return` the security level at its address has to be L whenever the method has low effect);
    - if $P_{\mathrm{main}}[s_{q+q_1} \cdot \mathsf{pc}] \neq \mathtt{return}$, there is an exception being thrown in a high security environment, which contradicts the hypothesis that the program is typable, since if $(s_{q+q_1} \cdot \mathsf{pc})$ not handled in $main$, necessarily $se''(s_{q+q_1} \cdot \mathsf{pc}) = L$;
  - $s_{q+q_1} \cdot \mathsf{pc} = \mathsf{jun}(a_q, j)$ for some j and $se''(s_{q+q_1} \cdot \mathsf{pc}) = L$. Then there exists a path $\langle a_q, s_q \cdot \mathsf{pc}\rangle \Rrightarrow \langle a_{q+1}, s_{q+1} \cdot \mathsf{pc}\rangle \ldots \Rrightarrow \langle a_{q+q_1}, s_{q+q_1}\rangle$. Necessarily $s_{q+q_1} \cdot \mathsf{pc} \notin \mathsf{region}(a, s_q)$, as if $s_{q+q_1} \cdot \mathsf{pc} \in \mathsf{region}(a_q, s_q)$, then $se''(s_{q+q_1} \cdot \mathsf{pc}) = H$, which contradicts the hypothesis. Hence by SOAP, $s_{q+q_1} \cdot \mathsf{pc} = \mathsf{jun}(a_q, s_q \cdot \mathsf{pc})$ or there exists some i s.t. $1 < i < q_1$ and $s_{q+i} \cdot \mathsf{pc} = \mathsf{jun}(a_q, s_q \cdot \mathsf{pc})$. In the latter case, $se''(\mathsf{jun}(a, s_1 \cdot \mathsf{pc})) = L$, which contradicts maximality of q_1. Hence $s_{q+q_1} \cdot \mathsf{pc} = \mathsf{jun}(a_q, s_q \cdot \mathsf{pc})$.

  Then, we show that $s'_{q+q_2} \cdot \mathsf{pc} = \mathsf{jun}(a_q, s_q \cdot \mathsf{pc})$ as above.

By Lemma 2 there exists $(st', se'), (st'', se') \in S^{s_{q+q_1}\cdot\mathsf{method}}_{a', s_{q+q_1}\cdot\mathsf{pc}}$, and $s_{q+1} \sim_{\mathsf{id}, st'_1, st'} s_{q+q_1}$ and $s'_{q+1} \sim_{\mathsf{id}, st'_2, st''} s'_{q+q_2}$ and by Lemma 3 $s_{q+q_1} \sim_{\beta', st', st''} s_{q+q_2}$. Thus we can apply Lemma 1 again, and repeat the process until reaching the final states of the reduction sequences.

6 Conclusion

We have devised a type system that guarantees non-interference for the JVM; to our best knowledge, there is no similar result for a realistic low-level language. Our work may be pursued in several directions, for example:

– extend the type system to the full sequential fragment of the JVM, and implement a prototype of an enhanced bytecode verifier that enforces non-interference, and experiment with the type system on smartcard applets;
– improve our type system by considering more precise analyses, e.g. control-flow sensitive analyses, or by integrating mechanisms for access control similar to those introduced in [4] for a fragment of the Java language;
– develop a certifying compiler for non-interference, taking as source language an extension to exceptions of the type system of Banerjee and Naumann, and as target language our type system. As suggested in [5], such a result would allow us to lift the results of Banerjee and Naumann to a calculus with exceptions. Furthermore, it would be extremely interesting to build up on Strecker's work [18, 19] to certify such a certifying compiler.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of NDSS'03*, pages 107–121. Internet Society, 2003.
2. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 2003. Submitted.
3. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.
4. A. Banerjee and D. A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *Proceedings of CSFW'03*. IEEE Computer Society Press, 2003.
5. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proceedings of VMCAI'04*, volume 2xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2004. To appear.
6. G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, 1999.
7. C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15, 2002.
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. A. Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *Proceedings of FTFJP'02*, 2002.
10. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of WITS'03*, 2003.

11. S. N. Freund and J. C. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
12. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society, 1982.
13. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
14. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
15. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
16. M. Oestreicher and K. Krishna. Object Lifetimes in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
17. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
18. M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Proceedings of CADE'02*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2002.
19. M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
20. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, pages 167–187, December 1996.

Proof. By a case analysis on the instruction that is executed. In the sequel, we use tl to denote the tail function. Except when the contrary is explicitly stated, we assume that $s_1 = \langle\langle i, \rho_1, os_1, m\rangle \cdot fs_1, h_1\rangle$ and $s_2 = \langle\langle i, \rho_2, os_2, m\rangle \cdot fs_2, h_2\rangle$. First we analize cases where the both operational steps were not produced by the exceptional rule:

Case: $P_m[i] = \mathsf{load}\ x$.
By the operational semantics,

$$\begin{aligned} s_1' &= \langle\langle i+1, \rho_1, \rho_1(x) :: s_1, m\rangle :: \mathsf{fs}_1, h_1\rangle \\ s_2' &= \langle\langle i+1, \rho_3, \rho_2(x) :: s_2, m\rangle :: \mathsf{fs}_2, h_2\rangle \end{aligned}$$

By the typing transfer rule for load,

$$\begin{aligned} st_1' &= (\Gamma(x) \sqcup se(i)) :: st_1 \\ st_2' &= (\Gamma(x) \sqcup se(i)) :: st_2 \end{aligned}$$

To see $s_1' \sim_{\beta,\langle st_1',\Gamma\rangle\cdot ST_1,\langle st_2',\Gamma''\rangle\cdot ST_1'} s_2'$ we have to check that $\rho_1(x) \sim_{\beta,\Gamma(x)\sqcup se(i)} \rho_2(x)$. Since $se(i) = L$, we have $se(i) \leq \Gamma(x)$ then $\Gamma(x) \sqcup se(i) = \Gamma(x)$ and we are done by hypothesis.

Case: $P_m[i] = \mathsf{store}\ x$. By the typing transfer rule,

$$\begin{aligned} st_1' &= \mathsf{tl}\ st_1 \\ st_2' &= \mathsf{tl}\ st_2 \end{aligned}$$

By the operational semantics,

$$\begin{aligned} s_1' &= \langle\langle i+1, \rho_1 \oplus \{x \mapsto \mathsf{hd}\ os_1\}, \mathsf{tl}\ os_1, m\rangle, h_1\rangle \\ s_2' &= \langle\langle i+1\rangle, \rho_2 \oplus \{x \mapsto \mathsf{hd}\ os_2\}, \mathsf{tl}\ os_2, m\rangle, h_2\rangle \end{aligned}$$

We must prove:

- $\mathsf{tl}\ os_1 \sim_{\beta,st_1',st_2'} \mathsf{tl}\ os_2$, which follows directly from $os_1 \sim_{\beta,st_1,st_2} os_2$ and the definition of $\sim$;
- $\mathsf{hd}\ os_1 \sim_{\beta,\Gamma(x)} \mathsf{hd}\ os_2$: if $\Gamma(x) = H$ then we are done by definition so assume that $\Gamma(x) = L$. From the typing transfer rule, we know that $\Gamma(x) \geq \mathsf{hd}\ st_1$ and $\Gamma(x) \geq \mathsf{hd}\ st_2$, hence $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = L$. As $os_1 \sim_{\beta,st_1,st_2} os_2$, it follows that $\mathsf{hd}\ os_1 \sim_{\beta,L} \mathsf{hd}\ os_2$, it follows that $\mathsf{hd}\ os_1 \sim_{\beta,L} \mathsf{hd}\ os_2$, and we are done.

Case: $P_m[i] = \mathsf{ifeq}\ j$. By the typing transfer rule,

$$\begin{aligned} st_1' &= \mathsf{lift}_{(\mathsf{hd}\ st_1)}(\mathsf{tl}\ st_1) \\ st_2' &= \mathsf{lift}_{(\mathsf{hd}\ st_2)}(\mathsf{tl}\ st_2) \end{aligned}$$

By the operational semantics,

$$\begin{aligned} s_1' &= \langle\langle j_1, \rho_1, \mathsf{tl}\ os_1, m\rangle, h_1\rangle \\ s_2' &= \langle\langle j_2, \rho_2, \mathsf{tl}\ os_2, m\rangle, h_2\rangle \end{aligned}$$

We must prove:

- $\mathsf{tl}\ os_1 \sim_{\beta,st_1',st_2'} \mathsf{tl}\ os_2$, which follows from the fact that for every values v, v' and security levels l, l' s.t. $l \leq l'$, $v \sim_{\beta,l} v'$ implies $v \sim_{\beta,l'} v'$;
- $se = se'$ and $j_1 = j_2$, or $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = H$. Assume that the second disjunct does not holds, i.e. $\mathsf{hd}\ st_1 \neq H$ or $\mathsf{hd}\ st_2 \neq H$. Necessarily $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = L$, by definition of $\sim$, and $\mathsf{hd}\ os_1 = \mathsf{hd}\ os_2$. The result follows.

Case: $P_m[i] = \mathsf{new}\ C'$.

By the operational semantics,

$$\begin{aligned} s_1' &= \langle\langle i+1, \rho_1, o_1 :: os_1, m\rangle :: \mathsf{fs}_1, h_1 \oplus \{o_1 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields(C)}\}\}\rangle \\ s_2' &= \langle\langle i+1, \rho_3, o_2 :: os_2, m\rangle :: \mathsf{fs}_2, h_2 \oplus \{o_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields(C)}\}\}\rangle \end{aligned}$$

where $o_1 = \text{fresh}(h_1, C')$ and $o_2 = \text{fresh}(h_2, C')$. By the typing transfer rule for new,

$$\begin{aligned} st_1' &= L :: st_1 \\ st_2' &= L :: st_2 \end{aligned}$$

Let β' be $\beta' = \beta \cup \{o_1, o_2\}$.

To see $s_1' \sim_{\beta',\langle st_1',\Gamma\rangle\cdot ST_1,\langle st_2',\Gamma''\rangle\cdot ST_1'} s_2'$ we have to check that $o_1 \sim_{\beta',L} o_2$.Given the definition of β' , indistinguishability holds by definition of $\sim$. To check that the two new heaps are indistinguishable is straigthforward by the definition of β' and $\sim$.

Case: $P_m[i] = \mathsf{putfield}\ f$.

As the program is safe and typable, st_1, st_2, os_1 and os_2 are of the form:

$$\begin{aligned} st_1 &= k_1 \cdot k_2 \cdot st_{11} \\ st_2 &= k_1' \cdot k_2' \cdot st_{22} \\ os_1 &= v_1 \cdot o_1 \cdot os_{11} \\ os_2 &= v_2 \cdot o_2 \cdot os_{22} \end{aligned}$$

We consider 2 cases:

Case $k_2 = L$, and by indistinguishability it occurs if and only if $k'_2 = L$.
There are two cases to consider again:

- $o_1 = null$, hence by indistinguishability $o_2 = null$. By the operational semantics, an exception has to be thrown:

$$s'_1 = \langle\langle o'_1\rangle_{\langle exc\rangle}\langle i, \rho_1, os_1, m\rangle :: fs_1, h_1 \oplus\{o'_1 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}\rangle$$
$$s'_2 = \langle\langle o'_2\rangle_{\langle exc\rangle}\langle i, \rho_2, os_2, m\rangle :: fs_2, h_2 \oplus\{o'_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}\rangle$$

where $o'_1 = \text{fresh}(h_1, \mathsf{Throwable})$ and $o'_2 = \text{fresh}(h_2, \mathsf{Throwable})$. By the typing rules for putfield [3], we have the transition:

$$st'_1 = \langle L\rangle st_1$$
$$st'_2 = \langle L\rangle st_2$$

Let $\beta' = \beta \cup \{o'_1, o'_2\}$. Then $h_1 \oplus\{o'_1 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\} \sim_{\beta'}$ $, h_2 \oplus\{o'_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}$.
By hypothesis $s_1 \sim_{\beta, st_1, st_2} s'_1$. It implies $\langle\langle i, \rho_1, os_1, m\rangle :: fs_1, h_1 \oplus\{o'_1 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}\rangle \sim_{\beta', st_1, st_2}$ $\langle\langle i, \rho_2, os_2, m\rangle :: fs_2, h_2 \oplus\{o'_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}\rangle$ and hence, $s'_1 \sim_{\beta', st'_1, st'_2} s'_2$.

- $o_1 \neq null$, hence by indistinguishability $\beta(o_1) = o_2$ and $o_2 \neq null$. By the operational semantics:

$$s'_1 = \langle\langle i+1, \rho_1, os_{11}, m\rangle :: \mathsf{fs}_1, h_1 \oplus\{o_1 \mapsto f \mapsto v_1\}\rangle$$
$$s'_2 = \langle\langle i+1, \rho_2, os_{22}, m\rangle :: \mathsf{fs}_2, h_2 \oplus\{o_2 \mapsto f \mapsto v_2\}\rangle$$

by the typing rules for putfield, we have the transition:

$$st'_1 = st_{11}$$
$$st'_2 = st_{22}$$

$v_1 \sim_{\beta, k_1} v_2$ holds by indistinguishability hypothesis. $\mathsf{sftype}(f) \geq k_1$, that holds by the typing rules, implies $v_1 \sim_{\beta, \mathsf{sftype}(f)} v_2$ holds and hence indistinguishabity w.r.t. β of the modified heaps.

Case $k_2 = H$, and by indistinguishability it occurs if and only if $k'_2 = H$.
Since the program is typable, we know by the typing rule, $\mathsf{sftype}(f) \geq k_2$, and then $\mathsf{sftype}(f) = H$.
There are four cases to consider:

- $o_1 = o_2 = null$. By operational semantics,

$$s'_1 = \langle\langle o'_1\rangle_{\langle exc\rangle}(\langle i, \rho_1, os_1, m\rangle \cdot fs_1), h_1 \oplus\{o'_1 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\rangle\}\rangle$$
$$s'_2 = \langle\langle o'_2\rangle_\langle exc\rangle\rangle(\langle i, \rho_2, os_2, m\rangle \cdot fs_2), h_2 \oplus\{o'_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}$$

where $o'_1 = \text{fresh}(h_1, \mathsf{Throwable})$ and $o'_2 = \text{fresh}(h_2, \mathsf{Throwable})$. By the typing rules,

$$\begin{array}{ll} st'_1 & = \langle H\rangle st''_1 \\ st'_2 & = \langle H\rangle st''_2 \\ \mathsf{high}\ st''_1 & \\ \mathsf{high}\ st''_2 & \end{array}$$

[3] Note that $\text{lift}_L(st)$ is st.

To check $s'_1 \sim_{\beta,\langle<H>st''_1,\Gamma\rangle\cdot ST_1,\langle<H>st''_2,\Gamma\rangle\cdot ST_2} s'_2$, we have to check that $os_1 \sim_{\beta,st''_1,st''_2} os_2$, which is straightforward since $\mathsf{high} st''_1$ and $\mathsf{high} st''_2$.

– $o_1 \neq null$ and $o_2 \neq null$
  This case is analog to the analog case above when $k_2 = L$. The heap indistinguishability is maintained since $\mathsf{sftype}(f) = H$.
– $o_1 \neq null$ and $o_2 = null$
  By the operational semantics,

$$s'_1 = \langle\langle i+1, \rho_1, os_{11}, m\rangle :: \mathsf{fs}_1, h_1 \oplus \{o_1 \mapsto f \mapsto v_1\}\rangle$$
$$s'_2 = \langle < o'_2 >_{\langle exc\rangle} (\langle i, \rho_2, os_2, m\rangle \cdot fs_2), h_2 \oplus \{o'_2 \mapsto \{f \mapsto v_0 | f \in \mathsf{fields}(\text{Throwable})\}\}\rangle$$

  where $o'_2 = \text{fresh}(h_2, \mathsf{Throwable})$. By the typing rules:

$$\begin{array}{ll} st'_1 & = st_{11} \\ st'_2 & = \langle H\rangle st''_2 \\ \mathsf{high}\ st_1 1 & \\ \mathsf{high}\ st''_2 & \end{array}$$

  To prove $s'_1 \sim_{\beta,\langle st_{11},\Gamma\rangle\cdot ST_1,\langle<H>st''_2,\Gamma\rangle\cdot ST_2} s'_2$, we have to check that $o'_2 \notin dom(\beta)$, but this holds because $o'_2 \notin dom(h_2)$ by operational semantics. $\mathsf{high}\ st_1 1$ and $\mathsf{high}\ st''_2$ implies $os_{11} \sim_{\beta,st''_1,st''_2} os_2$ holds. Hence by the operational semantics, field f is modified in the heap, indistinguishability for the modified heap will not be affected, since the field is high.
– $o_1 = null$ and $o_2 \neq null$
  This case is analog to the case above.

The analysis for getfield and throw are similar to this case and do not present further difficulties.

Case: $P_m[i] = \mathsf{invoke}\ (m')$.
The most important thing to notice for this case, is that if the security environment for the invoke instruction is low, then the method code to execute starting from indistinguishable states, is the same. This is important, since execution of different methods, due to dynamic binding, can lead to distinguishable states.
As the program is safe and typable, st_1 and st_2 are of the form:

$$\begin{array}{l} st_1 = st'_1 :: k_1 :: st_{11} \\ st_2 = st'_2 :: k_2 :: st_{22} \\ os_1 = os'_1 :: o_1 :: os_{11} \\ os_2 = os'_2 :: o_2 :: os_{22} \end{array}$$

By operational semantics,

$$s'_1 = \langle\langle 1, \{this \mapsto o_1, \boldsymbol{x} \mapsto s'_1\}, \epsilon, m'\rangle :: \langle i, \rho_1, s_{11}, m\rangle :: \mathsf{fs}_1, h_1\rangle$$
$$s'_2 = \langle\langle 1, \{this \mapsto o_2, \boldsymbol{x} \mapsto s'_2\}, \epsilon, m'\rangle :: \langle i, \rho_2, s_{22}, m\rangle :: \mathsf{fs}_2, h_2\rangle$$

Since the program is typable and hypothesis $se(i) = L$, $k_1 \leq L$, and it implies $k_1 = L$. By indistinguishability it occurs if and only if $k_2 = L$.

Then we have by indistinguishability hypothesis, that $\beta(o_1) = o_2$ and then, $\mathsf{loctype}(h_1, o_1) = \mathsf{loctype}(h_2, o_2)$. It means, that both methods m' we execute, have the same target object.. Supposing that method m' has effect low on the heap, we have to check first the indistinguishability of the mapping of local variables, in order to check state indistinguishability. It follows directly from indistinguishability of operand stack and the fact that $os'_1 \leq \boldsymbol{k_1}$, assuming $\boldsymbol{k_1}$ as the smtype declaration for the method with a target object greater or equal to L. The last thing to check for indistinguishability of the reached states, is indistinguishability of the frames: $\langle i, \rho_1, s_{11}, m\rangle \sim_{\beta, st_{11}, st_{22}} \langle i, \rho_2, s_{22}, m\rangle$, but it follows from the fact that the original frames were indistinguishables.

Case: $P_m[i] = \mathsf{return}$.

Since the security environment is low, we have that the $\mathsf{effect}(\Gamma(this)) = L$. Since the program is typable, it means that the method was called from a method in a low security environment. Hence, indistinguishability hypothesis is one to one w.r.t. frames, since the frames in the stack are all low. We only have to check that the values added to the modified operand stacks (with the return value of the current method before the return) of the frame of the caller methods are indistinguishable. But it follows from the fact the operand stacks of the top frames are indistinguishable.

tenemos que decir, que el programa es typable con un gamma acuerdo al target object

Case: $P_m[i] = \mathsf{instof}\ C'$.

By operational semantics,

$$s'_1 = \langle\langle i+1, \rho_1, n_1 :: \text{tail } os_1, m\rangle :: \mathsf{fs}_1, h_1\rangle$$
$$s'_2 = \langle\langle i+1, \rho_2, n_2 :: \text{tail } os_2, m\rangle :: \mathsf{fs}_2, h_2\rangle$$

and by the typing rules and hypothesis $se(i) = L$,

$$st'_1 = st_1$$
$$st'_2 = st_2$$

If top of st_1 is high, by indistinguishability, top of st_2 is high, and the new operand stacks are indistinguishable. If top of st_1 is low, by indistinguishability, top of st_2 is low. We have to check $n_1 \sim_{\beta,L} n_2$. It follows from the fact that $\beta(\text{top } os_1) = os_2$ and then $\mathsf{loctype}(\text{top } os_1, h_1) = \mathsf{loctype}(\text{top } os_2, h_2)$, so by the operational semantics $n_1 = n_2$.

Case: $P_m[i] = \mathsf{ret}\ x$.

By operational semantics,

$$s'_1 = \langle\langle \rho_1(x), \rho_1, os_1, m\rangle :: \mathsf{fs}_1, h_1\rangle$$
$$s'_2 = \langle\langle \rho_2(x), \rho_2, os_2, m\rangle :: \mathsf{fs}_2, h_2\rangle$$

Indistinguishability is straightforward. By the the typing rules:

$$st_1, se \Rightarrow \text{lift}_{\Gamma(x)}(st_1), \text{lift}_{\Gamma(x)}(se, \mathsf{region}(i, a)$$
$$st_2, se \Rightarrow \text{lift}_{\Gamma(x)}(st_2), \text{lift}_{\Gamma(x)}(se, \mathsf{region}(i, a')$$

For the "furthermore", we have to prove that $\Gamma(x) = L$ implies $\rho_1(x) = \rho_2(x)$ and hence $s'_1\mathsf{pc} = s'_2\mathsf{pc}$. If follow by indistinguishability hypothesis of $\rho_1 \sim_{\beta,\Gamma} \rho_2$.

Case: $s_1 = \langle o_1 \rangle_{\langle exc \rangle}(i, \rho_1, s_1, m) :: \mathsf{fs}_1, h_1$
By indistinguishability $st_1 = \langle k \rangle st_1'$.
There are two cases:
Case: $k = L$. By state indistinguishability $s_2 = \langle o_2 \rangle_{\langle exc \rangle}(i, \rho_2, s_2, m) :: \mathsf{fs}_2, h_2$.
State indistinguishability after one step of execution is straightforward.
Case: $k = H$. By the typing rules, i handled in m and the exception has been thrown in the same method where is going to be handled. By operational semantics, the only instructions that can raised exceptions are putfield f, getfield f and throw. Suppose that the instruction which thrown the exception is in address j. If the instruction was a throw ACA FALTA MOSTRAR ALGO .... necesary , $s_2 = \langle o_2 \rangle_{\langle exc \rangle}(i, \rho_2, s_2, m) :: \mathsf{fs}_2, h_2$ and state indistinguishability after one step of execution is straightforward. If the exception was thrown by putfield or getfield, at point j then by property of regions $\mathsf{CorrectHandler}_m(j)$ is in $\mathsf{region}(j)$. Also by the typing rule of putfield and getfield then $\mathsf{high} st_1$ and by indistinguishability of operand stack $\mathsf{high} st_2$.
It lack to check indistinguishability of states.
.... FALTA
Because the program is ty By property of $\mathsf{CorrectHandler_m(j)}$ is in $\mathsf{region}(j)$.
Case analysis for $s_2 = \langle o_2 \rangle_{\langle exc \rangle}(i, \rho_2, s_2, m) :: \mathsf{fs}_2, h_2$ is analog.

in the return instruction, que tenemos que demostrar si no hay un paso de ejecution por la semantica operational?

Cases for prim op, pop, checkcast C', jsr j and push n are similar to the case of load and do not present further difficulties. Case for goto j is straightforward since states do not change.