A Certified Lightweight Non-Interference Java Bytecode Verifier

Gilles Barthe, David Pichardie, Tamara Rezk

A Certified Lightweight Non-Interference Java Bytecode Verifier

GILLES BARTHE
INRIA Sophia Antipolis, France
DAVID PICHARDIE
IRISA/INRIA Rennes, France
and
TAMARA REZK
INRIA Sophia Antipolis, France

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.

Categories and Subject Descriptors: D.3.1 [**Formal Definitions and Theory**]: Semantics; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms: Languages, Security, Verification

Additional Key Words and Phrases: Non-interference, Program Analysis, Java Virtual Machine

Contents

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

1. INTRODUCTION

Java security. The Java security architecture combines static and dynamic mechanisms to enforce innocuity of applications; in particular, it features a bytecode verifier that guarantees statically safety properties such as the absence of arithmetic on references, and a stack inspection mechanism that performs access control verifications. However, it lacks of appropriate mechanisms to guarantee stronger confidentiality properties: for example, it has been suggested that the Java security model is not sufficient in security-sensitive applications such as smart cards [Girard 1999; Montgomery and Krishna 1999]. One weakness of the model is that it only concentrates on who accesses sensitive information, but not how sensitive information flows through programs.

Language-based security. The goal of language-based security [Sabelfeld and Myers 2003] is to provide enforcement mechanisms for end-to-end security policies that go beyond the basic isolation properties ensured by security models for mobile code. In contrast to security models based on access control, language-based security focuses on information-flow policies that track how sensitive information is propagated during execution.

Starting from the seminal work of Volpano and Smith [Volpano and Smith 1997], type systems have become a prominent approach for a practical enforcement of information flow policies, and recent research has proposed type-based enforcement mechanisms for advanced programming features such as exceptions, objects [Barthe and Serpette 1999; Banerjee and Naumann 2005], interactions [O'Neill et al. 2006], concurrency [Smith and Volpano 1998] and distribution [Mantel and Sabelfeld 2003]. This line of work has culminated in the design and implementation of information flow type systems for programming languages such as Java [Myers 1999] and Caml [Pottier and Simonet 2003].

Our contribution. It is striking to notice that, although mobile code security is the 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 assurances that their code does not leak information unduly, it is definitely preferable for users to be provided with enforcement mechanisms that operate at bytecode level, because Java applets are downloaded as JVM bytecode programs.

The contribution of this paper is the definition and machine-checked soundness proofs of a type system to enforce confidentiality of applications written in a sequential fragment of the Java Virtual Machine, with objects, methods, exceptions, and arrays. The analysis is compatible with bytecode verification and can thus be integrated in a standard Java security architecture, provided class files are suitably extended with appropriate information expressed as security signatures for methods. To our best knowledge, this is the first sound type-based analysis for a such an expressive fragment of the JVM.

Comparison with [Barthe et al. 2007] and [Barthe and Rezk 2005]. In [Barthe and Rezk 2005], the first and third authors introduce a provably secure information flow type system for an object-oriented language with a simple exception mechanism. Our current work adopts many of the ideas and techniques developed there, but we also improve substantially over this work: the operational semantics of the language is more realistic (we provide a treatment of exceptions that is close to that of Java) and both methods and arrays have been incorporated, the security policies are more expressive (we adopt arbitrary lattices of security levels instead of two-element lattices), the enforcement mechanism is more accurate (we rely on preliminary exception analyses to reduce the control flow graph of applications), and the soundness proof has been machine checked using the proof assistant Coq [Coq Development Team 2004].

This paper is an extended version of [Barthe et al. 2007]. The main differences are the incremental presentation of different language fragments, the inclusion of formal proofs, and a more detailed account of control dependence regions, and of related work. We also provide additional examples to illustrate the working of the typing rules.

Notations and conventions. For every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f \oplus \{x \mapsto v\}$ 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 hd and tl and :: and $+\!\!+$ to denote the head and tail and cons and concatenation operations on stacks.

For simplicity, examples throughout the paper take as partial order 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.

Finally, we also make the assumption that all methods return a result; this is a harmless departure from Java, which allows us to avoid duplicating many definitions. This assumption is done here for the sake of presentation, but the formal proofs do consider both the cases of methods returning a result, and methods returning no result.

2. INFORMAL OVERVIEW

The purpose of this section is to provide an informal account of our security condition, to highlight some salient features of our type system, and finally to provide a high level description of the type soundness proof. In order to avoid a profusion of technical details, we ignore exceptions in a first place, and indicate at the end of the section additional issues that arise when they are considered.

2.1 Policies and attacker model

The security policy is based on the assumption that the attacker can only draw observations on the input/output behavior of methods. On the other hand, we adopt a termination insensitive policy which assumes that the attacker is unable to observe non-termination of programs. Formally, the policy is given by a partial order $(\mathcal{S}, \leq)$ of security levels, and:

— a security level k_{obs} that determines the observational capabilities of the attacker. Essentially, the attacker can observe fields, local variables, and return values whose level is below k_{obs};

— a global policy $\mathsf{ft} : \mathcal{F} \to \mathcal{S}$ that attaches security levels to fields (we let $\mathcal{F}$ denote the set of 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. $\mathsf{ft}(f) \leq k_{\text{obs}}$; the formal definition of heap indistinguishability is rather involved and deferred to Section 5;

— local policies for each method (we let $\mathcal{M}$ denote the set of methods). In a setting where exceptions are ignored, local policies are expressed using security signatures of the form $\vec{k_v} \xrightarrow{k_h} k_r$ where $\vec{k_v}$ provides the security levels of the method's local variables (including method's arguments[1]), k_h is the effect of the method on the heap, and k_r is the return signature, i.e. the security level of the return value. The vector $\vec{k_v}$ of security levels is used to determine a notion of indistinguishability $\sim_{\vec{k_v}}$ between arrays of parameters, whereas the return signature is used to define a notion of indistinguishability $\sim_{k_r}$ between return values.

Essentially, a method is safe w.r.t. a signature $\vec{k_v} \xrightarrow{k_h} k_r$ if:

(1) two terminating runs of the method with $\sim_{\vec{k_v}}$-equivalent inputs, i.e. inputs that cannot be distinguished by an attacker, and equivalent heaps, yield $\sim_{k_r}$-equivalent results, i.e. results that cannot be distinguished by the attacker,
(2) the method does not perform field updates on fields whose security level is below k_h—as a consequence, it cannot modify the heap in a way that is observable by an attacker that has access to fields whose security level is below k_h.

Formally, the security condition is expressed relative to the operational semantics of the JVM, which is captured by judgments of the form $h_i, lv \Downarrow_m r, h_f$, meaning that executing the method m with initial heap h_i and parameters lv yields the final heap h_f and the result r.

Then, we say that a method m is *safe* w.r.t. a signature $\vec{k_v} \xrightarrow{k_h} k_r$ if its method body does not perform field updates on fields of level lower than k_h and if it satisfies the following non-interference property: for all heaps h_i, h_f, h'_i, h'_f, arrays of parameters $\vec{a}$ and $\vec{a'}$, and results r and r',

$$\left.\begin{array}{c} h_i, \vec{a} \Downarrow_m r, h_f \\ h'_i, \vec{a'} \Downarrow_m r', h'_f \\ h_i \sim h'_i \\ \vec{a} \sim_{\vec{k_v}} \vec{a'} \end{array}\right\} \Rightarrow \; h_f \sim h'_f \;\wedge\; r \sim_{k_r} r'$$

There are two important underlying choices in this security condition: first, the security condition focuses on input/output behaviors, and so does not consider the case of executions that hang; however, it also does not consider "wrong" executions that get stuck, as such executions are eliminated by bytecode verification. Second, the security condition is defined on methods, and not on programs, as we aim for a modular verification technique in the spirit of bytecode verification.

2.2 Dealing with unstructured programs

Preventing direct flows with stack types.. Any sound information flow type system must prevent direct information leakages that occur through assigning secret values to public variables. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for assignments; in a typical type system for a high-level language [Volpano and Smith 1997], the typing rule for assignments is of the form

$$\frac{\vdash e : \; k \qquad k \leq \vec{k_v}(x)}{\vdash x := e : \; \vec{k_v}(x)}$$

where $\vec{k_v}(x)$ is the security given to variable x by the policy and k is an upper bound of the security level of the variables occurring in the expression e. The constraint $k \leq \vec{k_v}(x)$ ensures that the value stored in x does not depend of any variable whose security level is greater than that of x, and thus that there is no illicit flow to x.

[1] JVM programs use a fragment of their local variables to store parameter values.

In a low level language where intermediate computations are performed with an operand stack, direct information flows are prevented by assigning a security level to each value in the operand stack, via a so-called *stack type*, and by rejecting programs that attempt storing a value in a low variable when the top of the stack type is high:

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow \vec{k_v}(x) :: st} \qquad \frac{P[i] = \mathsf{store}\ x \quad k \leq \vec{k_v}(x)}{i \vdash k :: st \Rightarrow st}$$

where st represents a stack type (a stack of security levels) and $\Rightarrow$ represents a relation between the stack type before execution and the stack type after execution of load.

For instance, $x_L = y_H$ is rejected by any sound information flow type system for a while language, because the constraint $H \leq L$ generated by the typing rule for assignment is violated. Likewise, the low level counterpart

$$\begin{array}{l} \mathsf{load}\ y_H \\ \mathsf{store}\ x_L \end{array}$$

cannot be typed as the typing rule for load forces the top of the stack type to high after executing the instruction, and the typing rule for store generates the constraint $H \leq L$.

Preventing indirect flows via security environments. Any sound information flow type system must also prevent information leakages that occur through the control flow of programs. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for branching statements; in a typical type system for a high-level language [Volpano and Smith 1997], the typing rule for if statements is of the form

$$\frac{\vdash e : \ k \qquad \vdash c_1 : \ k_1 \qquad \vdash c_2 : \ k_2 \qquad k \leq k_1, k_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \ k}$$

and ensures that the write effects of c_1 and c_2 are greater than the guard of the branching statement.

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

— is defined relative to a binary successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ between program points, and a set $\mathcal{PP}_r$ of return points. The successor relation and the set of return points are defined according to the semantics of instructions. Intuitively, j is a successor of i if performing one-step execution from a state whose program point is i may lead to a state whose program point is j. Likewise, j is a return point if it corresponds to a return instruction. In the sequel, we write $i \mapsto$ if $i \in \mathcal{PP}_r$;

— is captured by a function that maps a branching program point i (i.e. a program point with two or more successors) to a set of program points $\mathsf{region}(i)$, called the region of i, and by a partial function that maps branching program points to a junction point $\mathsf{jun}(i)$.

The intuition behind regions and junction points is that $\mathsf{region}(i)$ includes all program points executing under the guard of i and that $\mathsf{jun}(i)$, if it exists is the sole exit from the region of i; in particular, whenever $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. The properties to be satisfied by control dependence

regions, called SOAP properties (*Safe Over Approximation Properties*), are further discussed in next section.

In the type system, we use cdr information in conjunction with a security environment that attaches to each program point a security level, intuitively the upper bound of all the guards under which the program point executes. More precisely, programs are checked against a security environment se and global constraints arise in the type system as side conditions in the typing rules for branching statements. For instance, the rule for ifeq bytecode is of the form:

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \cdots}$$

(In Section 4, we discuss the possible choices for the result stack type in the conclusion.)

In order to prevent indirect flows, the typing rules for instructions with write effect, e.g. store and putfield, must check that the security level of the variable or field to be written is at least as high as the current security environment. For instance, the rule for store becomes:

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

The combination of both rules allows to prevent indirect flows. For instance, the standard example of indirect flow $\texttt{if}\ (\texttt{y}_\texttt{H})\ \{\texttt{x}_\texttt{L} = \texttt{0};\}\ \texttt{else}\ \{\texttt{x}_\texttt{L} = \texttt{1};\}$ is compiled in our low-level language as

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ & \mathsf{ifeq}\ l_1 \\ & \mathsf{push}\ 0 \\ & \mathsf{store}\ x_L \\ & \mathsf{goto}\ l_2 \\ l_1: & \mathsf{push}\ 1 \\ & \mathsf{store}\ x_L \\ l_2: & \ldots \end{array}$$

By requiring that $se(i) \leq \vec{k_v}(x)$ in the store rule and by requiring a global constraint on the security environment in the rule for ifeq, the type system ensures that the above program will be rejected: $se(i)$ must be H if the store instruction is under the influence of a high ifeq, and thus the transition for the store instruction cannot be typed.

2.3 Type system

Our information flow type system adopts the principles of Java bytecode verification, in the sense that it is modular (each method can be verified against its signature in isolation) and that it is defined as a data flow analysis of an abstract transition relation. Formally, the type system is parameterized by:

— a table Γ of method signatures, necessary for typing rules involving method calls;

— a global policy ft that provides the security level of fields;

— a cdr structure $(\mathsf{region}, \mathsf{jun})$ for the method under verification;

— a security environment se;

— a current method signature sgn.

The typing rules are designed to prevent information leakage through imposing appropriate constraints; typing rules are of one of the two forms below, where the

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

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

where $st, st' \in \mathcal{S}^\star$ are stacks of security levels, and *ins* is an instruction found at point i in program P. Typing rules are used to establish a notion of typability. Following Freund and Mitchell [Freund and Mitchell 2003], typability stipulates the existence of a function, that maps program points to stack types, such that each transition is well-typed.

Definition 2.3.1 *Typable method.* A method m is deemed typable w.r.t. a method signature table Γ, a global field policy ft, a signature *sgn* and a cdr $\mathsf{region} : \mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^\star$ such that $S_1 = \varepsilon$ (the operand stack is empty at the initial program point 1), and for all $i, j \in \mathcal{PP}$:

(1) $i \mapsto j$ implies that there exists $st \in \mathcal{S}^\star$ such that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$;

(2) $i \mapsto$ implies that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash S_i \Rightarrow$;

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

The 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.

Typability of a method against its signature can be performed via a dataflow analysis based on Kildall's algorithm [Nielson et al. 1999]. The analysis takes as inputs the local and global policies, the method table, the cdr structure, the security environment, the current signature, and either returns a type $S : \mathcal{PP} \to \mathcal{S}^\star$, or a tag indicating that type-checking has failed.

Assuming that the lattice of security levels satisfy the ascending chain property, i.e. that there is no infinite sequence of security levels

$$k_1 \sqsubset k_2 \sqsubset k_3 \ldots$$

it follows from the monotonicity of the typing rules that the analysis terminates.

We conclude this section by mentioning that there are alternatives to the definition of typable methods, and to verifying typability. One dimension of choice lies in the precision in the analysis: whereas our analysis is monovariant, our earlier work [Barthe et al. 2004] adopted a polyvariant analysis in which types assign to each program point a set of stack types. Polyvariant analyses rely on the finiteness of the set of stack types to guarantee termination. They type more programs, but yield less compact types.

2.4 Proving type soundness

For each of the fragment of the JVM considered in this paper, we adopt a similar strategy to prove soundness of the type system. The proof of soundness is based on some assumptions concerning the cdr information, two unwinding lemmas and two lemmas about preserving high context.

The unwinding lemmas show that execution of typable programs does not reveal secret information. They are stated relative to the small-step semantics $\rightsquigarrow$ and to a notion of state indistinguishability that is defined component-wise, i.e. two states s and t are indistinguishable if and only if their heaps, local variable maps, and

operand stacks are indistinguishable. As shall be explained in Section 4, indistinguishability between operand stacks is defined relative to stack types S and T, and hence we must also defined 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 deal with a program P that come equipped with its method signature table and a particular method m of P that comes equipped with its cdr structure $(\mathsf{region}, \mathsf{jun})$ and security environment se. We say that a type stack $S \in \mathcal{S}^\star$ is high if all levels in S are not lower than k_{obs}. We say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j) \not\leq k_{\text{obs}}$ for all $j \in \mathsf{region}(i)$.

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

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

In addition, we also need additional results that enable to repeatedly apply unwinding lemmas to sequences of execution steps. The first family of results deals with preservation of high contexts.

— *high branching:* if $s \sim_{S,T} t$ with $\mathsf{pc}(s) = \mathsf{pc}(t) = i$ and $\mathsf{pc}(s') \neq \mathsf{pc}(t')$, if $s \rightsquigarrow s'$, $t \rightsquigarrow t'$, $i \vdash S \Rightarrow S'$ and $i \vdash T \Rightarrow T'$, then S' and T' are high and se is high in region $\mathsf{region}(i)$.

— *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash S \Rightarrow S'$, and security environment at program point $\mathsf{pc}(s)$ is high, and S is high, then S' is high.

The second family of results deals with monotonicity of indistinguishability.

— *high stack type sub-typing:* if S' is a high type stack and $S' \sqsubseteq T'$ then T' is high.

— *indistinguishability double monotony:* if $s \sim_{S,T} t$, $S \sqsubseteq U$ and $T \sqsubseteq U$ then $s \sim_{U,U} t$.

— *indistinguishability single monotony:* if $s \sim_{S,T} t$, $S \sqsubseteq S'$ and S is high then $s \sim_{S',T} t$.

The combination of the unwinding lemmas, the high context lemmas, the monotonicity lemmas and the SOAP properties enable to prove that typable programs are non-interfering. The proof will be sketched in the next section. In addition, we prove that the SOAP properties, as well as the typability of programs, can be verified automatically. Summarizing, the main result of the paper is to prove for JVM fragments of increasing complexity, the following:

THEOREM 2.4.1. *Let P be a program in which each method comes equipped with its method signature, it cdr structure* $(\mathsf{region}, \mathsf{jun})$ *and its security environment.*

(*1*) *If all cdr structures satisfy the SOAP properties and all methods are typable with their signature, then each method is safe, and in particular the program is non-interfering (i.e. it main method is safe).*

(*2*) *The SOAP properties can be verified automatically for each method and associated cdr structure.*

(*3*) *The typability of a method against its signature can be verified automatically.*

2.5 Exceptions

Extending the outline of the previous section to exceptions is an important issue, because exceptions are pervasive in Java programs. However, accommodating the Java exception mechanism in information flow analyses raise significant challenges.

First, exceptions introduce several potential sources of information leakage; in particular, attackers may infer sensitive information from the termination mode of

programs. This possibility must be reflected both in the notion of state indistinguishability, and of method signatures, which become significantly more complex. For example, method signatures become of the form

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

with the output signature $\vec{k_r}$ now being of the form

$$\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$$

where k_n is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i.

Second, exceptions have an enormous impact on the control flow graph of programs, since many instructions become branching instructions. Curbing this explosion in the control flow graph is essential for maintaining a minimum of precision in the information flow analysis; therefore, the analysis must be performed in three successive phases:

(1) the PA (pre-analysis) 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 minimize the size of regions. In order to maximize accuracy, regions are defined relative to an exception class. Then, soundness is established relative to an extended set of SOAP properties.
(3) the IF (Information Flow) analyser performs information flow checking, using typing rules that exploit the information from the PA and CDR analyses to eliminate premises about branches that are provably unreachable from the program point being typed.

2.6 Summary of subsequent sections

The subsequent sections analyze in turn increasingly complex fragments of the JVM:

— the machine $\text{JVM}_{\mathcal{I}}$, studied in Section 4, includes basic operations to manipulate operand stacks as well as conditional and unconditional jumps, and is expressive enough for compiling programs written in a simple imperative language. In this section, we define and discuss operand stack indistinguishability. The definitions and type system for $\text{JVM}_{\mathcal{I}}$ are adapted from our earlier work [Barthe et al. 2007];

— the machine $\text{JVM}_{\mathcal{O}}$, studied in Section 5, is an object-oriented extension of $\text{JVM}_{\mathcal{I}}$ which includes features such as dynamic object creation, instance field accesses and updates, and is expressive enough for compiling intra-procedural statements from [Banerjee and Naumann 2005]. In this section, we define and discuss heap indistinguishability. The definitions and type system for $\text{JVM}_{\mathcal{I}}$ are adapted from our earlier work [Barthe and Rezk 2005];

— $\text{JVM}_{\mathcal{C}}$, studied in Section 6, is a procedural extension of $\text{JVM}_{\mathcal{O}}$ with method calls, and is expressive enough to compile the language of [Banerjee and Naumann 2005]. The main difficulty is to handle information leakages caused by dynamic method dispatch;

— $\text{JVM}_{\mathcal{G}}$, studied in Section 7, extends $\text{JVM}_{\mathcal{C}}$ with exceptions. The main difficulty is to handle information leakages caused by exceptions, especially when they escape the scope of the method in which they are raised.

— $\text{JVM}_{\mathcal{A}}$, studied in Section 8, extends $\text{JVM}_{\mathcal{O}}$ with arrays (for readability, we do not deal with exceptions nor methods in the presentation). The main difficulty is to propose a sufficiently fine type system which allows public arrays to handle secret informations.

For each fragment, we shall define programs, states, and semantics. Then we shall formulate the security policy and the typing rules; the unwinding lemmas are given in appendices. We also provide a detailed proof of the unwinding lemmas for the $\text{JVM}_{\mathcal{G}}$ in the appendix.

The final sections of the paper discuss issues that are transversal to the fragments considered: Section 11 provides an overview of related work, whereas Section 9 addresses more specifically the relation with information flow type systems for Java. Section 10 provides additional details on the formal proof developed in Coq.

3. CONTROL DEPENDENCE REGIONS

Our type system relies on the existence of control dependence regions that satisfy the SOAP properties. The purpose of this section is first to present these properties and explain how verify them efficiently. We will then relate the existence of CDRs, as used in this paper, and control dependence regions that are used in compilers. More precisely, we shall show how to define CDRs that comply SOAP using standard compilers techniques.

For the purpose of this section, we abstract away from the syntax of programs and consider given a set $\mathcal{P}$ of program points, a set $\mathcal{P}_r$ of return points, and a successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$, subject to the constraint that for every $i \in \mathcal{P}_r$, there is no $j \in \mathcal{P}$ such that $i \mapsto j$. In a first instance, we do not consider exceptions, i.e. we only consider one kind of return point. The cdr definition will then be extended to exceptions in Section 7. In order to be more precise, we will associate a region for each branching point and exception.

3.1 SOAPs: Safe Over Approximation Properties

Since information flow type checking of programs is performed w.r.t. control dependence regions, we assume that the cdr information comes bundled with the program, and its correctness is verified by a cdr checker that is included in the TCB[2] (see Section 10 for a detailed discussion on TCB).

Thus, we assume that the cdr information is given by functions region and jun. To guarantee the correctness of the information that they provide, these functions should satisfy the set of properties given below. Informally, the properties state that any successor of i either belongs to the region of i, or are equal to $\mathsf{jun}(i)$ (if defined), and $\mathsf{jun}(i)$ is the sole exit to the region of i; in particular if $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$.

Definition 3.1.1. A cdr structure $(\mathsf{region}, \mathsf{jun})$ satisfies the SOAP (Safe Over APproximation) properties if the following properties hold:

SOAP1. for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$;

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

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

In order to provide the reader with some intuition, we provide in Figure 1 examples of regions of two compiled programs, and show that with such definitions, we achieve the desired effect.

[2]Unlike what is claimed in e.g. [Yu and Islam 2006], the cdr information itself is not trusted.

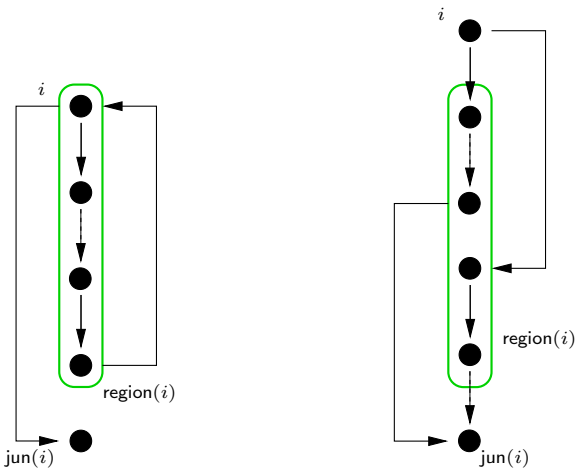

Fig. 1. Example of cdr for a while and an if construct.

Our motivations to bundle the cdr information with programs is that it streamlines the presentation and that it allows us to focus on the information flow analysis itself. However, it is by no means necessary that programs come equipped with their cdr information. In fact, the cdr information can be computed by an analyzer using standard graph algorithms, as it is explained in Section 3.3 but checking cdr can be done very efficiently with a simple algorithm we present now.

3.2 Checking cdr with a linear complexity

We note n the number of program point and b the maximum degree of node in the control flow graph. We suppose the cdr structure satisfies the *intersection property*, that is for all program points i, j such that $\mathsf{region}(i) \cap \mathsf{region}(j) \neq \emptyset$, $\mathsf{region}(i) \subseteq \mathsf{region}(j)$ or $\mathsf{region}(i) \supseteq \mathsf{region}(j)$. This assumption is not necessary to ensure the validity of our algorithm, only to obtain a linear complexity, so we don't need to check if a given cdr respects the intersection pretty. This is a natural property for program obtained by standard compilers.

A cdr structure satisfies the *inclusion property* if for all program points i, j such that $j \in \mathsf{region}(i)$ the following properties hold

IP1 $\mathsf{region}(j) \subseteq \mathsf{region}(i)$;

IP2 if $\mathsf{jun}(j)$ is undefined then $\mathsf{jun}(i)$ is undefined too;

IP3 if $\mathsf{jun}(j)$ is defined then it belongs to $\mathsf{region}(i)$ or is equal to $\mathsf{jun}(i)$.

Verifying if a given cdr respects the inclusion property can be done with a linear complexity if we use an adequate representation of cdr. For this purpose we represent each region $\mathsf{region}(i)$ by giving only the points which belong to $\mathsf{region}(i)$ but all the other regions they belong to contain $\mathsf{region}(i)$. If we note $\mathsf{compressed\_region}(i)$ this set, $\mathsf{region}(i)$ is implicitly defined by

$$\mathsf{region}(i) = \bigcup_{j \in \mathsf{compressed\_region}(i)} \mathsf{region}(j)$$

Using this representation, IP1 is true by construction. To check IP2 and IP3 we examine all points i' in each $\mathsf{compressed\_region}(i)$. If $\mathsf{jun}(i')$ is undefined we check that $\mathsf{jun}(i)$ is undefined too. If $\mathsf{jun}(i')$ is defined, we check it belongs to $\mathsf{region}(i)$ or is equal to $\mathsf{jun}(i)$. Since the cdr verify the intersection property we have necessary

$$\mathsf{compressed\_region}(i) \cap \mathsf{compressed\_region}(j) = \emptyset \qquad \forall i, j \in \mathcal{PP}$$

Hence the sets $(\mathsf{compressed\_region}(i))_{i\in\mathcal{PP}}$ is a partition of the set of program point of a program. The verification of the inclusion property has hence a complexity $\mathcal{O}(n)$.

SOAP1 is checked by enumerating all branching points and checking all their successors. This verification has hence a complexity $\mathcal{O}(b \cdot n)$.

SOAP2 is checked by examining region in increasing order of size. For a given region $\mathsf{region}(i)$ we should at first view examine all points in the region and test if their successors are in the same region or are equal to its junction point. This exhaustive enumeration is superfluous because points that are themselves in a subregion $\mathsf{region}(i')$ of $\mathsf{region}(i)$ doesn't need to be checked. Indeed if j is a point in $\mathsf{region}(i')$ and k is one of its successors, we have necessarily $k \in \mathsf{region}(i')$ or $k = \mathsf{jun}(i')$ since $\mathsf{region}(i')$ is a strict subregion which respects SOAP2. But since we suppose the cdr structure respects the inclusion property, $k \in \mathsf{region}(i)$ or $k = \mathsf{jun}(i)$. As a consequence for each region we only examine points that belongs to the region but not to any other subregions. If the cdr verify the intersection property, we hence globally only check one time each points and the verification has a complexity $\mathcal{O}(n)$.

Finally, SOAP3 is checked in a similar way by examining region in increasing order of size. Again we don't need an exhaustive enumeration because if a subregion $\mathsf{region}(i')$ of a region $\mathsf{region}(i)$, verified SOAP3 then for all return point $j \in \mathsf{region}(i')$, $\mathsf{jun}(i')$ is necessarily undefined and since we suppose the cdr structure respects the inclusion property, $\mathsf{jun}(i)$ is undefined too. We hence globally only check one time each points and the verification has again a complexity $\mathcal{O}(n)$.

3.3 Relating SOAP to the post-dominator notion

Following standard work in compilers, we define that a program point j post-dominates another program point i, written $j \triangleleft i$, if $i \neq j$ and for every return point k, all paths from i to k go through j. Then, we say that j is the junction point of i, written $\mathsf{jun}(i)$, if i is a branching point and j is post-dominated by all post-dominators of i. With such a definition, the junction point is a partial function: for example, a branching point that contains a return statement in one of its branches does not have a junction point.

Finally, we define $\mathsf{region}(i)$ as the set of points that can be reached from i and that are post-dominated by $\mathsf{jun}(i)$, i.e. $j \in \mathsf{region}(i)$ iff $i \mapsto^\star j$ and $\mathsf{jun}(i) \triangleleft j$—in particular, $\mathsf{jun}(i)$ is defined; if not $j \in \mathsf{region}(i)$ iff $i \mapsto^\star j$.

Using the above definitions, it is reasonably easy to prove that the SOAP properties hold.

3.4 Type soundness generic proof technique

We now explain how the combination of the unwinding lemmas, the high context lemmas, the monotonicity lemmas (all presented in Section 2.4) and the SOAP properties enable to prove that typable programs are non-interfering.

In the induction step[3] we have two executions $s_0 \rightsquigarrow \cdots \rightsquigarrow s_n$ and $s'_0 \rightsquigarrow \cdots \rightsquigarrow s'_m$ such that $\mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$ and $s_0 \sim_{S_{\mathsf{pc}(s_0)}, S_{\mathsf{pc}(s'_0)}} s'_0$ and we want to establish that states s_n and s'_m are indistinguishable:

$$s_n \sim_{S_{\mathsf{pc}(s_n)}, S_{\mathsf{pc}(s'_m)}} s'_m$$

or both stack types $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high.

We assume the property holds for any strictly shorter execution paths (induction hypothesis) and suppose $n > 0$ and $m > 0$. We note $i_0 = \mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$. We first remark that by the *locally respects* lemma and typability hypothesis, $s_1 \sim_{st,st'} s'_1$ for some stack types st and st' such that $i_0 \vdash S_{i_0} \Rightarrow st$, $st \sqsubseteq S_{\mathsf{pc}(s_1)}$, $i_0 \vdash S_{i_0} \Rightarrow st'$, $st' \sqsubseteq S_{\mathsf{pc}(s'_1)}$.

[3]Base cases depend on technical properties about return points that we omit in this Section.

instr	::=	binop *op*	binary operation on stack
	|	push c	push value on top of stack
	|	pop	pop value from top of stack
	|	swap	swap the top two operand stack values
	|	load x	load value of x on stack
	|	store x	store top of stack in variable x
	|	ifeq j	conditional jump
	|	goto j	unconditional jump
	|	return	return the top value of the stack

where $op \in \{+, -, \times, /\}$, $c \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{PP}$.

Fig. 2. Instruction set for $\text{JVM}_{\mathcal{I}}$

— If $\mathsf{pc}(s_1) = \mathsf{pc}(s'_1)$ we can apply the *indistinguishability double monotony* lemma to establish that $s_1 \sim_{S_{\mathsf{pc}(s_1)}, S_{\mathsf{pc}(s'_1)}} s'_1$ and conclude by induction hypothesis.

— If $\mathsf{pc}(s_1) \neq \mathsf{pc}(s'_1)$ we know by the *high branching* lemma that *se* is high in region $\mathsf{region}(i_0)$ and *st* and *st*′ are high. Thanks to the *high stack type subtyping* lemma it implies that both $S_{\mathsf{pc}(s_1)}$ and $S_{\mathsf{pc}(s'_1)}$ are high. By SOAP1 we know that $\mathsf{pc}(s_1) \in \mathsf{region}(i_0)$ or $\mathsf{pc}(s_1) = \mathsf{jun}(i_0)$. Now by induction on the path $s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$ we easily show that either there exists k, $1 \leq k \leq n$ such that $k = \mathsf{jun}(i_0)$ and $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{i_0}} s'_0$ (the high path reaches the junction point) or $\mathsf{pc}(s_n) \in \mathsf{region}(i_0)$ and $S_{\mathsf{pc}(s_n)}$ is high (the high path stays in the region). This is proved thanks to SOAP2, *high step* lemma and *indistinguishability single monotony* lemma. Note that in the second case where $\mathsf{pc}(s_n) \in \mathsf{region}(i_0)$, we have necessarily $\mathsf{jun}(i_0)$ undefined by SOAP3. A similar property holds for path $s'_1 \rightsquigarrow \cdots \rightsquigarrow s'_m$ and we can group the make different cases in two main cases:

(1) $\mathsf{jun}(i_0)$ is defined and there exists k, k', $1 \leq k \leq n$ and $1 \leq k' \leq m$ such that $k = k' = \mathsf{jun}(i_0)$ and $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{i_0}} s'_0$ $s_0 \sim_{S_{i_0}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$. Since $s_0 \sim_{S_{i_0}, S_{i_0}} s'_0$ we have by transitivity and symmetry of $\sim$, $s_k \sim_{S_{\mathsf{pc}(s_k)}, S_{\mathsf{pc}(s'_{k'})}} s'_{k'}$ with $\mathsf{pc}(s_k) = \mathsf{pc}(s'_{k'})$ and we can conclude by induction hypothesis.
(2) $\mathsf{jun}(i_0)$ is undefined and both $S_{\mathsf{pc}(s_n)}$ and $S_{\mathsf{pc}(s'_m)}$ are high.

This proof technique can be adapted to the different JVM presented in this article. The details of the $\text{JVM}_{\mathcal{G}}$ type soundness proof are given in appendix.

4. THE $\text{JVM}_{\mathcal{I}}$ SUBMACHINE

In this section, we define an information flow type system for a fragment of the JVM with conditional and unconditional jumps and operations to manipulate the stack.

4.1 Programs, memory model and operational semantics

Programs. In this fragment, a program consists of a single, non-recursive method. Thus we consider that a $\text{JVM}_{\mathcal{I}}$ program P is given by its list of instructions, taken from the instruction set of Figure 2. We let the set $\mathcal{X}$ be the set of local variables and $\mathcal{V}$ the set of values.

States. In this fragment, states do not feature a heap. Thus, the set $\mathsf{State}_{\mathcal{I}}$ of $\text{JVM}_{\mathcal{I}}$ states is defined as the set of triples $\langle i, \rho, os \rangle$, where $i \in \mathcal{PP}$ is the program counter that points to the next instruction to be executed; $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$ is a partial function from local variables to values (that can be view as an array of values) where the set $\mathcal{V}$ of values is defined as $\mathbb{Z}$, and $os \in \mathcal{V}^\star$ is an operand stack.

Operational semantics. The small-step operational semantics of the $\text{JVM}_{\mathcal{I}}$, is given in Figure 3 as a relation $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{I}} \times (\mathsf{State}_{\mathcal{I}} + \mathcal{V})$. This relation is implicitly

$$\frac{P[i] = \mathsf{push}\ n}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle} \qquad \frac{P[i] = \mathsf{binop}\ op \qquad n_2\ \underline{op}\ n_1 = n}{\langle i, \rho, n_1 :: n_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, n :: os\rangle}$$

$$\frac{P[i] = \mathsf{pop}}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \mathsf{swap}}{\langle i, \rho, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle i+1, \rho, v_2 :: v_1 :: os\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho, v :: os\rangle \rightsquigarrow v} \qquad \frac{P[i] = \mathsf{load}\ x}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: os\rangle}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad x \in \mathsf{dom}(\rho)}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, os\rangle} \qquad \frac{P[i] = \mathsf{ifeq}\ j}{\langle i, \rho, 0 :: os\rangle \rightsquigarrow \langle j, \rho, os\rangle}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad n \neq 0}{\langle i, \rho, n :: os\rangle \rightsquigarrow \langle i+1, \rho, os\rangle} \qquad \frac{P[i] = \mathsf{goto}\ j}{\langle i, \rho, os\rangle \rightsquigarrow \langle j, \rho, os\rangle}$$

Fig. 3. Operational Semantics for $\mathrm{JVM}_{\mathcal{I}}$

parameterized by a program P. $\underline{op}$ denotes here the standard interpretation of operation of op in the domain of values $\mathcal{V}$. The semantics of each instruction is quite standard. Instruction push c, pushes a constant c on top of the operand stack. Instruction binop op pops the two top operands of the stack and push the result of the binary operation op using these operands. Instruction pop, just pops the top of the operand stack. Instruction swap, swaps the top two operand stack values. Instruction return ends the execution with the top value of the operand stack. Instruction load x pushes the value currently found in local variable x, on top of the operand stack. Instruction store x pops the top of the stack and stores it in local variable x. Instruction ifeq j pops the top of the stack and depending on weather it is a null value or not, it jumps to the program point j or continue to the next program point. Instruction goto j unconditionally jumps to program point j. For clarity reasons, we hide program points in program examples and use labels to design jump targets.

The transitive closure of $\rightsquigarrow$ to a final value is inductively defined by:

$$\frac{\langle i, \rho, os\rangle \rightsquigarrow v}{\langle i, \rho, os\rangle \Downarrow v} \qquad \frac{\langle i, \rho, os\rangle \rightsquigarrow \langle j, \rho', os'\rangle \quad \langle j, \rho', os'\rangle \Downarrow v}{\langle i, \rho, os\rangle \Downarrow v}$$

The evaluation of a program $\rho \Downarrow v$, from an array of initial local variables ρ to final value is then defined by

$$\rho \Downarrow v \equiv \langle 1, \rho, \varepsilon\rangle \Downarrow v$$

because execution start at program point 1 with an empty operand stack.

Successor relation. The successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program P is defined by the clauses:

— if $P[i] = \mathsf{goto}\ j$, then $i \mapsto j$;
— if $P[i] = \mathsf{ifeq}\ j$, then $i \mapsto i+1$ and $i \mapsto j$;
— if $P[i] = \mathsf{return}$, then i has no successors, and we write $i \mapsto$;
— otherwise, $i \mapsto i+1$.

4.2 Non-Interference

In this fragment, there is no global policy, and a single local policy for the sole method of the program. Furthermore, the local policy does not refer to heap effect, and is thus of the form $\vec{k_v} \longrightarrow k_r$.

The first step to define the security policy is to introduce a notion of indistinguishability between values. In this case, value indistinguishability is trivial.

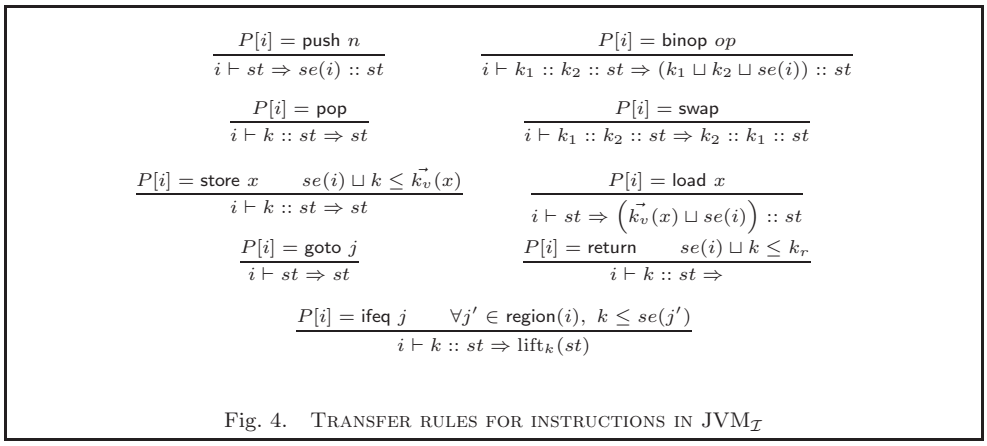

Fig. 4. Transfer rules for instructions in $\mathrm{JVM}_{\mathcal{I}}$

Definition 4.2.1 *Low value indistinguishability.* Two values v and v' are (low)-indistinguishable, written $v \sim v'$, iff $v = v'$.

Then, indistinguishability is extended to local variable maps.

Definition 4.2.2 *Local variables indistinguishability.* For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim \rho'$ if ρ and ρ' have the same domain and $\rho(x) \sim \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\vec{k_v}(x) \leq k_{\mathrm{obs}}$.

Strictly speaking, we should write $\sim_{\vec{k_v}}$, but usually we simply write $\sim$ since there is no risk of confusion.

Definition 4.2.3 *Non-interferent* $\mathrm{JVM}_{\mathcal{I}}$ *program.* A program P is *non-interferent* w.r.t. its policy $\vec{k_v} \longrightarrow k_r$, if for every ρ_1, ρ_2, v_1, v_2 such that $\rho_1 \Downarrow v_1$ and $\rho_2 \Downarrow v_2$ and $\rho_1 \sim_{\vec{k_v}} \rho_2$ and $k_r \leq k_{\mathrm{obs}}$, we have $v_1 \sim v_2$, i.e. $v_1 = v_2$.

4.3 Typing rules

Figure 4 presents a set of typing rules that guarantee non-interference for $\mathsf{JVM}_{\mathcal{I}}$. $\sqcup$ 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 $\lambda l.\ k \sqcup l$. All rules are implicitly parameterized by a cdr region, a security environment se and a signature $\vec{k_a} \longrightarrow k_r$.

Below we comment on some essential rules:

— The transfer rule for an instruction push n prevents direct flows by requiring that the value pushed on top of the operand stack has a security level greater than the security environment at the current program point. The following example, compiled from the source program `return` $\mathtt{y_H}\,\mathtt{?}\,\mathtt{0}\,\mathtt{:}\,\mathtt{1;}$, illustrates the need for this constraint:

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{push}\ 0 \\ \mathsf{goto}\ l_3\end{array}\right\} \mathsf{region}(l_1) \\ l_2: & \mathsf{push}\ 1 \\ l_3: & \mathsf{return} \end{array}$$

The program is interferent with respect to the policy $(y_H : H) \longrightarrow L$, but not typable. Typing rule for return instruction reject this program because the top of the stack type is high. Indeed, instruction push 0 and push 1 are in the region of the branching instruction ifeq l_1 and security environment se is high at this point. A similar constraint appears in the typing rule for binop for the same reasons.

— the typing rule for ifeq requires the stack type on the right hand side of $\Rightarrow$ to be lifted by the level of the guard, i.e. the top of the input stack type. It is

necessary to perform this lifting operation to avoid illicit flows through operand stack leakages. The following example illustrates why we need to lift the operand stack. This is a contrived example because it does not correspond to any simple source code, but it is nevertheless accepted by a standard bytecode verifier.

$$\begin{array}{rl} & \mathsf{push}\ 0 \\ & \mathsf{push}\ 1 \\ & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{swap}\\ \mathsf{pop}\\ \mathsf{goto}\ l_3\end{array}\right\}\mathsf{region}(l_1) \\ l_2: & \mathsf{pop} \\ l_3: & \mathsf{store}\ x_L \end{array}$$

In this example, the final value of variable x_L is equal to the value of y_H. So the program is interferent. It is nevertheless rejected by our type system, thanks to the lift of the operand stack at point l_1 that constrain the top of the stack at point l_3 to be a high value (store rule then prevents the assignment from high to low).
One may argue that lifting the entire stack is too restrictive, as it leads the typing system to reject safe programs; indeed, it should be possible, at the cost of added complexity, to refine the type system to avoid lifting the entire stack.
One may also argue that lifting the stack is unnecessary, because in most programs[4] the stack at branching points only has one element, in which case a more restrictive rule of the form below is sufficient:

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

— The transfer rule for return requires $se(i) \leq k_r$ that avoids return instructions under the guard of expressions with a security level greater than k_r. In addition, the rule requires that the value on top of the operand stack has a security level above k_r, since it will be observed by the attacker. The following example illustrates the need for preventing return instructions in high regions. It corresponds to a source program like `if` ($\mathtt{y_H}$) {`return 0;`} `else` {`return 1;`}.

$$\begin{array}{rl} & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{push}\ 0\\ \mathsf{return}\end{array}\right\}\mathsf{region}(l_1) \\ l_2: & \left.\begin{array}{l}\mathsf{push}\ 1\\ \mathsf{return}\end{array}\right\} \end{array}$$

This program is interferent because there is a return in a high ifeq. This program is rejected by the type system thanks to the ifeq rule which lifts the security environment, and the return rule which prevents the program from returning in a high security environment.

On a more general note, our type system does not support context sensitivity, as the security level of local variables is fixed throughout execution. As a consequence of context insensitivity, the type system restricts the possibilities of local variable reuse. However, Leroy [Leroy 2002] argues that removing local variable polymorphism for Java bytecode is important for efficient on-device verification and that it has a negligible impact on performance and resource usage. We believe that his observations remain applicable to our information flow type system.

[4] And even if this condition does not hold, code transformation is able to obtain an equivalent program respecting it [Leroy 2002].

4.4 Type system soundness

The type system is sound, in the sense that if a program is typable then it is non-interferent.

THEOREM 4.4.1. *Let P be a* $\mathrm{JVM}_{\mathcal{I}}$ *program and* (region, jun) *a safe cdr for P (according to SOAP properties). Suppose P is typable with respect to* region *and to a signature $\vec{k_a} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\vec{k_a} \longrightarrow k_r$.*

Soundness proof follows the method sketched in Section 2. The four base lemmas, are based on the notion of state indistinguishability. The main difficulty in defining state indistinguishability resides in defining a good notion of operand stack indistinguishability: in order to account for high branching instructions, indistinguishability between states must encompass states that have operand stacks of different length. Indistinguishability between operand stacks is needed to establish the lemmas that claim that during execution indistinguishability of states is invariant.

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 as shown in Figure 5. High operand stacks are defined relative to a stack type: formally, let $os \in \mathcal{V}^\star$ be an operand stack and $st \in \mathcal{S}^\star$ be a stack type; we write $\mathsf{high}(os, st)$ if os and st have the same length n and $st[i] \not\leq k_{\mathrm{obs}}$ for every $1 \leq i \leq n$.

Definition 4.4.2 *Operand stack indistinguishability.* Let $os, os' \in \mathcal{V}^\star$ and $st, st' \in \mathcal{S}^\star$. Then $os \sim_{st,st'} os'$ is defined inductively as follows:

$$\frac{\mathsf{high}(os, st) \qquad \mathsf{high}(os', st')}{os \sim_{st,st'} os'}$$

$$\frac{os \sim_{st,st'} os' \qquad v \sim v' \qquad k \leq k_{\mathrm{obs}}}{v :: os \sim_{k::st,k::st'} v' :: os'}$$

$$\frac{os \sim_{st,st'} os' \qquad k \not\leq k_{\mathrm{obs}} \qquad k' \not\leq k_{\mathrm{obs}}}{v :: os \sim_{k::st,k'::st'} v' :: os'}$$

Note that in the second rule the top of the two stack types are necessary equal (and low), while in the last rule they can be distinct (but not low). This distinction is necessary because we handle an arbitrary lattice of security levels.

Assuming that programs pass bytecode verification, one can simplify a bit the above definition. Indeed, bytecode verification requires that at each program point the height of the operand stack be fixed. Under this assumption, operand stack equivalence only requires that any two high operand stacks are equivalent and that operand stacks of the same height are equivalent if they are point-wise equivalent.

State indistinguishability can then be defined component-wise on state structure.

Definition 4.4.3 *State indistinguishability.* Two states $\langle i, \rho, os\rangle$ and $\langle i', \rho', os'\rangle$ are indistinguishable w.r.t. st, $st' \in \mathcal{S}^\star$, denoted $\langle i, \rho, os\rangle \sim_{st,st'} \langle i', \rho', os'\rangle$, iff $os \sim_{st,st'} os'$ and $\rho \sim \rho'$ hold.

Appendix A presents the four basic lemmas necessary for the soundness proof of theorem 4.4.1. The non-interference theorem is then proved following the method proposed in Section 2.4.

5. $\mathrm{JVM}_{\mathcal{O}}$: THE OBJECT-ORIENTED EXTENSION OF $\mathrm{JVM}_{\mathcal{I}}$

The object-oriented extension of $\mathrm{JVM}_{\mathcal{I}}$, namely $\mathrm{JVM}_{\mathcal{O}}$, includes instance fields, creation of new instances, and null pointers. We assume that programs are not enabled to do pointer arithmetic in $\mathrm{JVM}_{\mathcal{O}}$ (pointer arithmetic is prevented by standard bytecode verification).

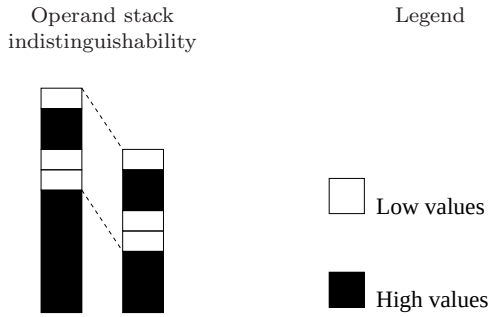

Fig. 5. Operand stack indistinguishability

instr	::= ...	
	| new C	create new object in the heap
	| getfield f	load value of field f on stack
	| putfield f	store top of stack in field f

where $C \in \mathcal{C}$ and $f \in \mathcal{F}$.

Fig. 6. Additional instruction set for $\text{JVM}_{\mathcal{O}}$

5.1 Programs, memory model and operational semantics

Programs. $\text{JVM}_{\mathcal{O}}$ programs are as $\text{JVM}_{\mathcal{I}}$ programs, but also come equipped with a set $\mathcal{C}$ of class names, and a set $\mathcal{F}$ of identifiers representing field names. Programs use an extended set of instructions, given in Figure 6.

States. Compared to $\text{JVM}_{\mathcal{I}}$, the set of $\text{JVM}_{\mathcal{O}}$ values is extended to $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations and *null* denotes the null pointer. A $\text{JVM}_{\mathcal{O}}$ state is now of the form $\langle i, \rho, os, h\rangle$, where i, ρ, and os are defined as in $\text{JVM}_{\mathcal{I}}$ and h is a heap, that accommodates dynamically created objects. Heaps are modeled as partial functions $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$, i.e. each object $o \in \mathcal{O}$ posses a class (noted $\mathsf{class}(o)$) and a partial function to access field values. We note $o.f$ the access to the value of field f, $o \oplus \{f \mapsto v\}$ denotes the update of an object o at field f with a value v ($h \oplus \{l \mapsto o\}$ is used in the same way for heap update) and Heap is the set of heaps.

Operational semantics. The operational semantics for the new instructions of $\text{JVM}_{\mathcal{O}}$ relies on an allocator function $\mathsf{fresh} : \mathsf{Heap} \to \mathcal{L}$ that given a heap returns the location for that object, and on a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$ that returns for each class a default object of that class. $\mathsf{default}$ is specified according to the standard Java convention[5]: for all defined field $f \in \mathcal{F}$ of a class $C \in \mathcal{C}$,

$$\mathsf{default}(C).f = \begin{cases} 0 & \text{if } f \text{ has a numeric type} \\ null & \text{if } f \text{ has a object type} \end{cases}$$

The semantics is given in Figure 7 as a relation $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{O}} \times (\mathsf{State}_{\mathcal{O}} + (\mathcal{V} \times \mathsf{Heap}))$.

Instruction $\mathsf{new}\ C$ pushes a fresh location on top of the operand stack associated to a new initialized object. The heap is updated with this new object. Instruction $\mathsf{getfield}\ f$ pops a location l from the operand stack. The value of the field f in location l is fetched and pushed onto the operand stack. Instruction $\mathsf{putfield}\ f$ uses the top of the stack to update the object associated with the location in second

[5]We assume each field f has a declared type.

$$\frac{P[i] = \mathsf{new}\ C \qquad l = \mathrm{fresh}(h)}{\langle i, \rho, os, h\rangle \rightsquigarrow \langle i+1, \rho, l :: os, h \oplus \{l \mapsto \mathsf{default}(C)\}\rangle}$$

$$\frac{P[i] = \mathsf{getfield}\ f \qquad l \in dom(h)}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, h(l).f :: os, h\rangle}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad l \in dom(h) \qquad f \in dom(h(l))}{\langle i, \rho, v :: l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, os, h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\}\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho, v :: os, h\rangle \rightsquigarrow v, h}$$

Fig. 7. Operational Semantics for additional $\mathrm{JVM}_{\mathcal{O}}$ instructions

position on the operand stack. Instruction return now returns the top of the operand stack, and the current heap.

As for $\mathrm{JVM}_{\mathcal{I}}$, we let $\Downarrow$ denote the transitive closure of $\rightsquigarrow$ as in $\mathrm{JVM}_{\mathcal{O}}$ and write $\rho, h \Downarrow v, h'$ as a shorthand for $\langle 1, \rho, \epsilon, h\rangle \Downarrow (v, h')$.

Successor relation. The successor relation is extended with the clause $i \mapsto i+1$ for all new instructions.

5.2 Non-Interference

Indistinguishability for $\mathrm{JVM}_{\mathcal{O}}$ states is extended and defined relative to a global mapping $\mathsf{ft} : \mathcal{F} \to \mathcal{S}$ that maps fields to security levels. ft will be left implicit in the rest of the paper. In order to extend the notion of indistinguishability to heaps we follow [Banerjee and Naumann 2005]. We consider that heaps with different allocations of "high" objects (i.e. objects that have been created in a high security environment) are indistinguishable by an attacker; therefore indistinguishability is defined relative to a bijection β on (a partial set of) locations in the heap. The bijection maps low objects (low objects are objects whose references might be stored in low fields or variables) allocated in the heap of the first state to low objects allocated in the heap of the second state. The objects might be indistinguishable, even if their locations are different during execution. Since values can now also be locations, definition of value indistinguishability is defined also relative to bijection β.

Definition 5.2.1 *Value indistinguishability.* Given two values $v_1, v_2 \in \mathcal{V}$, and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ value indistinguishability $v_1 \sim_\beta v_2$ is defined by the clauses:

$$null \sim_\beta null \qquad \frac{v \in \mathcal{N}}{v \sim_\beta v}$$

$$\frac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \sim_\beta v_2}$$

Operand stack indistinguishability and local variables indistinguishability are now parameterized by β since values on top of the operand stack and in variables can also be locations.

Definition 5.2.2 *Local variables indistinguishability.* For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, we have $\rho \sim_\beta \rho'$ if ρ and ρ' have the same domain and $\rho(x) \sim_\beta \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\vec{k_v}(x) \leq k_{\mathrm{obs}}$.

Definition 5.2.3 *Operand stack indistinguishability.* Let $os, os' \in \mathcal{V}^\star$, $st, st' \in \mathcal{S}^\star$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$. Then $os \sim_{st,st',\beta} os'$ is defined inductively as

follows:

$$\frac{\mathsf{high}(os, st) \qquad \mathsf{high}(os', st')}{os \sim_{st,st',\beta} os'}$$

$$\frac{os \sim_{st,st',\beta} os' \qquad v \sim v' \qquad k \leq k_{\text{obs}}}{v :: os \sim_{k::st,k::st',\beta} v' :: os'}$$

$$\frac{os \sim_{st,st',\beta} os' \qquad k \not\leq k_{\text{obs}} \qquad k' \not\leq k_{\text{obs}}}{v :: os \sim_{k::st,k'::st',\beta} v' :: os'}$$

The definition of object indistinguishability says that two objects are indistinguishable if they have the class and their field values are indistinguishable.

Definition 5.2.4 *Object indistinguishability.* Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a function $\beta \in \mathcal{LL} \rightharpoonup \mathcal{LL}$ if and only if o_1 and o_2 are objects of the same class and for all fields $f \in \mathsf{dom}(o_1)$ such that $\mathsf{ft}(f) \leq k_{\text{obs}}$, $o_1.f \sim_\beta o_2.f$.

Note that because o_1 and o_2 are objects of the same class we have $\mathsf{dom}(o_1) = \mathsf{dom}(o_2)$ and $o_2(f)$ is well defined.

Heap indistinguishability requires β to be a bijection between the *low domains* (*i.e.* locations that might be reachable from low local variables/fields) of the considered heaps.

Definition 5.2.5 *Heap indistinguishability.* Two heaps h_1 and h_2 are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_\beta h_2$, if and only if:

— β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$;
— $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$;
— for every $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_\beta h_2(\beta(l))$;

As in $\text{JVM}_\mathcal{I}$, state indistinguishability can then be defined component-wise on state structure.

Definition 5.2.6 *State indistinguishability.* Two states $\langle i, \rho, os, h\rangle$ and $\langle i', \rho', os', h'\rangle$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and two stack types $st, st' \in \mathcal{S}^\star$, denoted $\langle i, \rho, os, h\rangle \sim_{st,st',\beta} \langle i', \rho', os', h'\rangle$, iff $os \sim_{st,st',\beta} os'$, $\rho \sim_\beta \rho'$ and $h \sim_\beta h'$ hold.

Finally, non-interference in $\text{JVM}_\mathcal{O}$ is extended using the relations defined above.

Definition 5.2.7 *Non-interferent* $\text{JVM}_\mathcal{O}$ *program.* A program P is *non-interferent* w.r.t. its policy $\vec{k_v} \longrightarrow k_r$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $v_1, v_2 \in \mathcal{V}$ such that $\rho_1, h_1 \Downarrow v_1, h_1'$, $\rho_2, h_2 \Downarrow v_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\vec{k_v},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_{\beta'} v_2$.

Here $\beta \subseteq \beta'$ means that $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(\beta')$ and for all locations $l \in \mathsf{dom}(\beta)$, $\beta(l) = \beta'(l)$. The definition of non-interference allows for β to be extended, in order to handle objects that are dynamically created during execution.

In order to better understand the notion of partial function β in the definition of secure method, we end this section with a simpler property verified by particular non-interferent $\text{JVM}_\mathcal{O}$ programs.

LEMMA 5.2.1. *Let p a program returning a numerical value and which is non-interferent with respect to a policy $\vec{k_a} \longrightarrow k_r$. Let h_0, h_1, h_2 some heaps, ρ_1, ρ_2 two arrays of local variables such that for all variable x, $\mathsf{ft}(x) \leq k_{\text{obs}}$ implies $\rho_1(x) =$*

$$\frac{P[i] = \mathsf{new}\ C}{i \vdash st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f)}{i \vdash k_1 :: k_2 :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{getfield}\ f}{i \vdash k :: st \Rightarrow (k \sqcup \mathsf{ft}(f) \sqcup se(i)) :: st}$$

Fig. 8. Additional typing transfer rules for $\mathrm{JVM}_{\mathcal{O}}$

$\rho_2(x)$ (parameter are equal for low variables) and n_1, n_2 two numeric values such that

$$\rho_1, h \Downarrow (n_1, h_1) \qquad \textit{and} \qquad \rho_2, h \Downarrow (n_2, h_2)$$

Then, if $k_r \leq k_{\mathrm{obs}}$, both returned values are equals: $n_1 = n_2$.

5.3 Typing rules

The abstract transition system of the $\mathrm{JVM}_{\mathcal{O}}$ extends that of the $\mathrm{JVM}_{\mathcal{I}}$ with the typing transfer rules of Figure 8. As in $\mathrm{JVM}_{\mathcal{I}}$, all rules are implicitly parameterized by a cdr region, a security environment se and a signature $\vec{k_a} \longrightarrow k_r$.

— The transfer rule for new adds to the stack type the security level of the current program point, which imposes a constraint on security level from which the object can be accessed. For example, if new is executed in a high security environment, then the reference to the object cannot be accessed from a low variable. However, if the object is created in a low security environment it can either be stored in a high or low variable/field.

— The transfer rule for putfield requires that $k_1 \leq \mathsf{ft}(f)$ (where k_1 is the security type of the object of the field) in order to prevent an explicit flow from a high value to a low field. The constraint $se(i) \leq \mathsf{ft}(f)$ prevents an implicit flow caused by an assignment to a low field in a high security environment. Finally, the constraint $k_2 \leq \mathsf{ft}(f)$ prevents modifying low fields of high objects that are alias to a low object.
The following example illustrates this last point. It corresponds to a source program like

```
C xL = new C();
zH = yH ? new C() : xL;
zH.fL = 1;
```

We assume that C is a class that has a low field named f_L. Let x_L be a low variable and y_H, z_H high variables.

$$\begin{array}{ll}
 & \mathsf{new}\ C \\
 & \mathsf{store}\ x_L \\
 & \mathsf{load}\ y_H \\
l_1: & \mathsf{ifeq}\ l_2 \\
 & \mathsf{new}\ C \\
 & \mathsf{goto}\ l_3 \\
l_2: & \mathsf{load}\ x_L \\
l_3: & \mathsf{store}\ z_H \\
 & \mathsf{load}\ z_H \\
 & \mathsf{push}\ 1 \\
 & \mathsf{putfield}\ f_L
\end{array} \Big\} \mathsf{region}(l_1)$$

In this program, depending on the test on y_H, variable x_L and z_H might be aliases to the same object (of class C). Hence, the assignment to field f_L might have

side effect on the object in x_L. This program is rejected thanks to the putfield rule which avoids this type of leaks due to alias (with the constraint $k_2 \leq \mathsf{ft}(f)$ preventing assignments to low fields from high target objects).

— In the rule for getfield f the value pushed on the operand stack has a security level at least greater than $\mathsf{ft}(f)$ and the level k of the location (to prevent explicit flows) and at least greater than $se(i)$ for implicit flows.

5.4 Type system soundness

THEOREM 5.4.1. *Let P be a* $\mathrm{JVM}_{\mathcal{O}}$ *program and* (region, jun) *a safe cdr for P (according to SOAP properties). Suppose P is typable with respect to* region *and to a signature $\vec{k_v} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\vec{k_v} \longrightarrow k_r$.*

The soundness proof closely follows the $\mathrm{JVM}_{\mathcal{I}}$ soundness proof , except that now we have to manipulate heaps and some partial function β.

The four basic lemmas necessary for the soundness proof of theorem 5.4.1 are listed in Appendix B. In the first case, the partial function β may be extended if $P[i]$ is of the form new C and the context is low ($se(i) \leq k_{\mathrm{obs}}$). Note that we do not need to extend partial function β when the step occurs in a high context ($se(i) \not\leq k_{\mathrm{obs}}$).

6. $\mathrm{JVM}_{\mathcal{C}}$: THE METHOD EXTENSION OF $\mathrm{JVM}_{\mathcal{O}}$

The purpose of this section is to extend our analysis to methods. The extension is compatible with bytecode verification, in the sense that the analysis is modular.

6.1 Programs, memory model and operational semantics

Programs. Each program comes equipped with a set $\mathcal{M}$ of method names, and a set $\mathcal{C}$ of classes, as in $\mathrm{JVM}_{\mathcal{O}}$. The set of classes is now organised as a hierarchy to model the inheritance of class. This hierarchy will be used to resolve virtual calls.

Each method m possesses a list of instructions P_m. For simplicity, we impose that all methods return a value. The set of instructions of $\mathrm{JVM}_{\mathcal{O}}$ is extended with the new instruction invokevirtual m_{ID} for calling a virtual method. Here m_{ID} is a method identifier which may correspond to several methods in the class hierarchy according to overriding of methods. We assume there is a function lookup_P attached to each program P that takes a method identifier and a class name and returns the method to be executed.

States. While JVM states contain a frame stack to handle method invocations, it is convenient for showing the correctness of static analyzes to rely on an equivalent semantics where method invocation is performed in one big step transition. Hence a $\mathrm{JVM}_{\mathcal{C}}$ state is defined as in $\mathrm{JVM}_{\mathcal{O}}$.

Operational semantics. While small-step semantics uses a call stack to store the calling context and retrieve it during a return instruction, the big step semantics directly calls the full evaluation of the called method from an initial state to a return value and uses it to continue the current computation. The big-step operational semantics is given in Figure 9. As can be seen in the first rule, semantics of instructions is like in $\mathrm{JVM}_{\mathcal{O}}$, except for the new instruction invokevirtual. The second rule gives the semantics of the virtual call. 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 (trough the lookup operator). The transitive closure of $\rightsquigarrow_m$ 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

$$\frac{P_m[i] = ins \qquad ins \neq \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \langle i, \rho, os, h\rangle \rightsquigarrow_{\mathrm{JVM}_{\mathcal{O}}} \langle i', \rho', os', h'\rangle}{\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ \langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow^{(n)}_{m'} v, h'\end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \overset{(n+1)}{\rightsquigarrow}_m \langle i+1, \rho, v :: os_2, h'\rangle}$$

$$\frac{\langle i, \rho, os, h\rangle \rightsquigarrow_{\mathrm{JVM}_{\mathcal{O}}} v}{\langle i, \rho, os, h\rangle \Downarrow^{(0)}_m v} \qquad \frac{\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_m \langle j, \rho', os', h'\rangle \quad \langle j, \rho', os', h'\rangle \Downarrow^{(p)}_m v}{\langle i, \rho, os, h\rangle \Downarrow^{(n+p)}_m v}$$

Fig. 9. Operational Semantics for $\mathrm{JVM}_{\mathcal{C}}$

for the other variables[6].

We opt for a big-step operational semantics because it simplifies the notion of indistinguishability between states. In the presence of a small-step semantics states possess stack of frames (one frame corresponding to each method in the calling chain) and hence indistiguishability must take account of frames of high and low methods which can throw and propagate low and high exceptions. It is also needed for indistiguishability to state if the method is invoked in a low or high target object. Using this alternative semantics has brought a significant simplification to the proofs of the analysis.

The relation $\rightsquigarrow$ is now parameterized by a counter representing the number of method calls occurring between two consecutive steps. The transitive closure of $\rightsquigarrow$ is simultaneously defined with a similar counter.

As in other JVM fragments, we note $\rho, h \Downarrow^{(n)}_m v, h'$ when $\langle 1, \rho, \epsilon, h\rangle \Downarrow^{(n)}_m v, h'$. We note $\rho, h \Downarrow_m v, h'$ for $\exists n,\ \rho, h \Downarrow^{(n)}_m v, h'$.

Successor relation. We extend the successor relation of $\mathrm{JVM}_{\mathcal{O}}$ with the clause $i \mapsto i+1$ for the new instruction invokevirtual. It illustrates our modular verification technique : *cdr* is computed method by method.

6.2 Non-Interference

Non-Interference for a $\mathsf{JVM}_{\mathcal{C}}$ program is given by local policies defined by security signatures for every method and a global policy defined by a mapping of fields to security levels, namely ft.

As mentioned in Section 2, method signatures are of the form

$$\vec{k_v} \xrightarrow{k_h} k_r$$

where $\vec{k_v}$ provides the security level of the method arguments (and to all intermediate variables used in the method), k_h is the effect of the method on the heap and k_r is the security level of the result of the method.

The *heap effect level* k_h is needed to make a modular analysis. It represents a lower bound for security levels of fields that are affected during execution of the method.

A method is allowed to perform field updates only on fields whose level is greater than k_h. We formally define this notion of *side effect preorder*.

Definition 6.2.1 *Side effect preorder.* Two heaps $h_1, h_2 \in \mathsf{Heap}$ are *side effect preordered* with respect to a security level $k \in \mathcal{S}$ (noted $h_1 \preceq_k h_2$) if and only if

[6]We assume all other variable used for local computation in the method are initialised by a default value according to their type

$\mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2)$ and for all location $l \in \mathsf{dom}(h_1)$ and all fields $f \in \mathcal{F}$ such that $k \not\leq \mathsf{ft}(f)$, $h_1(l).f = h_2(l).f$.

This allows to define the notion of *side-effect-safe method*.

Definition 6.2.2. A method m is *side-effect-safe* with respect to a security level k_h if for all local variable $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$, all heaps $h, h' \in \mathsf{Heap}$ and value $v \in \mathcal{V}$, $\rho, h \Downarrow_m v, h'$ implies $h \preceq_{k_h} h'$.

The notion of non-interferent method can be stated using the same indistiguishability relation as in $\text{JVM}_{\mathcal{O}}$. A method m is called *non-interferent for signature* $\vec{k_v} \longrightarrow k_r$ if every time it is executed with indistinguishable arguments according to $\vec{k_v}$ and indistiguishable heaps according to the global policy ft, then the results of the method by normal termination are indistinguishable by k_r and its heaps are indistiguishable according to the global policy.

Definition 6.2.3 *Non-interferent* $\text{JVM}_{\mathcal{C}}$ *method.* A method m is *non-interferent* w.r.t. a policy $\vec{k_v} \longrightarrow k_r$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $v_1, v_2 \in \mathcal{V}$ such that $\rho_1, h_1 \Downarrow_m v_1, h_1'$, $\rho_2, h_2 \Downarrow_m v_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\vec{k_v},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_{\beta'} v_2$.

The notion of safe method is then defined by conjunction of the two previous definitions.

Definition 6.2.4 *Safe* $\text{JVM}_{\mathcal{C}}$ *method.* A method m is *safe* w.r.t. a policy $\vec{k_v} \xrightarrow{k_h} k_r$ if m is side-effect-safe with respect to k_h and m is non-interferent with respect to $\vec{k_v} \longrightarrow k_r$.

Let Γ be a table of method signatures. This table associates to each method identifier[7] m_{ID} and security level $k \in \mathcal{S}$, a security signature $\Gamma_m[k]$. This signature gives the security policy of the method m called on object of level k (as in [Banerjee and Naumann 2005] for source program). The set of security signature of a method m is defined as $\text{Policies}_\Gamma(\text{m}) = \{\ \Gamma_\text{m}[\text{k}] \mid \text{k} \in \mathcal{S}\ \}$. In the rest of the paper Γ will often be left implicit. We use it to define the notion of *safe program*.

Definition 6.2.5 *Safe* $\text{JVM}_{\mathcal{C}}$ *program.* A program is *safe* with respect to a table of method signatures Γ if for all its method m, m is safe with respect to all policies in $\text{Policies}_\Gamma(\text{m})$.

Example 6.2.6. Let P be a program that includes a method m and a class C with field f. Let m have variables x_1, x_2, y and no handlers defined. Let the non-interference policy for P be given by a security signature $L, L, H \xrightarrow{L} H$ for m and a security signature $\xrightarrow{L} L$ for **main**, and a global mapping ft such that $\mathsf{ft}(f) = L$.

If the code of m is defined by:

```
      new C
      store x2
      load x1
      ifeq l1
      load x2
      push 1
      putfield f
l1 :  load x2
      getfield f
      return
```

[7] Associating signatures with method identifiers instead of methods allows to enforce that method overriding preserves declared security signatures.

$$\frac{P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f)}{\mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash k_1 :: k_2 :: st \Rightarrow st}$$

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

Fig. 10. New transfer rules for instructions of $\mathrm{JVM}_{\mathcal{C}}$

then method m is non-interferent because: starting from equal values for x (x represents the low part of the state, since the security signature says that x is low), the result of the method will always be the value of the low field f that is 1, hence every results are indistiguishable by L, as stated in the signature of m. The method cannot terminate abnormally since $\mathsf{new}\ C$ is not a null value and there are no affections to the high fields. This respects the low write effect of the method required by the policy.

Now assume that in program point 6 there is an instruction $\mathsf{load}\ y$ instead of $\mathsf{load}\ x_2$. Since according to the local policy, y is a high variable, its value might be different for 2 different indistinguishable states. For example assume that for two states with value of x equal to 1 variable y has values *null* and a location in the domain of the heap that points to an object of class C. Then m will terminate abnormally in the first case, and normally in the second. Since this depends on the value of high variable y, the results are not indistinguishable by L as stated by the security signature (exceptional effect). Hence, m is interferent.

6.3 Typing rules

The information flow type system enforces a method-wise verification strategy, using method signatures in the transfer rule for method invocation. All typing rules are those of the $\mathrm{JVM}_{\mathcal{O}}$ typing rules, except for putfield which needs a modification and the virtual call rule which is new. This two rules are given in Figure 10.

Concerning putfield only one constraint is added w.r.t. the previous $\mathrm{JVM}_{\mathcal{O}}$ rule. The new constraint $k_h \leq \mathsf{ft}(f)$ prevents modification of fields with a level not greater than the heap effect of the current method.

The typing rule for virtual call contains several constraints. The heap effect level of the called method is constrained in several ways. The goal of the constraint $k \leq k'_h$ is to avoid invocation of methods with low effect on the heap with high target objects. Two different target objects (in two executions) may mean that the body of the method to be executed is different in each execution. If the effect of the method is low ($k_h \leq k_{\mathrm{obs}}$), then low memory is differently modified in both executions, leading to information leak. 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 field with a level lower that k_h.

The security level of the return value is $(k'_r \sqcup k \sqcup se(i))$. The security level k'_r in $(k'_r \sqcup k \sqcup se(i))$, obtained from the signature of m_{ID} , prevents that its result flows to variables or fields with lower security level. The security level k prevents flows due to execution of two distinct methods.

We include here an example that illustrates how object-oriented features can lead to interference. We refer to [Banerjee and Naumann 2005] for further examples.

Example 6.3.1. Let class C be a super class of a class D. Let method *foo* be declared in D, and a method m declared in C and overridden in D as illustrated

by the following source program[8]:

```
class C {
  int m() {return 0;}
}
class D extends C {
  int m() {return 1;}
  int foo(boolean y_H) {return (y_H ? new C() : this).m();}
}
```

$D.foo$:	$C.m$:	$D.m$:
load y_H	push 0	push 1
ifeq l_1	return	return
new C		
goto l_2		
l_1 : load $this$		
l_2 : invokevirtual m		
return		

At run time, either code $C.m$ or code $D.m$ is executed depending on the value of high variable y_H. Information about y_H may be inferred by observing the return value of method m.

6.4 Type system soundness

THEOREM 6.4.1. *Let P be a* $\text{JVM}_{\mathcal{C}}$ *program,* Γ *a table of signatures and for all method m in P,* $(\mathsf{region}_m, \mathsf{jun}_m)$ *a safe cdr for m (according to SOAP properties). Suppose all methods m in P are typable with respect to* region_m *and to all signatures in* $\text{Policies}_\Gamma(\text{m})$. *Then P is safe with respect to* Γ.

Main proof steps are given in Appendix C.

7. $\text{JVM}_{\mathcal{G}}$: THE EXCEPTION-HANDLING EXTENSION OF $\text{JVM}_{\mathcal{C}}$

In this section we show how $\mathsf{JVM}_{\mathcal{C}}$ is extended with an exception handling mechanism. The extension of the type system to multiple exceptions is achieved by a fine-grained definition of control dependence regions that is parameterized by a class-analysis and an exception-analysis (which form a part of the PA analyser introduced in Section 2). The class analysis returns an over-approximation of classes of exceptions of a program point, and the exception analysis returns an over-approximation of escaping exceptions of a method. For the soundness of the information flow type system, we assume that both the class-analysis and the exception-analysis are in the Trusted Computing Base. Thus, the type system exploits the information of the class analysis and signature of methods (that coincides with the exception-analysis results) to add constraints on the security environment according to adequate regions for the type of escaping exceptions (if any).

7.1 Programs, memory model and operational semantics

Programs. Programs are similar to those in the $\text{JVM}_{\mathcal{C}}$ model. However, the instruction set of the $\text{JVM}_{\mathcal{C}}$ is extended with the bytecode throw.

Furthermore, we assume that programs come equipped with a partial function[9] $\mathsf{Handler}_m : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$ that for each method m selects the appropriate handler for a given program point. If an exception of class $C \in \mathcal{C}$ is thrown at program

[8] We omit the call of the initializer.

[9] This opaque handling function hides the notions of handler list and sub-class used in Java.

point $i \in \mathcal{PP}$ then, if $\mathsf{Handler}_m(i, C) = t$, then the control will be transfered to program point t, and if $\mathsf{Handler}_m(i, C)$ is undefined (noted $\mathsf{Handler}_m(i, C) \uparrow$), the exception is uncaught in method m.

States. $\mathrm{JVM}_{\mathcal{G}}$ states include $\mathrm{JVM}_{\mathcal{C}}$ states and extend them with new final states. We model final states as $(\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$: a final state is either of the form $(v, h) \in \mathcal{V} \times \mathsf{Heap}$ for normal termination, or of the form $(\langle l \rangle, h) \in \mathcal{L} \times \mathsf{Heap}$ for abrupt termination by an uncaught exception pointed by a location l in the heap h.

Operational semantics. We give in Figure 11 the semantics of exception-throwing instructions in $\mathrm{JVM}_{\mathcal{G}}$. Rules for the rest of the instructions are as in $\mathrm{JVM}_{\mathcal{C}}$. There are three more rules for the virtual call instruction. The first and the second model the cases where execution of the called method terminates by an uncaught exception. In the first rule the thrown exception is caught in method m while in the second rule it is uncaught and m then terminates abnormally. In both cases, we impose that the thrown exception has been statically predicted by the $\mathsf{excAnalysis}(m_{\mathrm{ID}})$ result of the exception analysis[10]. The third rule corresponds to a null pointer exception thrown because the virtual call occurred on a null reference. We use **np** as the class associated to the null pointer exception. When a native exception **np** is thrown the catching mechanism is modeled by the function RuntimeExceptionHandling. Each instruction which performs an access on a reference (getfield f, putfield f and throw) have a similar semantics. The last two rules concern the new instruction throw which throws the exception pointed by the reference on top of the stack. Transitions are now parameterized by a tag $\tau \in \{\emptyset\} + \mathcal{C}$ to describe the nature of the transition (see the successor relation below). We will sometimes omit the tag τ in the notation $\overset{(n)}{\rightsquigarrow}_{m,\tau}$ for clarity.

Successor relation. The successor relation is now decorated by an element (called *tag*) in $\{\emptyset\} + \mathcal{C}$ in order to reflect the nature of the underlying semantics step: $\emptyset$ for a normal step (as in $\mathrm{JVM}_{\mathcal{C}}$) and $c \in \mathcal{C}$ for a step where an exception of class C has been thrown. The definition of this new relation is given in Figure 12. This relation can be statically computed thanks to the handler function of each method. Successors of a throw instruction are approximated thanks to the class analysis result and successors of a invokevirtual thanks to the exception analysis result of the called method.

SOAP properties. Cdr results are now associated not only to program points but also to tags:

$$\mathsf{region}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to \wp(\mathcal{PP}) \qquad \mathsf{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

We call *return point* a point i such that there exists $\tau \in \{\emptyset\} + \mathcal{C}$ with $i \mapsto^{\tau}$. When possible we will write $i \mapsto j$ for $\exists \tau,\ i \mapsto^{\tau} j$.

SOAP1:. for all program points i, j, k and tag τ such that $i \mapsto j$, $i \mapsto^{\tau} k$ and $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i, \tau)$ or $k = \mathsf{jun}(i, \tau)$;

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

SOAP3:. for all program points i, j and tag τ, if $j \in \mathsf{region}(i, \tau)$ (or $i = j$) and j is a return point then $\mathsf{jun}(i, \tau)$ is undefined;

SOAP4:. for all program points i and tags τ_1, τ_2, if $\mathsf{jun}(i, \tau_1)$ and $\mathsf{jun}(i, \tau_2)$ are defined and $\mathsf{jun}(i, \tau_1) \neq \mathsf{jun}(i, \tau_2)$ then $\mathsf{jun}(i, \tau_1) \in \mathsf{region}(i, \tau_2)$ or $\mathsf{jun}(i, \tau_2) \in \mathsf{region}(i, \tau_1)$;

[10]This hypothesis is directly put as precondition of the semantics rules, in the same way that only well-typed states are considered when assuming a program is byte-code verified. It is straightforward to show that our instrumented semantics coincides with the standard semantics if the exception analysis is safe.

$$\frac{\langle i,\rho,os,h\rangle \overset{(n)}{\rightsquigarrow}_m \langle i',\rho',os',h'\rangle \text{ in } \mathrm{JVM}_{\mathcal{C}} \text{ semantics}}{\langle i,\rho,os,h\rangle \overset{(n)}{\rightsquigarrow}_{m,\emptyset} \langle i',\rho',os',h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i]=\mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m'=\mathsf{lookup}_P(m_{\mathrm{ID}},\mathsf{class}(h(l))) \\ l\in dom(h) \qquad \mathsf{length}(os_1)=\mathsf{nbArguments}(m_{\mathrm{ID}}) \\ \langle 1,\{this\mapsto l,\vec{x}\mapsto os_1\},\epsilon,h\rangle \Downarrow^{(n)}_{m'} \langle l'\rangle,h' \\ e=\mathsf{class}(h'(l')) \qquad \mathsf{Handler}_m(i,e)=t \qquad e\in\mathsf{excAnalysis}(m_{\mathrm{ID}})\end{array}}{\langle i,\rho,os_1::l::os_2,h\rangle \overset{(n+1)}{\rightsquigarrow}_{m,e} \langle t,\rho,l'::\epsilon,h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i]=\mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m'=\mathsf{lookup}_P(m_{\mathrm{ID}},\mathsf{class}(h(l))) \\ l\in dom(h) \qquad \mathsf{length}(os_1)=\mathsf{nbArguments}(m_{\mathrm{ID}}) \\ \langle 1,\{this\mapsto l,\vec{x}\mapsto os_1\},\epsilon,h\rangle \Downarrow^{(n)}_{m'} \langle l'\rangle,h' \\ e=\mathsf{class}(h'(l')) \qquad \mathsf{Handler}_m(i,e)\uparrow \qquad e\in\mathsf{excAnalysis}(m_{\mathrm{ID}})\end{array}}{\langle i,\rho,os_1::l::os_2,h\rangle \overset{(n+1)}{\rightsquigarrow}_{m,e} \langle l'\rangle,h'}$$

$$\frac{P_m[i]=\mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad l'=\mathrm{fresh}(h) \qquad \mathsf{length}(os_1)=\mathsf{nbArguments}(m_{\mathrm{ID}})}{\langle i,\rho,os_1::null::os_2,h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h,l',\mathbf{np},i,\rho)}$$

$$\frac{P_m[i]=\mathsf{getfield}\ f \qquad l'=\mathrm{fresh}(h)}{\langle i,\rho,null::os,h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h,l',\mathbf{np},i,\rho)}$$

$$\frac{P_m[i]=\mathsf{putfield}\ f \qquad l'=\mathrm{fresh}(h)}{\langle i,\rho,v::null::s,h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h,l',\mathbf{np},i,\rho)}$$

$$\frac{P_m[i]=\mathsf{throw} \qquad l'=\mathrm{fresh}(h)}{\langle i,\rho,null::s,h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h,l',\mathbf{np},i,\rho)}$$

$$\frac{\begin{array}{c}P_m[i]=\mathsf{throw} \qquad l\in\mathsf{dom}(h) \qquad e=\mathsf{class}(h(l)) \\ \mathsf{Handler}_m(i,e)=t \qquad e\in\mathsf{classAnalysis}(m,i)\end{array}}{\langle i,\rho,l::os,h\rangle \overset{(0)}{\rightsquigarrow}_{m,e} \langle t,\rho,l::\epsilon,h\rangle}$$

$$\frac{\begin{array}{c}P_m[i]=\mathsf{throw} \qquad l\in\mathsf{dom}(h) \qquad e=\mathsf{class}(h(l)) \\ \mathsf{Handler}_m(i,e)\uparrow \qquad e\in\mathsf{classAnalysis}(m,i)\end{array}}{\langle i,\rho,l::os,h\rangle \overset{(0)}{\rightsquigarrow}_{m,e} \langle l\rangle,h}$$

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

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

Fig. 11. New operational semantics rules for $\mathrm{JVM}_{\mathcal{G}}$

$$\frac{i\mapsto_{\mathrm{JVM}_{\mathcal{C}}} j}{i\mapsto^{\emptyset} j} \qquad \frac{i\mapsto_{\mathrm{JVM}_{\mathcal{C}}}}{i\mapsto^{\emptyset}}$$

$$\frac{P_m[i]\in\{\mathsf{getfield}\ f,\mathsf{putfield}\ f,\mathsf{throw},\mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i,\mathbf{np})=t}{i\mapsto^{\mathbf{np}} t}$$

$$\frac{P_m[i]\in\{\mathsf{getfield}\ f,\mathsf{putfield}\ f,\mathsf{throw},\mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i,\mathbf{np})\uparrow}{i\mapsto^{\mathbf{np}}}$$

$$\frac{P_m[i]=\mathsf{throw} \qquad C\in\mathsf{classAnalysis}(m,i) \qquad \mathsf{Handler}(i,C)=t}{i\mapsto^{C} t}$$

$$\frac{P_m[i]=\mathsf{throw} \qquad C\in\mathsf{classAnalysis}(m,i) \qquad \mathsf{Handler}(i,C)\uparrow}{i\mapsto^{C}}$$

$$\frac{P_m[i]=\mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C\in\mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i,C)=t}{i\mapsto^{C} t}$$

$$\frac{P_m[i]=\mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C\in\mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i,C)\uparrow}{i\mapsto^{C}}$$

Fig. 12. Successor relation for $\mathrm{JVM}_{\mathcal{G}}$

SOAP5:. for all program points i, j and tag τ, if $j \in \mathsf{region}(i, \tau)$ (or $i = j$) and j is a return point then for all tag τ' such that $\mathsf{jun}(i, \tau')$ is defined, $\mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$.

Junction points uniquely delimits 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.

Figure 13 presents an example of safe cdr for an abstract transition system.

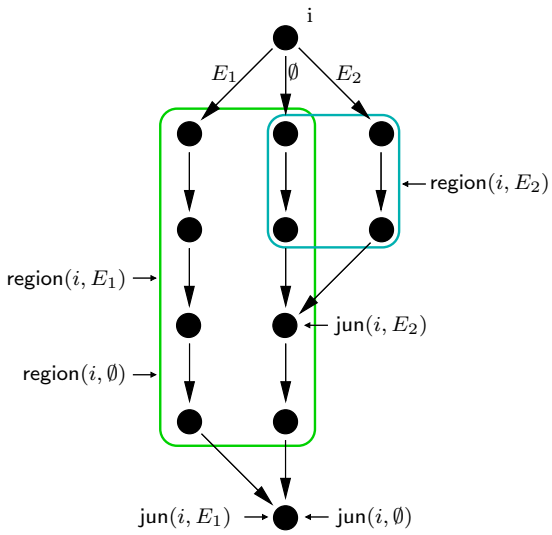

Fig. 13. Example of cdr in $\mathrm{JVM}_{\mathcal{G}}$. Only relevant tags are presented here.

7.2 Non-Interference

Method signatures are now of the form

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

where $\vec{k_v}$, k_h are defined as in $\mathsf{JVM}_{\mathcal{C}}$ but $\vec{k_r}$ (called *output level*) is now a list of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, where k_n is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i. In the rest of the paper we will write $\vec{k_r}[n]$ instead of k_n and $\vec{k_r}[e_i]$ instead of k_{e_i}.

This new notion of output level is associated with a new notion of *output indistinguishability*.

Definition 7.2.1 *Output indistinguishability.* Given a partial function $\beta \in \mathcal{LL} \rightharpoonup \mathcal{LL}$, an output level $\vec{k_r}$, indistinguishability of two final states in method m is defined

by the clauses:

$$\frac{h_1 \sim_\beta h_2 \qquad \vec{k_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_1(l_1)) : k \in \vec{k_r} \qquad k \leq k_{\text{obs}} \qquad l_1 \sim_\beta l_2}{(\langle l_1 \rangle, h_1) \sim_{\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_1(l_1)) : k \in \vec{k_r} \qquad k \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_2(l_2)) : k \in \vec{k_r} \qquad k \not\leq k_{\text{obs}}}{(v_1, h_1) \sim_{\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{\begin{array}{c} h_1 \sim_\beta h_2 \qquad \mathsf{class}(h_1(l_1)) : k_1 \in \vec{k_r} \qquad \mathsf{class}(h_2(l_2)) : k_2 \in \vec{k_r} \\ k_1 \not\leq k_{\text{obs}} \qquad k_2 \not\leq k_{\text{obs}} \end{array}}{(\langle l_1 \rangle, h_1) \sim_{\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

In each case, 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 $\vec{k_r}$. 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 $\vec{k_r}$ is low.

The previous definition and the next definition of non-interference rely on indistinguishability definitions already proposed for the JVM$_\mathcal{O}$ (*c.f.* page 20).

Definition 7.2.2 *Non-interferent* JVM$_\mathcal{G}$ *method.* A method m is *non-interferent* w.r.t. a policy $\vec{k_v} \longrightarrow \vec{k_r}$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $r_1, r_2 \in \mathcal{V} + \mathcal{L}$ such that $\rho_1, h_1 \Downarrow_m r_1, h_1'$, $\rho_2, h_2 \Downarrow_m r_2, h_2'$ and $h_1 \sim_\beta h_2$, $\rho_1 \sim_{\vec{k_v},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$ and $(r_1, h_1) \sim_{\beta',\vec{k_r}} (r_2, h_2)$.

Like in JVM$_\mathcal{C}$, we impose a *side-effect-safe* (*c.f.* page 25 for a formal definition) on methods. This notion is used when virtual call occur in a high context in order to enforce that no modification is done on low information during the execution of the called method.

Definition 7.2.3 *Safe* JVM$_\mathcal{G}$ *method.* A method m is *safe* w.r.t. a policy $\vec{k_v} \xrightarrow{k_h} \vec{k_r}$ if m is side-effect-safe with respect to k_h and m is non-interferent with respect to $\vec{k_v} \longrightarrow \vec{k_r}$.

Definition 7.2.4 *Safe* JVM$_\mathcal{G}$ *program.* 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 $\text{Policies}_\Gamma(\text{m})$[11].

7.3 Typing rules

Typing rules for JVM$_\mathcal{C}$ are extended (or sometimes modified) with rules given in Figure 14. These rules concern only exception-throwing and branching instructions. Rules for other instructions are as in JVM$_\mathcal{C}$.

Observe that the typing judgement is now parameterized by a tag $\tau \in \{\emptyset\} + \mathcal{C}$. It will be used to describe without ambiguity which typing constraint must be verified according to the kind of execution performed in the semantics.

This notion of tag requires to update the notion of *typable method*.

Definition 7.3.1 *Typable method.* 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 :

[11] $\text{Policies}_\Gamma(\text{m})$ has been defined in page 25.

$$\frac{P_m[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i, \emptyset),\ k \leq se(j')}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k(st)}$$

$$\frac{P_m[i] = \mathsf{return} \qquad k \sqcup se(i) \leq \vec{k_r}[n]}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow}$$

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

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

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

$$\frac{\begin{array}{c} P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i, \emptyset),\ k_2 \leq se(j) \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: st \Rightarrow \mathrm{lift}_{k_2} st}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i, \mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) = t \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow k_2 \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ k_2 \leq \vec{k_r}[\mathbf{np}] \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \emptyset),\ k \leq se(j)}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st)}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) = t}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} (k :: st \Rightarrow k \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i, \mathbf{np}),\ k \leq se(j) \qquad \mathsf{Handler}(i, \mathbf{np}) \uparrow \qquad k \leq \vec{k_r}[\mathbf{np}]}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ \forall j \in \mathsf{region}(i, e),\ k \leq se(j) \qquad \mathsf{Handler}(i, e) = t \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} k :: st \Rightarrow k \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ k \leq \vec{k_r}[e] \qquad \forall j \in \mathsf{region}(i, e),\ k \leq se(j) \qquad \mathsf{Handler}(i, e) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} k :: st \Rightarrow}$$

Fig. 14. Transfer rules for instructions of $\mathrm{JVM}_{\mathcal{G}}$

$\mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^\star$ such that $S_1 = \varepsilon$ and for all $i, j \in \mathcal{PP}$, $e \in \{\emptyset\} + \mathcal{C}$:

(1) $i \mapsto^e j$ implies there exists $st \in \mathcal{S}^\star$ such that $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow st$ and $st \sqsubseteq S_j$;
(2) $i \mapsto^e$ implies $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow$

7.4 A typable example

Figure 15 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 $m : (this : L,\ x : L,\ y : H) \xrightarrow{H} \{n : H,\ C : L,\ \mathbf{np} : H\}$ with the cdr (given only for branching points), the type stacks and the security environment given in Figure 15.

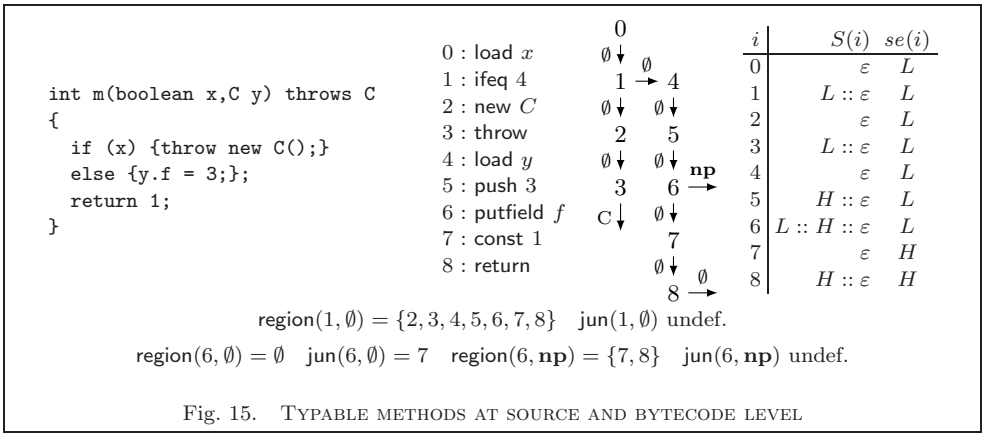

Fig. 15. Typable methods at source and bytecode level

Figure 16 gives another example[12] 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

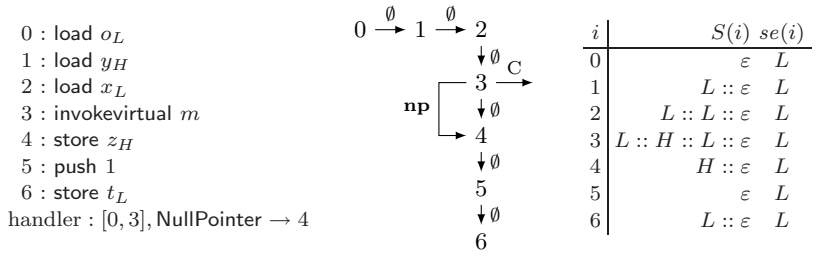

Fig. 16. Typable fragment with virtual call

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 $\vec{k_r}[\mathbf{np}] = H$ in the region $\mathsf{region}(3, \mathbf{np})$ (instead of $\mathsf{region}(3, C)$).

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

instr ::= ...		
	newarray t	create new array of element of type t in the heap
|	arraylength	get the length of an array
|	arrayload	load value from an array
|	arraystore	store value in array

Fig. 17. Additional instruction set for $\mathrm{JVM}_{\mathcal{A}}$

7.5 Type system soundness

Theorem 7.5.1. *Let P be a $\mathrm{JVM}_{\mathcal{G}}$ program, Γ a table of signatures, ft a global field policy, and for all method m in P, $(\mathsf{region}_m, \mathsf{jun}_m)$ a safe cdr for m (according to SOAP properties). Suppose all methods m in P are typable with respect to Γ, ft, region_m and to all signatures in $\mathrm{Policies}_\Gamma(\mathrm{m})$. Then P is safe with respect to Γ.*

Full proofs are given in Appendix D.

8. $\mathrm{JVM}_{\mathcal{A}}$: THE ARRAY-HANDLING EXTENSION OF $\mathrm{JVM}_{\mathcal{O}}$

The purpose of this section is to extend our analysis to arrays. To keep a digest level of detail we focus on a restricted JVM similar to $\mathrm{JVM}_{\mathcal{O}}$ (*i.e.* without exceptions or method calls) but the Coq development handles arrays directly on the complete $\mathrm{JVM}_{\mathcal{G}}$.

8.1 Programs, memory model and operational semantics

Programs. We first extend the set of instruction of $\mathrm{JVM}_{\mathcal{O}}$ programs. New instructions are presented in Figure 17.

States. $\mathrm{JVM}_{\mathcal{A}}$ extends $\mathrm{JVM}_{\mathcal{O}}$ semantic domains only at the heap level. Heaps now not only contains objects but also arrays. They are now modeled as partial functions $h : \mathcal{L} \rightharpoonup \mathcal{O} + \mathcal{A}$, where the set $\mathcal{A}$ of arrays is modeled as $\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}) \times \mathcal{PP}$, i.e. each array $a \in \mathcal{A}$ posses a length (noted $a.\mathsf{length}$), a partial function to access array values and a creation point. Strictly speaking, this last information is useless to present JVM semantics but we use it as a safe formalisation facility in the style followed by Freund and Mitchell to formalise Java bytecode object initialisation [Freund and Mitchell 2003]. Given a heap h, we note $\mathsf{dom}_{\mathcal{O}}(h)$ the set of location associated with an object in h and $\mathsf{dom}_{\mathcal{A}}(h)$ the set of location associated with an array.

Operational semantics. The operational semantics of the new instructions of $\mathrm{JVM}_{\mathcal{A}}$ is given in Figure 18. We only give here the normal execution cases but we should add all the cases where an exception is thrown (null pointer exception, array out-of-bound exception or wrong typed array store exception). Instruction $\mathsf{newarray}\ t$ pops a positive integer n from the operand stack to create a new initialized array and pushes the corresponding fresh location on top of the operand stack. The heap is updated with this new array including it length n and its creation point (m, i). Instruction $\mathsf{arraylength}$ pops a location l from the operand stack and pushes the length of the corresponding array. Instruction $\mathsf{arrayload}$ pops an index j and a location l from the operand stack. The content of the array at index j abd location l is fetched and pushed onto the operand stack. Instruction $\mathsf{arraystore}$ uses the top of the stack to update the array associated with the location in third position on the operand stack, at position pointed by the index in second position.

Successor relation. The successor relation is extended with the clause $i \mapsto i + 1$ for all new instructions.

$$\frac{P[i] = \mathsf{newarray}\ t \qquad l = \mathrm{fresh}(h) \qquad n \geq 0}{\langle i, \rho, n :: os, h\rangle \rightsquigarrow \langle i+1, \rho, l :: os, h \oplus \{l \mapsto (n, \mathsf{defaultArray}(n,t), i)\}\rangle}$$

$$\frac{P[i] = \mathsf{arraylength} \qquad l \in dom(h)}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, h(l).\mathsf{length} :: os, h\rangle}$$

$$\frac{P[i] = \mathsf{arrayload} \qquad l \in dom(h) \qquad 0 \leq j < h(l).\mathsf{length}}{\langle i, \rho, j :: l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, h(l)[j] :: os, h\rangle}$$

$$\frac{P[i] = \mathsf{arraystore} \qquad l \in dom(h) \qquad 0 \leq j < h(l).\mathsf{length}}{\langle i, \rho, v :: j :: l :: os, h\rangle \rightsquigarrow \langle i+1, \rho, os, h \oplus \{l \mapsto h(l) \oplus \{j \mapsto v\}\}\rangle}$$

Fig. 18. Operational Semantics for additional $\mathrm{JVM}_{\mathcal{A}}$ instructions

8.2 Non-Interference

The information flow type system we proposed for arrays follows the recommendations from Askarov and Sabelfeld [Askarov and Sabelfeld 2005] who argue that public arrays must be allowed to handle secret informations in order to achieve any realistic case study like the mental poker they have programmed in Jif [Myers 1999]. As a consequence the basic security level domain $\mathcal{S}$ of our previous type system is extended to a set $\mathcal{S}^{\mathrm{ext}}$ inductively defined by

$$\frac{k \in \mathcal{S}}{k \in \mathcal{S}^{\mathrm{ext}}} \qquad \frac{k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}}}{k[k_c] \in \mathcal{S}^{\mathrm{ext}}}$$

The idea is to use an extended level of the form $k[k_c]$ to represent the security level of an array reference, distinguishing the level k_c of the content of the array (which could be itself arrays) and k the level of the length of the array and the reference on the array. Simple level $k \in \mathcal{S}$ are still used to represent levels of object locations and numerical values. An extended level $k[k_c]$ is considered to be low (noted $k[k_c] \leq k_{\mathrm{obs}}$) if $k \leq k_{\mathrm{obs}}$.

Indistinguishability for $\mathrm{JVM}_{\mathcal{G}}$ states is extended and defined relative to a global mapping $\mathsf{at} : \mathcal{PP} \to \mathcal{S}^{\mathrm{ext}}$ that maps creation point of arrays to security levels for their contents. at will be left implicit in the rest of the paper. We will abusively note $\mathsf{at}(a)$ the level associates with the creation point of an array a. The definition of array indistinguishability says that two arrays are indistinguishable if they have the same length and if their contents are indistinguishable when their level is low.

Definition 8.2.1 *Array indistinguishability.* Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable with respect to a function $\beta \in \mathcal{LL} \rightharpoonup \mathcal{LL}$ if and only if $a_1.\mathsf{length} = a_2.\mathsf{length}$, $\mathsf{at}(a_1) = \mathsf{at}(a_2)$ and if $\mathsf{at}(a_1) \leq k_{\mathrm{obs}}$ implies, for all index i such that $0 \leq i < a_1.\mathsf{length}$, $a_1[i] \sim_\beta a_2[i]$.

Heap indistinguishability follows the definition given for $\mathrm{JVM}_{\mathcal{O}}$.

Definition 8.2.2 *Heap indistinguishability.* Two heaps h_1 and h_2 are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_\beta h_2$, if and only if:

— β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$;

— $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$;

— for every $l \in \mathsf{dom}(\beta)$, either $l \in \mathsf{dom}_{\mathcal{O}}(h_1) \cap \mathsf{dom}_{\mathcal{O}}(h_2)$ and $h_1(l) \sim_\beta h_2(\beta(l))$, or $l \in \mathsf{dom}_{\mathcal{A}}(h_1) \cap \mathsf{dom}_{\mathcal{A}}(h_2)$ and $h_1(l) \sim_\beta h_2(\beta(l))$;

The definition of non-interferent $\mathrm{JVM}_{\mathcal{A}}$ program is then exactly the same as for $\mathrm{JVM}_{\mathcal{O}}$ program.

$$\frac{P[i] = \mathsf{newarray}\ t \qquad k \in \mathcal{S}}{i \vdash k :: st \Rightarrow k[\mathsf{at}(i)] :: st}$$

$$\frac{P[i] = \mathsf{arraylength} \qquad k \in \mathcal{S} \qquad k_c \in \mathcal{S}^{\mathrm{ext}}}{i \vdash k[k_c] :: st \Rightarrow k :: st}$$

$$\frac{P[i] = \mathsf{arrayload} \qquad k_1, k_2 \in \mathcal{S} \qquad k_c \in \mathcal{S}^{\mathrm{ext}}}{i \vdash k_1 :: k_2[k_c] :: st \Rightarrow ((k_1 \sqcup k_2) \sqcup^{\mathrm{ext}} k_c) :: st}$$

$$\frac{P[i] = \mathsf{arraystore} \qquad ((k_2 \sqcup k_3) \sqcup^{\mathrm{ext}} k_1) \leq^{\mathrm{ext}} k_c \qquad k_2, k_3 \in \mathcal{S} \qquad k_1, k_c \in \mathcal{S}^{\mathrm{ext}}}{i \vdash k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow st}$$

Fig. 19. Additional typing transfer rules for $\mathrm{JVM}_{\mathcal{A}}$

8.3 Typing rules

Figure 8.3 presents the typing transfer rules for $\mathrm{JVM}_{\mathcal{A}}$ specific instructions.

— The transfer rule for newarray creates a new security level for the new created array, combining the length level k length and the content level $\mathsf{at}(i)$.

— The transfer rule for arraylength only uses the length level k of the extended level $k[k_c]$ found on top of the stack type to give a security level to the length of an array.

— The transfer rule for arrayload pushes on top of the stack a security level $(k_1 \sqcup k_2) \sqcup^{\mathrm{ext}} k_c$. $\sqcup^{\mathrm{ext}} \in \mathcal{S} \times \mathcal{S}^{\mathrm{ext}} \to \mathcal{S}^{\mathrm{ext}}$ is defined by $k' \sqcup^{\mathrm{ext}} k = k' \sqcup k$ when $k, k' \in \mathcal{S}$ and $k' \sqcup^{\mathrm{ext}} k[k_c] = (k' \sqcup k)[k_c]$ when $k, k' \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\mathrm{ext}}$. Here k_1 allows to prevent implicit flows trough a high index and k_2 trough alias.
The following example illustrates this first point. It corresponds to a source program like

$$\mathtt{int}\ \mathtt{x_L}\ =\ \mathtt{a_{L[L]}[i_H]};$$

Let $\mathtt{x_L}$ be a low variable, $\mathtt{a_{L[L]}}$ a low array variable (both for reference and content levels) and $\mathtt{i_H}$ a high integer variable.

$$\begin{array}{l}\mathsf{load}\ a_{L[L]}\\ \mathsf{load}\ i_H\\ \mathsf{arrayload}\ y_H\\ \mathsf{store}\ x_L\end{array}$$

In this program, if the low array $\mathtt{a_{L[L]}}$ only contents distinct elements, an attacker could learn the value of $\mathtt{i_H}$ by looking at the result of $\mathtt{a_{L[L]}[i_H]}$. This program is rejected by our type system because $\mathtt{a_{L[L]}[i_H]}$ receives a type H in the arrayload rule and storing a high value in a low variable is impossible thanks to the store rule.
The second point corresponds to an access $\mathtt{a_{H[L]}[i_L]}$ where $\mathtt{a_{H[L]}}$ may be either aliased to a low array $\mathtt{a^0_{L[L]}}$ containing only the 0 integer or aliased to a low array $\mathtt{a^1_{L[L]}}$ containing only the 1 integer. Hence observing the value of $\mathtt{a_{H[L]}[i_L]}$ would allow an attacker to know which of this array is aliased to $\mathtt{a_{H[L]}}$.

— The transfer rule for arraystore uses the partial order $\leq^{\mathrm{ext}}$ on $\mathcal{S}^{\mathrm{ext}}$ which is inductively defined by

$$\frac{k \leq k' \quad k, k' \in \mathcal{S}}{k \leq^{\mathrm{ext}} k'} \qquad \frac{k \leq k' \quad k, k' \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}}}{k[k_c] \leq^{\mathrm{ext}} k'[k_c]}$$

The rule constrains k_1 and k_c to prevent an explicit flow from a high value to an array declared with a low content. It constrains also k_2 and k_c to prevent information leak by updating a low array content with a high index. Without it, an assignment of the form $\mathtt{a_{L[L]}[i_H] = 1}$ in a low array $\mathtt{a_{L[L]}}$ only containing the 0 integer would allow to reveal the value of $\mathtt{i_H}$.

Finally, the constraint between k_3 and k_c prevents modifying low array contents of high array reference that may be alias to a low array. This is for example necessary if an array $\mathtt{a}_{\mathtt{H[L]}}$ may be aliased to two distinct low array $\mathtt{a}^0_{\mathtt{L[L]}}$ and $\mathtt{a}^1_{\mathtt{L[L]}}$. Observing which of these low arrays is modified by side effect of the affectation $\mathtt{a}_{\mathtt{H[L]}}[\mathtt{i}_{\mathtt{L}}] = \mathtt{v}_{\mathtt{L}}$ would allow an attacker to learn which of these arrays is effectively equal to $\mathtt{a}_{\mathtt{H[L]}}$.

8.4 Type system soundness

THEOREM 8.4.1. *Let P be a* $\mathrm{JVM}_{\mathcal{A}}$ *program and* $(\mathsf{region}, \mathsf{jun})$ *a safe cdr for P (according to SOAP properties). Suppose P is typable with respect to* region *and to a signature $\vec{k_v} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\vec{k_v} \longrightarrow k_r$.*

This soundness proof is similar to the previous proofs.

9. RELATION WITH INFORMATION FLOW TYPE SYSTEMS FOR JAVA

While the development of information flow checkers for Java bytecode has received relatively little attention, there has been substantial interest in building information flow type systems for Java. One of the earliest and still most active effort in this direction is Jif, an extension of Java with information flow types developed by Myers and co-workers. Jif builds upon the decentralized label model and offers a flexible and expressive framework to define information flow policies. There are some commonalities between our type system and the type system of Jif:

— the form of method signatures,
— the use of pre-analyses to reduce the control flow graph,
— the support for modular verification,
— public arrays are able to handle secret informations.

On the other hand, there are many features of Jif that are missing in our work; the most prominent such features are:

— declassification: often non-interference is too strict to characterize confidentiality policies where private information needs to be released at some point in the program. Jif has support for declasification policies though a language construct, namely declassify.

— label polymorphism: Jif implements label polymorphism, that allows methods with generic information flow types, to be invoked in different contexts.

The rich set of features supported by Jif has proved useful in realistic case studies such as an implementation of mental poker [Askarov and Sabelfeld 2005], but makes it difficult to prove that the information flow type system is sound.

Banerjee and Naumann [Banerjee and Naumann 2005] develop a provably sound information flow type system for a fragment of Java with objects and methods. The type system is simpler than Jif since they omit language features such as exceptions and arrays, and do not provide mechanism for declassification. Their type system has been formally verified in PVS [Naumann 2005], and [Sun et al. 2004] present a type inference algorithm that dispenses users of writing all security annotations.

The above discussion illustrates similarities and differences between our type system for bytecode and previously defined information flow type systems for Java. One interesting question is whether source Java programs that are accepted by an information flow type system are compiled into bytecode programs that are accepted by our checker. Such a type-preserving compilation result is interesting from a theoretical perspective, since it allows to derive soundness of information flow type systems for Jif-like languages from our results, and also from a practical perspective, since it shows that our type system is sufficiently expressive to accept compiled

versions of the examples of [Banerjee and Naumann 2005], and more generally that (a useful fragment of) the experimental programming language JFlow [Myers 1999] can be used to develop information-flow aware applications that are accepted by our type system.

In [Barthe et al. 2006; Rezk 2006] it is shown that source and bytecode type systems are related. That is, a standard (non-optimizing) Java compiler will translate programs that are typable in a Jif-like information-flow type system to programs typable in our system.

To illustrate type-preserving compilation, we give an example of a JFlow typable program, and show that its compiled bytecode is typable in our system. The commented lines are the intermediate types needed to infer the whole program type $\{n : L; nv : H; rv : L; C : L; np : L\}$. The numbering before the types will relate to the program points of the compiled code, given later.

try{	(0)$\{\underline{n} : L; nv : L; rv : L; C : L; np : L\}$
$int\ z_H = o_H.m(x_L, y_H);$	(4)$\{n : L; nv : H; rv : L; C : L; np : H\}$
}catch($NullPointerException$){	
$x_H = 1;$	(7)$\{n : H; nv : H; rv : L; C : L; np : L\}$
};	(7)$\{n : L; nv : H; rv : L; C : L; np : L\}$
$int\ t_L = 1;$	
	(8)$\{n : L; nv : H; rv : L; C : L; np : L\}$

We recall that in JFlow, the symbol $\underline{n}$ represents normal termination of the program, the symbol $\underline{nv}$ and $\underline{rv}$ represent the labels of the normal value of an expression and the return value of a statement, respectively. Notice that we omit symbol r in the types because it is only relevant for return statements.

The compiled code for the above program and its type is given in Figure 20.

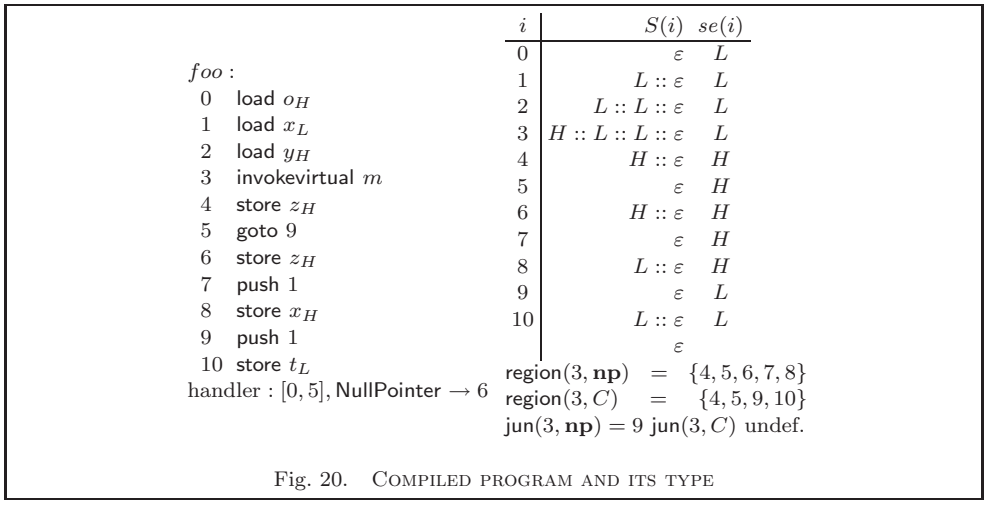

Fig. 20. Compiled program and its type

We informally show how the JFlow types translates into the bytecode type for its compiled counterpart. That is, we show a connection between a JFlow type of the form $\{n : L; nv : H; rv : L; C : L; np : L\}$ and a security envirommment se and a stack type st for bytecode.

For the type in line (4), the security type in $\underline{nv}$ is equal to the type on top of the stack type. Notice that the type of the method call in line (4) includes two paths for exceptions, namely C and **np**, of type L and H respectively. At bytecode levels, this is reflected in the control dependence regions of C and **np** for instruction 4.

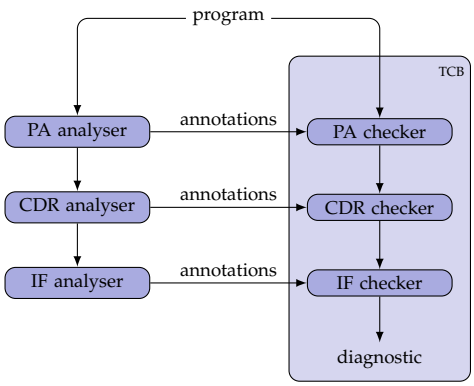

Fig. 21. Information flow analyser and checker

In fact, we can observe that in all security enviroments of successor instructions of the invokevirtual, region$(3, \mathbf{np})$ and region$(3, C)$ are constrained. In particular for all program points i in region$(3, \mathbf{np})$, security enviroment at i is high. In line (7), the JFlow type $\underline{\mathbf{np}}$ is lowered to L, because of the try-catch typing rule. At bytecode level, we have that the junction point of region$(3, \mathbf{np})$ corresponds to the point just after the catch. Hence the typing rule for invokevirtual does not constrain this program point thought the security enviroment ($se(9)$ is low). The symbol $\underline{n}$ in a JFlow type can be seen as $se(i)$ level in the bytecode code, where i is an instructions that belongs to the compilation of the command of the type.

10. MACHINE-CHECKED PROOF

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 we have machine checked their soundness in the proof assistant Coq. The purpose of this section is to provide an overview of the architecture of the proof, and more importantly, to discuss the benefits of using formal proofs.

10.1 Lightweight verification

The complexity of the analysis is a strong argument for adopting ideas of lightweight verification, and to only require consumers to perform checking, as described on the right part of Figure 21. Here it is assumed that programs are annotated with (part of) the results of the PA, CDR, and IF analysers, that are performed by the code producer on the left hand part of the Figure:

(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 3. Its correctness relies on the correctness of the PA checker;
(3) the IF checker verifies type correctness in the style of lightweight bytecode verification. In a nutshell, the idea is to check whether the types computed at junction points are compatible with the declared type in the certificate, and to follow the analysis with the latter if it is the case. As opposed to a dataflow

	Line of code
JVM semantics (Bicolano), bytecode program manipulation tools	4287
Non-Interference type checker	
General non-interference proof	942
Unwinding lemmas	3527
Typing rules (definitions, properties, checker)	5236
Indistinguishability	2157
CDR checker	1003
Total	17152

Fig. 22. Size of the Coq development

analysis, which performs a fixpoint computation by repeating iterations over the program, the IF checker performs the analysis in one pass.

10.2 Structure of the formal development

The whole Coq development is about 17,000 lines of definitions and proofs. Figure 22 presents the repartition of the development size. The core of the work is the information flow type checker but the formal definition of the JVM semantics is in itself a big piece of code. Is it available at

`http://www.irisa.fr/lande/pichardie/iflow/`

We have formalised in Coq several predicates:

(1) the security condition as SAFE;
(2) the correctness of program annotations as PA;
(3) the SOAP properties as CDR (given in Section 3);
(4) the information flow type checker as IF based on the notion of typable program.

We have machine-checked the following theorem which corresponds to the two first items of Theorem 2.4.1.

THEOREM 10.2.1.

(*1*) CDR *and* IF *are decidable predicates.*
(*2*) *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. What is left for future work is to define a decidable predicate $\mathsf{check}_{\mathrm{PA}}$ that entails PA.

The function $\mathsf{check}_{\mathrm{IF}}$ uses lightweight bytecode verification techniques. It relies on a formal semantics of the JVM in Coq, called Bicolano[13], which has been developed within the Mobius project to serve as a common basis for certification of proof carrying code technologies in Coq.

Bicolano closely follows the official JVM specification—although some features are omitted, e.g. initialization, subroutines (which shall soon disappear), multi-threading, dynamic class loading, garbage collection, 64-bit arithmetic and floats. As a consequence, the dynamic behavior of a bytecode program is described in term of an elementary small-step relation $\cdot \rightarrow \cdot$ between states of the virtual machine. In Bicolano, execution traces can terminate normally, diverge, or get stuck. It is the role of (standard) bytecode verification to ensure progress, and thus to eliminate programs whose execution may get stuck.

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

The basic layer of Bicolano is complemented by additional layers that are used between the non interference theorem and its proofs, as shown in Figure 23.

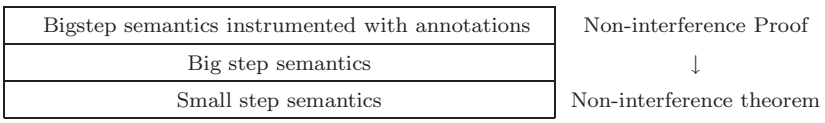

Fig. 23. The several semantic levels of the formalisation

The first layer is an adaptation of the small-step semantics where method calls are performed in one step, as it has been presented in Section 6. Figure 24 presents the small-step semantics rule for method calls and return which must be compared to the big step semantics rule for virtual method invocation presented in Figure 9 (page 24). While small-step semantics uses a call stack to store the calling context and retrieve it during a return instruction, the big step semantics directly calls the full evaluation of the called method from an initial state to a return value and uses it to continue the current computation.

$$\frac{\begin{array}{c}P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ f' = [m', 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \varepsilon] \qquad f'' = [m, \mathsf{pc}, \rho, os_2]\end{array}}{\langle h, [m, \mathsf{pc}, \rho, l :: os_1 :: os_2], sf\rangle \rightarrow \langle h, f', f'' :: sf\rangle}$$

$$\frac{\mathrm{instrAt}(m, \mathsf{pc}, \mathrm{return})}{\langle h, [m, \mathsf{pc}, \rho, v :: os], [m', \mathsf{pc}', \rho', os'] :: sf\rangle \rightarrow \langle h, [m', \mathsf{pc}' + 1, \rho', v :: os'], sf\rangle}$$

Fig. 24. Small-step semantics rule for virtual method call

The alternative semantics is used in the proofs, but not in the final theorem where we have used the equivalence between the two semantics to derive non-interference for the small-step semantics. More formally, to prove non-interference for execution expressed with the small-step semantics we prove that each iterative execution of the JVM to a final value implies the corresponding judgment of the big step semantics.

$$\begin{array}{c}\left(\begin{array}{c}\langle h, [m, \mathsf{pc}, \rho, os], \varepsilon\rangle \rightarrow^* \langle h', [m, \mathsf{pc}', \rho', v' :: os'], \varepsilon\rangle \\ \text{with } P_m[\mathsf{pc}'] = \mathsf{return}\end{array}\right) \\ \Longrightarrow \\ \langle h, \mathsf{pc}, l, s\rangle \Downarrow_m (h', v')\end{array} \tag{1}$$

A similar result is necessary for execution terminating with an uncaught exception.

Using this alternative semantics has brought a significant simplification w.r.t. the (unpublished) proof of [Barthe and Rezk 2005]: having only one frame, the notion of state equivalence has been greatly simplified, especially concerning the indistinguishability relation previously required for call stacks. The proofs are consequently far easier to manage and to understand. However, a well-known defect of the alternative semantics is that it is not immediate to extend our results to multi-threading [Barthe et al. 2007].

The proof of non-interference relies on an instrumented semantics of annotated programs. In such an instrumented semantics, extra properties taken from annotation information are assumed in the premise of the transition rules. Figure 25 gives an example of instrumented transition. Annotations take the form of flags safe attached to program points where the pre-analyser predict that no exception may be thrown here. Exception hence only happens at program points which are not annotated as safe. Assuming the annotations are correct, it is straightforward

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad l' = \mathrm{fresh}(h) \qquad \boxed{\mathsf{annnot}_m[i] \neq \mathsf{safe}}}{\langle i, \rho, null :: os, h\rangle \overset{(0)}{\rightsquigarrow}_{m,\mathbf{np}} \mathsf{RuntimeExceptionHandling}(h, l', \mathbf{np}, i, \rho)}$$

Fig. 25. Example of annotated semantic rule

to prove that each judgment of the big step semantics implies the corresponding judgment of the instrumented big step semantics.

$$\langle h, \mathsf{pc}, l, s\rangle \Downarrow_m (h', v') \wedge \mathrm{Sound}(\mathrm{annot}) \Longrightarrow \langle h, \mathsf{pc}, l, s\rangle \Downarrow_m^{\mathrm{annot}} (h', v') \qquad (2)$$

10.3 Benefits of using formal proofs

One evident benefit of formal proofs is to increase confidence in the correctness of the type system. The need for machine-checked proofs is particularly important here because non-interference proofs are particularly involved (w.r.t. say standard type safety proofs discussed in [Aydemir et al. 2005]), and because of the complexity of the fragment of the JVM considered. For example, 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* [Appel and Felty 2000] (FPCC), since the Trusted Computed Base is here relegated to the Coq type checker and the formal definition of non-interference. Figure 26 presents this TCB, updating the previous scheme of Figure 21 where formal proofs were not mentioned. 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.

Of course, 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* [Besson et al. 2006], 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.

11. RELATED WORK

We refer to the survey article of Sabelfeld and Myers [Sabelfeld and Myers 2003] for a more complete account of recent developments in language-based security, and only focus on related work that deals with low-level languages, or develop ideas that are relevant to consider in future work.

For convenience, we separate related work between works that deal with typed assembly languages, and higher-order low-level languages and finally with the JVM and Java. Then, we focus on issues of concurrency and information release, which are not considered in the present work.

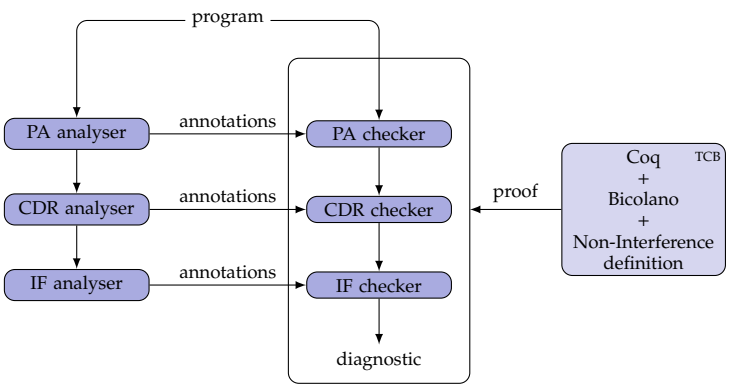

Fig. 26. Information flow analyser and checker with Coq TCB

11.1 Typed assembly languages

The idea of typing low-level programs and ensuring that compilation preserves typing is not original to information flow, and has been investigated in connection with type-directed compilation. Morrisett, Walker, Crary and Glew [Morrisett et al. 1999] develop a typed assembly language (TAL) based on a conventional RISC assembly language, and show that typable programs of System F can be compiled into typable TAL programs.

The study of non-interference for typed assembly languages has been initiated by Medel, Bonelli, and Compagnoni [Bonelli et al. 2005], who developed a sound information flow type system for a simple assembly language called SIFTAL. A specificity of SIFTAL is to introduce pseudo-instructions that are used to enforce structured control flow using a stack of continuations; more concretely, the pseudo-instructions are used to push or retrieve linear continuations from the continuation stack. Unlike the stack of call frames that is used in the JVM to handle method calls, the stack of continuations is used for control flow within the body of a method. The use of pseudo-instructions allows to formulate global constraints in the type system, and thus to guarantee non-interference. More recent work by the same authors [Medel et al. 2005] and by Yu and Islam [Yu and Islam 2006] avoids the use of pseudo-instructions. In addition, Yu and Islam consider a richer assembly language and prove type-preserving compilation for an imperative language with procedures.

11.2 Higher-order low-level languages

Zdancewic and Myers [Zdancewic and Myers 2002] develop a sound information flow type system for a CPS calculus that uses linear continuations and prove type-preservation for a linear CPS translation from an imperative higher-order language inspired from SLAM [Heintze and Riecke 1998] to their CPS language, providing thus one early type-preservation result for information flow. The use of linear continuations in the CPS translation is essential to guarantee type-preserving compilation.

In a similar line of work, Honda and Yoshida [Honda and Yoshida 2002] develop a sound information flow type system for the π-calculus and prove type-preserving compilation for the Dependency Core Calculus [Abadi et al. 1999] and for an imperative language inspired from Volpano and Smith [Volpano and Smith 1997]. Furthermore, they derive soundness of the source type systems from the soundness

of the type system for the π-calculus. As in the work of Zdancewic and Myers, linearity is used crucially to ensure that the compilation is type-preserving.

11.3 JVM

Lanet *et al.* [Bieber et al. 2002] provide an early study of information flow the JVM. Their method consists in specifying in the SMV model checker an abstract transition semantics of the JVM that manipulates security levels, and that can be used to verify that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their method is directed towards smart card applications, and thus only covers a sequential fragment of the JVM. While their method has been used successfully to detect information leaks in a case study involving multi-application smartcards, it is not sup ported by any soundness result. In a series of papers initiating with [Bernardeschi and Francesco 2002], Bernardeschi and co-workers also propose to use abstract interpretation and model-checking techniques to verify secure information.

The fist information flow type system for a low level language was proposed by Kobayashi and Shirane [Kobayashi, Shirane 2002] for a subset of the JVM similar to what we called $\mathrm{JVM}_{\mathcal{I}}$ in this paper. In a predecessor to this work, Barthe, Basu and Rezk [Barthe et al. 2004] propose a sound information flow type system for a simple assembly language that closely resembles the $\mathrm{JVM}_{\mathcal{I}}$ fragment of this paper, and show type-preserving compilation for the imperative language and type system of [Volpano and Smith 1997]. Later, Barthe and Rezk [Barthe and Rezk 2005] extend this work to a language with objects and a simplified treatment of exceptions, and Barthe, Naumann and Rezk [Barthe et al. 2006] show type-preserving compilation for a Java-like language with objects and a simplified treatment of exceptions.

Genaim and Spoto [Genaim and Spoto 2005] have shown how to represent information flow for Java bytecode through boolean functions; the representation allows checking via binary decision diagrams. Their analysis is fully automatic and does not require that methods are annotated with security signatures, but it is less efficient than type checking.

11.4 Java

The relation to [Myers 1999] and [Banerjee and Naumann 2005] has already been discussed in a previous section, so we focus on other relevant work.

As ours, the type systems of [Myers 1999] and of [Banerjee and Naumann 2005] rely on the assumption 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. However, Hedin and Sands [Hedin and Sands 2006] have recently observed that the assumption is unvalidated by methods from the Java API, and exhibited a Jif program that does not use declassification but leaks information through invoking API methods. Their attack relies on the assumption that the function that allocates new objects on the heap is deterministic; however, this assumption is perfectly reasonable and satisfied by many implementations of the JVM. In addition to demonstrating the attack, Hedin and Sands show how a refined information flow type system can thwart such attacks for a language that allows to cast references as integers. Intuitively, their type system tracks the security level of references as well as the security levels of the fields of the object its points to.

Hammer, Krinke and Snelting [Hammer et al. 2006] have developed an information flow analysis based on control dependence regions; they use path conditions to achieve precision in their analysis, and to exhibit security leaks if the program is insecure. Their approach is automatic and flow-sensitive, but less efficient than type-based approach.

11.5 Logical analysis of non-interference for Java

In a different line of work, several authors have investigated the use of program logics to enforce non-interference of Java programs [Barthe et al. 2004; Beringer and Hofmann 2007; Darvas et al. 2005; Terauchi and Aiken 2005]. One possible encoding is based on the idea of self-composition, where the program is composed with a renaming of itself to ensure properties that involve two executions of a program. The idea of self-composition has also been put in practice by Dufay and co-workers [Dufay et al. 2005], who used an extension of the Krakatoa tool [Marché et al. 2004] with self-composition primitives to verify that data mining programs from the open source repository WEKA adhere to privacy policies cast in terms of information flow. Both [Darvas et al. 2005; Dufay et al. 2005] are application-oriented and do not attempt to provide a theoretical study of self-composition for Java. In a recent article, Naumann [Naumann 2006] sets out the details of self-composition in presence of a dynamically allocated heap; in short, one main issue tackled by Naumann is the definition of a meaningful notion of "renaming" for the heap.

Independently, Banerjee and his co-workers [Amtoft et al. 2006] develop a logic that allows to verify non-interference without resorting to self-composition. The logic, which is tailored to object-oriented languages, handles the heap using independence assertions inspired from separation logic.

11.6 Concurrency

Extending information flow type systems to concurrent languages is notoriously difficult because the parallel composition of secure sequential programs may itself not be secure [Smith and Volpano 1998]. The problem is caused by so-called internal timing leaks that arise when secret information is revealed through the scheduling of threads. In order to avoid internal timing leaks, many works on information flow type systems for concurrent languages focus on a stronger notion of non-interference that considers intermediate execution steps of programs. Thus, information flow type systems for concurrent languages typically enforce bisimulation-based notions of non-interference, at the cost of being very conservative, e.g. by rejecting programs that contain a loop with a high guard, or that perform a low assignment after a high branching statement, see e.g. [Almeida Matos 2006; Boudol and Castellani 2002]. Another approach consists in wrapping high branching statements in a protect primitive that forces the execution of the branching statement to be atomic; however, it is not clear how to implement such a primitive.

Motivated by the desire to provide flexible and practical enforcement mechanisms for concurrent languages, Russo and Sabelfeld [Russo and Sabelfeld 2006] develop a sound information flow type system that enforces termination-insensitive non-interference in a concurrent setting. The originality of their approach resides in constraining the behavior of the scheduler so as to avoid internal timing leaks. More precisely, Russo and Sabelfeld require that the scheduler does not pick any low thread for execution as long as a one thread is executing within a high branching statement. Their work focus on a simple imperative language extended with two commands that provide directives to the compiler. In a follow-up to the present work, Barthe, Rezk, Russo and Sabelfeld [Barthe et al. 2007] have adapted their approach to bytecode languages, and thus provided the first provably secure information flow checker for concurrent bytecode. While the proof deals with the $\mathrm{JVM}_{\mathcal{I}}$, we believe that it is possible to generalize our results to the $\mathrm{JVM}_{\mathcal{G}}$.

11.7 Declassification

Information flow type systems have not found substantial applications in practice, in particular because information flow policies based on non-interference are too

rigid and do not authorize information release. In contrast, many applications often release deliberately some amount of sensitive information. Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases. In a recent survey [Sabelfeld and Sands 2005], A. Sabelfeld and D. Sands identify four dimensions of declassification: what, when, where, and who, and provide a classification of the declassification policies found in the signatures along these dimensions. Handling declassification policies that combine several dimensions is an important next step towards applicability of information flow type systems, and of our type system in particular.

12. CONCLUSION

We have introduced a provably sound information flow type system for a fragment of the JVM that includes objects, methods, exceptions, and arrays. To our best knowledge, no previous work has provided a sound type system for such an expressive fragment of the sequential JVM. In combination with our work on type-preserving compilation, our results provide a sound basis for end-to-end security solutions for Java-based mobile code. An important goal is to extend our results to multi-threaded Java, which is prominent in mobile code. To this end, we intend to build upon the proposal of Russo and Sabelfeld [Russo and Sabelfeld 2006], and its adaptation to bytecode [Barthe et al. 2007]. Besides, we intend to provide support for declassification, in order to be able to type-check realistic case studies. More generally, extending our results to type systems that support dynamic policies and label polymorphism could significantly contribute to the applicability of our type system.

REFERENCES

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. 1999. A core calculus of dependency. In *Proceedings of POPL'99*. ACM Press, 147–160.

Almeida Matos, A. 2006. Typing secure information flow: declassification and mobility. Ph.D. thesis, Ecole Nationale Supérieure des Mines de Paris.

Amtoft, T., Bandhakavi, S., and Banerjee, A. 2006. A logic for information flow in object-oriented programs. In *Proceedings of POPL'06*. ACM Press, 91–102.

Appel, A. and Felty, A. 2000. A semantic model of types and machine instuctions for proof-carrying code. In *Proceedings POPL'00*. ACM Press, 243–253.

Askarov, A. and Sabelfeld, A. 2005. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of ESORICS'05*, S. D. C. di Vimercati, P. Syverson, and D. Gollmann, Eds. Lecture Notes in Computer Science, vol. 3679. Springer-Verlag, 197–221.

Aydemir, B., Bohannon, A., Fairbairn, M., Foster, J., Pierce, B., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings of TPHOLs'05*, J. Hurd and T. Melham, Eds. Lecture Notes in Computer Science, vol. 3603. Springer-Verlag, 50–65.

Banerjee, A. and Naumann, D. 2005. Stack-based access control for secure information flow. *Journal of Functional Programming 15*, 131–177.

Barthe, G., Basu, A., and Rezk, T. 2004. Security types preserving compilation. In *Proceedings of VMCAI'04*, B. Steffen and G. Levi, Eds. Lecture Notes in Computer Science, vol. 2934. Springer-Verlag, 2–15.

Barthe, G., Basu, A., and Rezk, T. 2007. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*. To appear.

Barthe, G., D'Argenio, P., and Rezk, T. 2004. Secure Information Flow by Self-Composition. In *Proceedings of CSFW'04*, R. Foccardi, Ed. IEEE Press, 100–114.

Barthe, G., Naumann, D., and Rezk, T. 2006. Deriving an Information Flow Checker and Certifying Compiler for Java. In *Symposium on Security and Privacy, 2006*. IEEE Press.

Barthe, G., Pichardie, D., and Rezk, T. 2007. A certified lightweight non-interference java bytecode verifier. In *European Symposium on Programming, ESOP'07*, R. D. Niccola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer-Verlag, 125-140.

Barthe, G. and Rezk, T. 2005. Non-interference for a JVM-like language. In *Proceedings of TLDI'05*, M. Fähndrich, Ed. ACM Press, 103–112.

Barthe, G., Rezk, T., Russo, A., and Sabelfeld, A. 2007. Security of multithreaded programs by compilation. Manuscript.

Barthe, G. and Serpette, B. 1999. Partial evaluation and non-interference for object calculi. In *Proceedings of FLOPS'99*, A. Middeldorp and T. Sato, Eds. Lecture Notes in Computer Science, vol. 1722. Springer-Verlag, 53–67.

Beringer, L. and Hofmann, M. 2007. Secure information flow and program logics. In *Proceedings of CSF'07*. IEEE Computer Society Press.

Bernardeschi, C. and Francesco, N. D. 2002. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In *Proceedings of VMCAI'02*, A. Cortesi, Ed. Lecture Notes in Computer Science, vol. 2294. 1–15.

Besson, F., Jensen, T., and Pichardie, D. 2006. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*. To appear.

Bieber, P., Cazin, J., Wiels, V., Zanon, G., Girard, P., and Lanet, J.-L. 2002. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security 10*, 369–398.

Bonelli, E., Compagnoni, A., and Medel, R. 2005. Information flow analysis for a typed assembly language with polymorphic stacks. In *Proceedings of CASSIS'05*, G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, Eds. Lecture Notes in Computer Science, vol. 3956. Springer-Verlag, 37–56.

Boudol, G. and Castellani, I. 2002. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science 281,* 1, 109–130. Preliminary version available as INRIA Research report 4254.

Coq Development Team. 2004. *The Coq Proof Assistant User's Guide. Version 8.0*.

Darvas, A., Hähnle, R., and Sands, D. 2005. A theorem proving approach to analysis of secure information flow. In *Proceedings International Conference on Security in Pervasive Computing*, D. Hutter and M. Ullmann, Eds. Lecture Notes in Computer Science, vol. 3450. Springer-Verlag, 193–209. Preliminary version in the informal proceedings of WITS'03.

Dufay, G., Felty, A., and Matwin, S. 2005. Privacy-sensitive information flow with JML. In *Proceedings of CADE'05*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, vol. 3632. Springer-Verlag, 116–130.

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

Genaim, S. and Spoto, F. 2005. Information Flow Analysis for Java Bytecode. In *Proceedings of VMCAI'05*, R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer-Verlag, 346–362.

Girard, P. 1999. Which security policy for multiapplication smart cards? In *Proceedings of Smartcard'99*. USENIX Association.

Hammer, C., Krinke, J., and Snelting, G. 2006. Information flow control for java based on path conditions in dependence graphs. In *Proceedings of ISSSE'06*. IEEE Computer Society Press, 87–96.

Hedin, D. and Sands, D. 2006. Noninterference in the presence of non-opaque pointers. In *Proceedings of CSFW'06*. IEEE Computer Society Press, 255–269.

Heintze, N. and Riecke, J. 1998. The SLam calculus: programming with secrecy and integrity. In *Proceedings of POPL'98*. ACM Press, 365–377.

Honda, K. and Yoshida, N. 2002. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*. ACM Press, 81–92.

Kobayashi, N. and Shirane, K. 2002. Type-Based Information Analysis for Low-Level Languages. In *Proceedings of APLAS'02*. 302-316

Leroy, X. 2002. Bytecode verification on Java smart cards. *Software–practice and experience 32,* 4 (Apr.), 319–340.

Mantel, H. and Sabelfeld, A. 2003. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security 11,* 4, 615–676.

Marché, C., Paulin-Mohring, C., and Urbain, X. 2004. The Krakatoa tool for certification of Java/JavaCard Programs annotated with JML Annotations. *Journal of Logic and Algebraic Programming 58*, 89–106.

Medel, R., Compagnoni, A., and Bonelli, E. 2005. A typed assembly language for non-interference. In *Proceedings of ICTCS 2005*, M. Coppo, E. Lodi, and G. Pinna, Eds. Lecture Notes in Computer Science, vol. 3701. Springer-Verlag, 360–374.

Montgomery, M. and Krishna, K. 1999. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21,* 3 (Nov.), 527–568.

Myers, A. 1999. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*. ACM Press, 228–241.

Naumann, D. 2005. Verifying a secure information flow analyzer. In *Proceedings of TPHOLs'05*, J. Hurd and T. Melham, Eds. Lecture Notes in Computer Science, vol. 3603. Springer-Verlag, 211–226. Preliminary version appears as Report CS-2004-10, Stevens Institute of Technology, 2003.

Naumann, D. 2006. From coupling relations to mated invariants for checking information flow (extended abstract). In *Proceedings of ESORICS'06*, D. Gollmann and A. Sabelfeld, Eds. Lecture Notes in Computer Science, vol. 3xxx. Springer-Verlag, xxx–xxx. To appear.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*. Springer-Verlag.

O'Neill, K. R., Clarkson, M. R., and Chong, S. 2006. Information-flow security for interactive programs. In *Proceedings of CSFW'06*. IEEE Computer Society, 190–201.

Pottier, F. and Simonet, V. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems 25,* 1 (Jan.), 117–158.

Rezk, T. 2006. Verification of confidentiality policies for mobile code. Ph.D. thesis, Université de Nice Sophia-Antipolis.

Russo, A. and Sabelfeld, A. 2006. Securing interaction between threads and the scheduler. In *Proceedings of CSFW'06*.

Sabelfeld, A. and Myers, A. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications 21*, 5–19.

Sabelfeld, A. and Sands, D. 2005. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press.

Smith, G. and Volpano, D. 1998. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*. ACM Press, 355–364.

Sun, Q., Banerjee, A., and Naumann, D. 2004. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of SAS'04*, R. Giacobazzi, Ed. Lecture Notes in Computer Science, vol. 3148. Springer-Verlag, 84–99.

Terauchi, T. and Aiken, A. 2005. Secure information flow as a safety problem. In *Proceedings of SAS'05*, C. Hankin and I. Siveroni, Eds. Lecture Notes in Computer Science, vol. 3672. Springer-Verlag, 352–367.

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

Yu, D. and Islam, N. 2006. A typed assembly language for confidentiality. In *Proceedings of ESOP'06*, P. Sestoft, Ed. Lecture Notes in Computer Science, vol. 3924. Springer-Verlag, 162–179.

Zdancewic, S. and Myers, A. 2002. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation 15,* 2–3 (Sept.), 209–234.

A. $\mathrm{JVM}_{\mathcal{I}}$ UNWINDING LEMMAS

Lemma A.0.1 $\mathrm{JVM}_{\mathcal{I}}$ locally respect. *Let $s_1, s_2 \in \mathrm{State}_{\mathcal{I}}$ be two $\mathrm{JVM}_{\mathcal{I}}$ states at the same program point i and let two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2} s_2$.*

Let $s_1', s_2' \in \mathrm{State}_{\mathcal{I}}$ and $st_1', st_2' \in \mathcal{S}^\star$ such that $s_1 \rightsquigarrow s_1'$, $s_2 \rightsquigarrow s_2'$, $i \vdash st_1 \Rightarrow st_1'$ and $i \vdash st_2 \Rightarrow st_2'$ then $s_1' \sim_{st_1',st_2'} s_2'$.

Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash st_1 \Rightarrow$ and $i \vdash st_2 \Rightarrow$ then $k_r \leq k_{\mathrm{obs}}$ implies $v_1 \sim v_2$.

Lemma A.0.2 $\mathrm{JVM}_{\mathcal{I}}$ step consistent. *Let $\langle i, \rho, os\rangle, s_0 \in \mathrm{State}_{\mathcal{I}}$ two $\mathrm{JVM}_{\mathcal{I}}$ states and two stack types $st, st_0 \in \mathcal{S}^\star$ such that $\langle i, \rho, os\rangle \sim_{st,st_0} s_0$, $se(i) \not\leq k_{\mathrm{obs}}$ and $\mathsf{high}(os, st)$.*

If there exists a state $\langle i', \rho', os'\rangle \in \mathrm{State}_{\mathcal{I}}$ and a stack type $st' \in \mathcal{S}^\star$ such that $\langle i, \rho, os\rangle \rightsquigarrow \langle i, \rho', os'\rangle$ and $i \vdash st \Rightarrow st'$ then $\langle i', \rho', os'\rangle \sim_{st',st_0} s_0$.

If there exists a value $v \in \mathcal{V}$ such that $\langle i, \rho, os\rangle \rightsquigarrow v$ and $i \vdash st \Rightarrow$ then $k_r \not\leq k_{\mathrm{obs}}$.

Lemma A.0.3 $\mathrm{JVM}_{\mathcal{I}}$ high branching. *Let $s_1, s_2 \in \mathrm{State}_{\mathcal{I}}$ be two $\mathrm{JVM}_{\mathcal{I}}$ states at the same program point i and let $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2} s_1'$.*

If two states $\langle i_1, \rho'_1, os'_1\rangle, \langle i_2, \rho'_2, os'_2\rangle \in \text{State}_{\mathcal{I}}$ *and two stack types* $st'_1, st'_2 \in \mathcal{S}^\star$ *such that* $i_1 \neq i_2$, $s_1 \rightsquigarrow \langle i_1, \rho'_1, os'_1\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho'_2, os'_2\rangle$, $i \vdash st_1 \Rightarrow st'_1$, $i \vdash st_2 \Rightarrow st'_2$ *then* $\mathsf{high}(os'_1, st'_1)$, $\mathsf{high}(os'_2, st'_2)$ *and for all* $j \in \mathsf{region}(i)$, $se(j) \not\leq k_{\text{obs}}$.

LEMMA A.0.4 $\text{JVM}_{\mathcal{I}}$ HIGH STEP. *Let* $\langle i, \rho, os\rangle, \langle i', \rho', os'\rangle \in \text{State}_{\mathcal{I}}$ *be two* $\text{JVM}_{\mathcal{I}}$ *states and let* $st, st' \in \mathcal{S}^\star$ *be two stack types such that* $\langle i, \rho, os\rangle \rightsquigarrow \langle i, \rho', os'\rangle$, $i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$ *then* $\mathsf{high}(os', st')$.

B. $\text{JVM}_{\mathcal{O}}$ UNWINDING LEMMAS

LEMMA B.0.5 $\text{JVM}_{\mathcal{O}}$ LOCALLY RESPECT. *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_{\mathcal{O}}$ *two* $\text{JVM}_{\mathcal{O}}$ *states at the same program point* i *and two stack types* $st_1, st_2 \in \mathcal{S}^\star$ *such that* $s_1 \sim_{st_1, st_2, \beta} s_2$.

If $s'_1, s'_2 \in \text{State}_{\mathcal{O}}$ *and two stack types* $st'_1, st'_2 \in \mathcal{S}^\star$ *such that* $s_1 \rightsquigarrow s'_1$, $s_2 \rightsquigarrow s'_2$, $i \vdash st_1 \Rightarrow st'_1$ *and* $i \vdash st_2 \Rightarrow st'_2$ *then there exists* $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that* $s'_1 \sim_{st'_1, st'_2} s'_2$ *and* $\beta \subseteq \beta'$.

If $v_1, v_2 \in \mathcal{V}$ *and* $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash st_1 \Rightarrow$ *and* $i \vdash st_2 \Rightarrow$ *then* $k_r \leq k_{\text{obs}}$ *implies* $v_1 \sim_\beta v_2$.

LEMMA B.0.6 $\text{JVM}_{\mathcal{O}}$ STEP CONSISTENT. *Let* β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, *let* $\langle i, \rho, os, h\rangle, \langle i_0, \rho_0, os_0, h_0\rangle \in \text{State}_{\mathcal{O}}$ *two* $\text{JVM}_{\mathcal{O}}$ *states and two stack types* $st, st_0 \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \sim_{st, st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$.

If a state $\langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{O}}$ *and a stack type* $st' \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \rightsquigarrow \langle i, \rho', os', h'\rangle$ *and* $i \vdash st \Rightarrow st'$ *then* $\langle i', \rho', os', h'\rangle \sim_{st', st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$.

If there exists a value $v \in \mathcal{V}$ *such that* $\langle i, \rho, os, h\rangle \rightsquigarrow v$ *and* $i \vdash st \Rightarrow$ *then* $h' \sim_\beta h_0$ *and* $k_r \not\leq k_{\text{obs}}$.

LEMMA B.0.7 $\text{JVM}_{\mathcal{O}}$ HIGH BRANCHING. *Let* β *be a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_{\mathcal{O}}$ *two* $\text{JVM}_{\mathcal{O}}$ *states at the same program point* i *and two stack types* $st_1, st_2 \in \mathcal{S}^\star$ *such that* $s_1 \sim_{st_1, st_2, \beta} s'_1$.

Let $\langle i_1, \rho'_1, os'_1, h'_1\rangle, \langle i_2, \rho'_2, os'_2, h'_2\rangle \in \text{State}_{\mathcal{O}}$ *be two states and let* $st'_1, st'_2 \in \mathcal{S}^\star$ *be two stack types such that* $i_1 \neq i_2$, $s_1 \rightsquigarrow \langle i_1, \rho'_1, os'_1, h'_1\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho'_2, os'_2, h'_2\rangle$. *If* $i \vdash st_1 \Rightarrow st'_1$, $i \vdash st_2 \Rightarrow st'_2$ *then* $\mathsf{high}(os'_1, st'_1)$, $\mathsf{high}(os'_2, st'_2)$ *and for all* $j \in \mathsf{region}(i)$, $se(j) \not\leq k_{\text{obs}}$.

LEMMA B.0.8 $\text{JVM}_{\mathcal{O}}$ HIGH STEP. *Let* $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{O}}$ *two* $\text{JVM}_{\mathcal{O}}$ *states and two stack types* $st, st' \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \rightsquigarrow \langle i, \rho', os', h'\rangle$, $i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$ *then* $\mathsf{high}(os', st')$.

C. $\text{JVM}_{\mathcal{C}}$ TYPE SYSTEM SOUNDNESS

We assume now P is a program with an associated signature table Γ.

Side-effect safety. The first part of the soundness proof consist in proving that all methods of a typable program are side-effect-safe.

In this paragraph m is suppose to be a method of P, region a cdr function for m, se a security environment, and $\vec{k_a} \xrightarrow{k_h} k_r$ a security signature.

We show that all instruction step transforms a heap h into a heap h' such that $h \preceq_{k_h} h'$. In this first lemma neither virtual call or return instructions are considered.

LEMMA C.0.9. *Let* $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ *be two states such that*

$$\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ *such that*

$$\mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

The next lemma treats the case of the return instruction.

LEMMA C.0.10. *Let a method m and $\langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{C}}$ a state, $h' \in \mathsf{Heap}$ a heap and $v \in \mathcal{V}$ a value such that*

$$\langle i, \rho, os, h\rangle \overset{(0)}{\rightsquigarrow}_m v, h'$$

Let a stack type $st \in \mathcal{S}^\star$ such that

$$\mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow$$

then $h \preceq_{k_h} h'$.

For the special case of virtual call we need an inductive hypothesis about the side-effect safety of all methods in P. We hence introduce the notion of *Side-effect-safe at order n*.

Definition C.0.1 *Side-effect-safe at order n.* A method m is *side-effect-safe at order n* with respect to a security level k_h if for all state $\langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{C}}$, all heap $h' \in \mathsf{Heap}$ and value $v \in \mathcal{V}$, $\langle i, \rho, os, h\rangle \Downarrow_m^{(n)} v, h'$ implies $h \preceq_{k_h} h'$.

LEMMA C.0.11. *Let n an integer and suppose all method m' in P are side-effect-safe at order n with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$.
Let $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ two states such that

$$\langle i, \rho, os, h\rangle \overset{(n+1)}{\rightsquigarrow}_m \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ such that

$$\mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

We then can conclude about the side-effect safety of all methods in P, using an induction on the number of virtual call on semantics derivation and an induction on the derivation length.

LEMMA C.0.12. *For all method m in P, let $(\mathsf{region}_m, \mathsf{jun}_m)$ be a safe cdr for m. Suppose all methods m in P are typable with respect to region_m and to all signatures in* $\text{Policies}_\Gamma(\text{m})$. *Then all method m is side-effect-safe with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m})$.

Non-interference. The soundness proof for non-interference reuses all the basic lemmas proved for $\text{JVM}_{\mathcal{O}}$ which are still valid for the instruction of the $\text{JVM}_{\mathcal{O}}$ (except invokevirtual). The virtual call requires specific lemmas given below.

Definition C.0.2 *non-interference at order n.* A method m is *non-interferent at order n* with respect to a security signature $\vec{k_v} \longrightarrow k_r$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every states $\langle 1, \rho_1, \epsilon_1, h_1\rangle, \langle 1, \rho_2, \epsilon, h_2\rangle \in \text{State}_{\mathcal{C}}$, every heaps $h_1', h_2' \in \mathsf{Heap}$, every values $v_1, v_2 \in \mathcal{V}$ and every integer $n_1, n_2 \in \mathbb{N}$ such that $\langle i, \rho_1, os_1, h_1\rangle \Downarrow_m^{(n_1)} v_1, h_1'$, $\langle i, \rho_2, os_2, h_2\rangle \Downarrow_m^{(n_2)} v_2, h_2'$, $n_1 \leq n$, $n_2 \leq n$ and $\langle 1, \rho_1, \epsilon, h_1\rangle \sim_{\beta,\epsilon,\epsilon} \langle 1, \rho_2, \epsilon, h_2\rangle$ there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1' \sim_{\beta'} h_2'$ and $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_{\beta'} v_2$.

LEMMA C.0.13 $\text{JVM}_{\mathcal{C}}$ LOCALLY RESPECT FOR VIRTUAL CALLS. *Let n an integer, let P a program and a table of signature Γ such that all its method m' are non-interferent at order n, with respect to all the policies in* $\text{Policies}_\Gamma(\text{m}')$ *and side-effect-safe with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$.
Let m be a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \text{State}_{\mathcal{C}}$ two $\text{JVM}_{\mathcal{C}}$ *states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2,\beta} s_2$.*

If there exist two states $s'_1, s'_2 \in \text{State}_{\mathcal{C}}$ *and two stack types* $st'_1, st'_2 \in \mathcal{S}^\star$ *such that* $s_1 \leadsto_m^{(n_1+1)} s'_1$ *with* $n_1 \leq n$, $s_2 \leadsto_m^{(n_2+1)} s'_2$ *with* $n_2 \leq n$, $\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st_1 \Rightarrow st'_1$ *and* $\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st_2 \Rightarrow st'_2$ *then there exists* $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that* $s'_1 \sim_{st'_1, st'_2, \beta'} s'_2$ *and* $\beta \subseteq \beta'$.

LEMMA C.0.14 $\text{JVM}_{\mathcal{C}}$ STEP CONSISTENT FOR VIRTUAL CALLS. *Let* P *be a program and a table of signature* Γ *such that all its method* m' *are side-effect-safe with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$. *Let* m *a method in* P, β *a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle, s_0 \in \text{State}_{\mathcal{C}}$ *two* $\text{JVM}_{\mathcal{C}}$ *states, and two stack types* $st, st_0 \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \sim_{st, st_0, \beta} s_0$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$.

If there exists a state $\langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ *and a stack type* $st' \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \leadsto_m^{(n+1)} \langle i, \rho', os', h'\rangle$ *and* $\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$ *then* $\langle i', \rho', os', h'\rangle \sim_{st', st_0, \beta} s_0$.

Virtual call is not a branching source in $\text{JVM}_{\mathcal{C}}$ so no high branching lemma is required for this instruction.

LEMMA C.0.15 $\text{JVM}_{\mathcal{C}}$ HIGH STEP FOR VIRTUAL CALLS. *Let* m *a method, let two* $\text{JVM}_{\mathcal{C}}$ *states* $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{C}}$ *and two stack types* $st, st' \in \mathcal{S}^\star$ *such that* $\langle i, \rho, os, h\rangle \leadsto_m^{(n+1)} \langle i, \rho', os', h'\rangle$, $\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash st \Rightarrow st'$, $se(i) \not\leq k_{\text{obs}}$ *and* $\mathsf{high}(os, st)$ *then* $\mathsf{high}(os', st')$.

D. $\text{JVM}_{\mathcal{G}}$ TYPE SYSTEM SOUNDNESS

Side-effect safety. The first part of the soundness proof consist in proving that all methods of a typable program are side-effect-safe.

In this paragraph m is supposed to be a method of P, region a cdr function for m, se a security environment and $\vec{k_v} \xrightarrow{k_h} \vec{k_r}$ a security signature.

We show that all instruction step transform a heap h into a heap h' such that $h \preceq_{k_h} h'$. For the special case of virtual call we need an inductive hypothesis about the side-effect safety of all methods in P. We hence introduce the notion of *Side-effect-safe at order n*.

Definition D.0.3 *Side-effect-safe at order n.* A method m is *side-effect-safe at order n* with respect to a security level k_h if for all state $\langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{G}}$, all heap $h' \in \mathsf{Heap}$ and result $t \in \mathcal{V} + \mathcal{L}$, $\langle i, \rho, os, h\rangle \Downarrow_m^{(n)} r, h'$ implies $h \preceq_{k_h} h'$.

LEMMA D.0.16. *Let* n *an integer and suppose all method* m' *in* P *are side-effect-safe at all order* p, $p < n$ *with respect to the heap effect level of all the policies in* $\text{Policies}_\Gamma(\text{m}')$.

Let $\langle i, \rho, os, h\rangle, \langle i, \rho', os', h'\rangle \in \text{State}_{\mathcal{G}}$ *two states and* $p \in \mathbb{N}$, $p \leq n$ *such that*

$$\langle i, \rho, os, h\rangle \overset{(p)}{\leadsto}_{m,\tau} \langle i', \rho', os', h'\rangle$$

Let two stack types $st, st' \in \mathcal{S}^\star$ *such that*

$$i \vdash^\tau st \Rightarrow st'$$

then $h \preceq_{k_h} h'$.

PROOF. By a case analysis on the instruction $P_m[i]$. According to the form $\langle i, \rho, os, h\rangle \leadsto_m \langle i', \rho', os', h'\rangle$ of the semantic step, only four cases appear:

Case 1:. $P_m[i]$ does not modify the heap. We can simply conclude by reflexivity of $\preceq_{k_h}$.

Case 2:. $P_m[i] = \mathsf{new}\ C$ or an instruction throwing a null pointer exception. h' is of the form $h \oplus \{\text{fresh}(h) \mapsto \mathsf{default}(C)\}$ and we can conclude with Lemma E.2.1.

Case 3:. $P_m[i] = \mathsf{putfield}\ f$. h and h' only differ in field f and the corresponding typing rule implies $k_h \leq \mathsf{ft}(f)$. Hence h and h' correspond for all field f' such that $k_h \not\leq \mathsf{ft}(f')$: $h \preceq_{k_h} h'$ holds.

Case 4:. $P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}}$ and the called method has terminated normally or with an exception which is caught in m. p is necessarily of the form $p'+1$ and we have an hypothesis like

$$\langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(p')} (r', h')$$

But $p' < n$ so m' is side-effect safe at order p' and we can hence conclude that $h \preceq_{k_h} h'$.

□

The next lemma treats the case of return instruction.

LEMMA D.0.17. *Let n be an integer and suppose all method m' in P is side-effect-safe at all order p, $p < n$ with respect to the heap effect level of all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m}')$.

Let a method m and $\langle i, \rho, os, h\rangle \in \mathrm{State}_{\mathcal{G}}$ a state, $h' \in \mathsf{Heap}$ a heap, $r \in \mathcal{V} + \mathcal{L}$ a result and $p \in \mathbb{N}$, $p \leq n$ such that

$$\langle i, \rho, os, h\rangle \overset{(p)}{\rightsquigarrow}_{m,\tau} r, h'$$

Let a stack type $st \in \mathcal{S}^\star$ such that

$$i \vdash^\tau st \Rightarrow$$

then $h \preceq_{k_h} h'$.

PROOF. By a case analysis on the instruction in $P_m[i]$. The form of the semantic step constrains $P_m[i]$ to be an instruction ending the execution of m. We hence are in one of the following cases:

Case 1. $P_m[i] = \mathsf{return}$. h is hence equal to h' and we conclude by reflexivity of $\preceq_{k_h}$.

Case 2. if $P_m[i]$ is an instruction throwing a null pointer exception, h' is of the form $h \oplus \{\mathrm{fresh}(h) \mapsto \mathsf{default}(C)\}$ and we can conclude with Lemma E.2.1.

Case 3:. $P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}}$ and the called method has terminated abnormally with an exception uncaught in m. p is necessarily of the form $p'+1$ and we have an hypothesis like

$$\langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \Downarrow_{m'}^{(p')} \langle l'\rangle, h'$$

But $p' < n$ so m' is side-effect safe at order p' and we can hence conclude that $h \preceq_{k_h} h'$.

□

We then can conclude about the side-effect safety of all methods in P, using an induction on the number of virtual calls on semantics derivation and an induction on the derivation length.

PROPOSITION D.0.1 SIDE-EFFECT SAFETY. *Let for all method m in P,* $(\mathsf{region}_m, \mathsf{jun}_m)$ *a safe cdr for m. Suppose all methods m in P are typable with respect to* region_m *and to all signatures in* $\mathrm{Policies}_\Gamma(\mathrm{m})$. *Then all method m are side-effect-safe with respect to the heap effect level of all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m})$.

PROOF. We show for all $n \in \mathbb{N}$ that all method m in P is side-effect-safe at order n with respect to the heap effect level of all the policies in $\mathrm{Policies}_\Gamma(\mathrm{m})$. We use a strong

induction on n. So we suppose all method m in P is side-effect-safe at order k, if $k < n$. We take a method m and prove that it is side-effect-safe at order n with respect to any heap effect level k_h of all the policies in $\text{Policies}_\Gamma(\text{m})$.

Given a state $\langle i_0, \rho_0, os_0, h_0\rangle$ and a final state (r, h') such that $\langle i_0, \rho_0, os_0, h_0\rangle \Downarrow^{(n)} (r, h')$ we have to prove that $h_0 \preceq_{k_h} h'$.

There is necessarily a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

with $n_0 + \cdots + n_k \leq n$.

P is typable so there exists $S \in \mathcal{PP} \to \mathcal{S}^\star$ and $st_1, \ldots, st_{k-1} \in \mathcal{S}^\star$ such that

$$i_0 \vdash^{\tau_0} S(i_0) \Rightarrow st_1 \quad i_1 \vdash^{\tau_1} S(i_1) \Rightarrow st_2 \quad \cdots \quad i_k \vdash^{\tau_k} S(i_k) \Rightarrow$$

For all $i \in \{0, \ldots, k\}$, $n_i \leq n$ so by Lemmas D.0.16 and D.0.17 we have

$$h_0 \preceq_{k_h} h_1 \preceq_{k_h} \cdots \preceq_{k_h} h_k \preceq_{k_h} h$$

We conclude by transitivity of $\preceq_{k_h}$. □

Non-interference proof. When a method execution ends in a high context (*i.e.* in a region where the security environment is high), the type system enforces the output level to be high according to the kind of termination of the method (normal or with an uncaught exception). This notion of *high result* is captured by the following formal definition.

Definition D.0.4 *High result.* Given $(r, h) \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and an output level $\vec{k_r}$, the predicate $\mathsf{highResult}_{\vec{k_r}}(r, h)$ is inductively defined by:

$$\frac{\vec{k_r}[n] \not\leq k_{\text{obs}} \qquad v \in \mathcal{V}}{\mathsf{highResult}_{\vec{k_r}}(v, h)} \qquad \frac{\vec{k_r}[\mathsf{class}(h(l))] \not\leq k_{\text{obs}} \qquad l \in \mathsf{dom}(h)}{\mathsf{highResult}_{\vec{k_r}}(\langle l\rangle, h)}$$

We then prove non-interference using the four lemmas sketched in Section 2. From now, we assume P is a program and Γ a table of signatures such that all its method m are side-effect-safe with respect to the heap effect level of all the policies in $\text{Policies}_\Gamma(\text{m})$.

In this paragraph m is suppose to be a method of P, $(\mathsf{region}, \mathsf{jun})$ a cdr function for m, se a security environment, $S \in \mathcal{PP} \to \mathcal{S}^\star$ a stack type annotation and $\vec{k_v} \xrightarrow{k_h} \vec{k_r}$ a security signature.

LEMMA D.0.18 $\text{JVM}_\mathcal{G}$ HIGH STEP. *Let* $\langle i, \rho, os, h\rangle, \langle i', \rho', os', h'\rangle \in \text{State}_\mathcal{G}$ *be two* $\text{JVM}_\mathcal{G}$ *states and two stack types* $st, st' \in \mathcal{S}^\star$ *such that*

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i, \rho', os', h'\rangle \qquad i \vdash st \Rightarrow^\tau st'$$
$$se(i) \not\leq k_{\text{obs}} \qquad \mathsf{high}(os, st)$$

then

$$\mathsf{high}(os', st')$$

PROOF. By case analysis on $P_m[i]$. Only non final steps are considered here. For each case we first resume the hypotheses and state the result to prove in table of Figure 27.

Each case is easily proved using the fact that $se(i) \not\leq k_{\text{obs}}$ and the following generic property:

$$\forall k \in \mathcal{L}, os \in \mathcal{V}^\star, st \in \mathcal{S}^\star,\ \mathsf{high}(os, st) \text{ implies } \mathsf{high}(os, \text{lift}_k(st))$$

□

$P_m[i]$	Hypotheses	Goal
push n	$\mathsf{high}(os, st)$	$\mathsf{high}(n :: os, se(i) :: st)$
binop op	$\mathsf{high}(n_1 :: n_2 :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(n_1 \ \underline{op} \ n_2 :: os, \overline{se(i)} :: (k_1 \sqcup k_2 \sqcup se(i)) :: st)$
pop	$\mathsf{high}(v :: os, k :: st)$	$\mathsf{high}(os, st)$
store x	$\mathsf{high}(v :: os, k :: st)$	$\mathsf{high}(os, st)$
load x	$\mathsf{high}(os, st)$	$\mathsf{high}(\rho(x) :: os, \left(\vec{k_v}(x) \sqcup se(i)\right) :: st)$
goto j	$\mathsf{high}(os, st)$	$\mathsf{high}(os, st)$
ifeq j	$\mathsf{high}(n :: os, k :: st)$	$\mathsf{high}(os, \mathrm{lift}_k(st))$
new C	$\mathsf{high}(os, st)$	$\mathsf{high}(l :: os, se(i) :: st)$
getfield f	$\mathsf{high}(l :: os, k :: st)$	$\mathsf{high}(h(l).f :: os, \mathrm{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st))$
	$\mathsf{high}(null :: os, k :: st)$	$\mathsf{high}(l' :: \epsilon, k \sqcup se(i) :: \epsilon)$
putfield f	$\mathsf{high}(v :: l :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(os, \mathsf{lift}_{k_2} st)$
	$\mathsf{high}(v :: null :: os, k_1 :: k_2 :: st)$	$\mathsf{high}(l' :: \epsilon, k_2 \sqcup se(i) :: \epsilon)$
invokevirtual m_{ID}	$\mathsf{high}(os_1 :: l :: os_2, st_1 :: k :: st_2)$	$\mathsf{high}(v :: os_2, \mathrm{lift}_{k \sqcup k_e} \left((\vec{k_r}[n] \sqcup se(i)) :: st_2\right))$
	$\mathsf{high}(os_1 :: null :: os_2, st_1 :: k :: st_2)$	$\mathsf{high}(l' :: \epsilon, k \sqcup \vec{k_r}[e] :: \epsilon)$
throw	$\mathsf{high}(l :: os, k :: st)$	$\mathsf{high}(l :: \epsilon, k \sqcup se(i) :: \epsilon)$
	$\mathsf{high}(null :: os, k :: st)$	$\mathsf{high}(l' :: \epsilon, k \sqcup se(i) :: \epsilon)$

Fig. 27. Case analysis for Lemma D.0.18

LEMMA D.0.19 $\mathrm{JVM}_{\mathcal{G}}$ STEP CONSISTENT. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, let $\langle i, \rho, os, h\rangle, \langle i_0, \rho_0, os_0, h_0\rangle \in \mathrm{State}_{\mathcal{G}}$ two $\mathrm{JVM}_{\mathcal{G}}$ states and two stack types $st, st_0 \in \mathcal{S}^\star$ such that*

$$\langle i, \rho, os, h\rangle \sim_{st, st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle \qquad se(i) \not\leq k_{\mathrm{obs}} \qquad \mathsf{high}(os, st)$$

(1) If there exists a state $\langle i', \rho', os', h'\rangle \in \mathrm{State}_{\mathcal{G}}$, a tag $\tau \in \{\emptyset\} + \mathcal{C}$ and a stack type $st' \in \mathcal{S}^\star$ such that

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle \qquad i \vdash^\tau st \Rightarrow st'$$

then

$$\langle i', \rho', os', h'\rangle \sim_{st', st_0, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$$

(2) If there exists a result $r \in \mathcal{V} + \mathcal{L}$, a heap $h' \in \mathsf{Heap}$ and a tag $\tau \in \{\emptyset\} + \mathcal{C}$ such that

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} r, h' \qquad i \vdash^\tau st \Rightarrow$$

then

$$\mathsf{highResult}_{\vec{k_r}}(r, h') \quad \text{and} \quad h' \sim_\beta h_0$$

PROOF.

(1) Non-terminal step of the form $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$. Proofs can be reduce to examining local variables and heaps because, from $\mathsf{high}(os, st)$ and $os \sim_{st, st_0, \beta} os_0$ we first remark that $\mathsf{high}(os_0, st_0)$. Using the previous lemma ($\mathrm{JVM}_{\mathcal{G}}$ high step), we know that $\mathsf{high}(os', st')$. As a consequence, we already know that $os' \sim_{st', st_0, \beta} os_0$ for any kind of instruction. We now prove the remaining $\rho' \sim_{\vec{k_v}, \beta} \rho_0$ and $h' \sim_\beta h_0$ properties.
Local variables: Only the instruction store x modifies the local variable. We hence have to prove

$$\rho \oplus \{x \mapsto v\} \sim_{\vec{k_v}, \beta} \rho_0$$

We remark that $\vec{k_v}(x) \not\leq k_{\mathrm{obs}}$ thanks to the typing rule which impose $k \leq \vec{k_v}(x)$ and the hypothesis $\mathsf{high}(v :: os, k :: st)$ which gives $k \not\leq k_{\mathrm{obs}}$. Since local

variables indistinguishability only depends on low variable and the modified variable x is high, we are done.

Heaps: Instruction which modify heaps are $\mathsf{new}\ C$, normal execution of $\mathsf{putfield}\ f$, $\mathsf{invokevirtual}\ m_{\mathrm{ID}}$ and all instructions which throw a null pointer exception. We now examine this different cases and conclude using an appropriate technical lemma put in appendix :

— $\mathsf{new}\ C$. We conclude by lemma E.2.2.
— $\mathsf{putfield}\ f$. We conclude by lemma E.2.3.
— $\mathsf{invokevirtual}\ m_{\mathrm{ID}}$. We conclude by lemma E.2.4.
— Null pointer throwing. We conclude by lemma E.2.2.

(2) We make a case analysis on $P_m[i]$ for step of the form $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} r, h'$.

— return. The corresponding typing rule enforce $k \leq \vec{k_r}[n]$ with $\mathsf{high}(v :: os, k :: st)$ so $\vec{k_r}[n] \not\leq k_{\mathrm{obs}}$ holds. The heap is not modified here so we are done.
— $\mathsf{getfield}\ f$ (uncaught null pointer exception). By the typing rule, we have $k \leq \vec{k_r}[\mathbf{np}]$ and $\mathsf{high}(null :: os, k :: st)$ so $\vec{k_r}[\mathbf{np}] \not\leq k_{\mathrm{obs}}$. The proof concerning heap is similar to the case where a null pointer exception is caught (see previous item).
— $\mathsf{putfield}\ f$ (uncaught null pointer exception). By the typing rule, we have $k_2 \leq \vec{k_r}[\mathbf{np}]$ and $\mathsf{high}(v :: null :: os, k_1 :: k_2 :: st)$ so $\vec{k_r}[\mathbf{np}] \not\leq k_{\mathrm{obs}}$. The proof concerning heap is similar to the case where a null pointer exception is caught (see previous item).
— $\mathsf{invokevirtual}\ m_{\mathrm{ID}}$ (uncaught exception e). By the typing rule, we have $k \leq \vec{k_r}[e]$ and $\mathsf{high}(os_1 :: v :: os_2, st_1 :: k :: st_2)$ so $\vec{k_r}[e] \not\leq k_{\mathrm{obs}}$. The proof concerning heap is similar to the case where the exception is caught (see previous item).
— throw. $k \leq \vec{k_r}[e]$ and $\mathsf{high}(l :: os, k :: st)$ (or $\mathsf{high}(null :: os, k :: st)$) lead to the expected condition $\vec{k_r}[e] \not\leq k_{\mathrm{obs}}$. If $e \neq \mathbf{np}$, the heap is not modified. If $e = \mathbf{np}$ we conclude about the heap condition as when a null pointer exception is caught (see previous item).

□

These two lemmas about high steps are then used (with SOAP properties) to characterize execution of high fragment of code. The proof is more complex than the one sketched in Section 2 because of the tag that now decorate regions.

We now give the definition of *typable execution* that will be used in the next proofs.

Definition D.0.5 *typable execution.*

— An execution step $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if there exists st' such that, $i \vdash^\tau S_i \Rightarrow st'$ and $st' \sqsubseteq S_{i'}$.

— An execution step $\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} (r, h')$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if $i \vdash^\tau S_i \Rightarrow$.

— An execution sequence $s_0 \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_0} \ldots s_k \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$ is typable with respect to $S \in \mathcal{PP} \to \mathcal{S}^\star$ if

— for all i, $0 \leq i < k$, $s_i \overset{(n_i)}{\rightsquigarrow}_{m,\tau_i} s_{i+1}$ is typable with respect to S;
— $s_n \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$ is typable with respect to S.

The first lemma prove that high executions end in a junction point or in a return point.

LEMMA D.0.20 ITERATED STEP CONSISTENT. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i_0, \rho_0, os_0, h_0\rangle, \langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ states and a stack type $st \in \mathcal{S}^\star$ such that*

$$\langle i_0, \rho_0, os_0, h_0\rangle \sim_{S_{i_0}, st, \beta} \langle i, \rho, os, h\rangle \qquad \mathsf{high}(os_0, S_{i_0})$$

Let $i \in \mathcal{PP}$ and $\tau \in \{\emptyset\} + \mathcal{C}$ such that

$$i_0 \in \mathsf{region}(i, \tau) \qquad \textit{se is high in region } \mathsf{region}(i, \tau)$$

Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h')$$

and suppose this derivation is typable with respect to S. Then one of the following cases holds:

(1) $\mathsf{jun}(i, \tau)$ is defined and there exists j with $0 < j \leq k$ such that

$$i_j = \mathsf{jun}(i, \tau), \quad \langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, st, \beta} \langle i, \rho, os, h\rangle \quad \textit{and} \quad \mathsf{high}(os_j, S_{i_j})$$

(2) $\mathsf{jun}(i, \tau)$ is undefined, $k \in \mathsf{region}(i, \tau)$, $\mathsf{highResult}_{\vec{k_r}}(r, h')$ and $h' \sim_\beta h$.

PROOF. By induction on k.

— $k = 0$ we can directly apply case 2 of Lemma D.0.19 and SOAP3 to conclude that $\mathsf{jun}(i, \tau)$ is undefined. Hence we are in case 2.

— we suppose the statement is true for a given k and we prove it now for $k + 1$. First note that $se(i_0) \not\leq k_{\text{obs}}$, $\mathsf{high}(os_0, S_{i_0})$ and $i_0 \vdash S_{i_0} \Rightarrow st_1'$ hold for some st_1' such that $st_1' \sqsubseteq S_{i_1}$, so we have

$$\langle i_1, \rho_1, os_1, h_1\rangle \sim_{st_1', st, \beta} s \quad \text{and} \quad \mathsf{high}(os_1, st_1')$$

by case 1 of Lemma D.0.19 and by Lemma D.0.18.
By subtyping lemmas E.4.3 and E.4.1 (stated in page 73) we have:

$$\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1}, st, \beta} s \quad \text{and} \quad \mathsf{high}(os_1, S_{i_1})$$

Then, using SOAP2, we have $i_1 \in \mathsf{region}(i, \tau)$ or $i_1 = \mathsf{jun}(i, \tau)$.
In the first case, it is sufficient to invoke induction hypothesis on derivation

$$\langle i_1, \rho_1, os_1, h_1\rangle \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \cdots \langle i_{k+1}, \rho_{k+1}, os_{k+1}, h_{k+1}\rangle \overset{(n_{k+1})}{\rightsquigarrow}_{m,\tau_{k+1}} (r, h)$$

to conclude. We can apply inductive hypothesis because:
— $i_1 \in \mathsf{region}(i, \tau)$;
— the derivation is on length k and is typable;
— $\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1}, st, \beta} s$ and $\mathsf{high}(os_1, S_{i_1})$ hold
In the second case we can conclude we are in case 1 by taking $j = 1$ and because we know $\langle i_1, \rho_1, os_1, h_1\rangle \sim_{S_{i_1}, st, \beta} s$ and $\mathsf{high}(os_1, S_{i_1})$ hold.

□

In order to describe high executions starting from high branching we first prove a technical lemma where one execution starts in a junction point. Such a lemma would not have been necessary if we had not distinguished regions according to tags.

LEMMA D.0.21 ITERATED STEP CONSISTENT ON JUNCTION POINT. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, os_0, h_0\rangle, \langle i_0', \rho_0', os_0', h_0'\rangle \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ states such that*

$$\langle i_0, \rho_0, os_0, h_0\rangle \sim_{S_{i_0}, S_{i_0'}, \beta} \langle i_0', \rho_0', os_0', h_0'\rangle, \qquad \mathsf{high}(os_0, S_{i_0}), \quad \textit{and} \quad \mathsf{high}(os_0', S_{i_0'})$$

Let $i \in \mathcal{PP}$ *and* $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ *such that*

$$\begin{array}{cc} i_0 \in \mathsf{region}(i,\tau) & \textit{se is high in region } \mathsf{region}(i,\tau) \\ i'_0 = \mathsf{jun}(i,\tau') & \textit{se is high in region } \mathsf{region}(i,\tau') \end{array}$$

Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r,h)$$

and suppose this derivation is typable with respect to S*. Suppose we have a derivation*

$$\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots \langle i'_k, \rho'_k, os'_k, h'_k\rangle \overset{(n'_k)}{\rightsquigarrow}_{m,\tau'_k} (r',h')$$

and suppose this derivation is typable with respect to S*.*

Then one of the following case holds:

(1) there exists j, j' *with* $0 \le j \le k$ *and* $0 \le j' \le k'$ *such that*

$$i_j = i'_{j'} \quad \textit{and} \quad \langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_{j'}}, \beta} \langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle$$

(2) $(r,h) \sim_{\vec{k_r},\beta} (r',h')$

Proof. We first invoke Lemma D.0.20 on the derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots (r,h)$$

and the region $\mathsf{region}(i,\tau)$. We have then two cases:

(1) There exists j, with $0 < j \le k$ such that $i_j = \mathsf{jun}(i,\tau)$ and $\langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0\rangle$.
If $\mathsf{jun}(i,\tau) = \mathsf{jun}(i,\tau')$ where are in case 1 with $j' = 0$.
If not, we can invoke SOAP4: $\mathsf{jun}(i,\tau) \in \mathsf{region}(i,\tau')$ or $\mathsf{jun}(i,\tau') \in \mathsf{region}(i,\tau)$.
— If $i_j = \mathsf{jun}(i,\tau) \in \mathsf{region}(i,\tau')$, we invoke Lemma D.0.20 on the derivation $\langle i_j, \rho_j, os_j, h_j\rangle \overset{(n_j)}{\rightsquigarrow}_{m,\tau_j} \cdots (r,h)$ and the region $\mathsf{region}(i,\tau')$. Either there exists q, with $j < q \le k$ such that $i_q = \mathsf{jun}(i,\tau') = i'_0$ and $\langle i_q, \rho_q, os_q, h_q\rangle \sim_{S_{i_q}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0\rangle$ and we can conclude we are in case 1 with $j = q$ and $j' = 0$. Or $k \in \mathsf{region}(i,\tau')$ and we obtain a contradiction (thanks to SOAP3) because $\mathsf{jun}(i,\tau')$ is defined and k is a return point.
— If $i'_0 = \mathsf{jun}(i,\tau') \in \mathsf{region}(i,\tau)$, we invoke Lemma D.0.20 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r',h')$ and the region $\mathsf{region}(i,\tau)$. Either there exists j', with $0 < j' \le k'$ such that $i'_{j'} = \mathsf{jun}(i,\tau) = i_j$ and $\langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle \sim_{S_{i'_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$ and we can conclude we are in case 1. Or $k' \in \mathsf{region}(i,\tau)$ and we obtain a contradiction (tanks to SOAP3) because $\mathsf{jun}(i,\tau) = i_j$ is defined and k' is a return point.

(2) $\mathsf{jun}(i,\tau)$ is undefined, $k \in \mathsf{region}(i,\tau)$, $\mathsf{highResult}_{\vec{k_r}}(r,h)$ and $h \sim_\beta h'_0$. k is a return point in region $\mathsf{region}(i,\tau)$ so, thanks to SOAP5, we know that $i'_0 = \mathsf{jun}(i,\tau') \in \mathsf{region}(i,\tau)$. We can hence invoke Lemma D.0.20 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r',h')$ and the region $\mathsf{region}(i,\tau)$. Either there exists j', with $0 < j' \le k'$ such that $i'_{j'} = \mathsf{jun}(i,\tau) = i_j$ and $\langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle \sim_{S_{i'_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$ and we can conclude we are in case 1. Or $k' \in \mathsf{region}(i,\tau)$, $\mathsf{highResult}_{\vec{k_r}}(r',h')$, $h' \sim_\beta h_0$ and we can conclude we are in case 2.

□

Thanks to the previous lemma, we establish now that after a high branching, both executions stay high until reaching a common point or a return point.

LEMMA D.0.22 ITERATED STEP CONSISTENT AFTER BRANCHING. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, os_0, h_0\rangle, \langle i'_0, \rho'_0, os'_0, h'_0\rangle \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ states such that*

$$\langle i_0, \rho_0, os_0, h_0\rangle \sim_{S_{i_0}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0\rangle \qquad \mathsf{high}(os_0, st_0) \qquad \mathsf{high}(os'_0, st'_0)$$

Let $i \in \mathcal{PP}$ and $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ such that

$$i \mapsto^{\tau} i_0 \quad \textit{and} \quad i \mapsto^{\tau'} i'_0$$

Suppose that se is high in region $\mathsf{region}_m(i, \tau)$ and also in region $\mathsf{region}_m(i, \tau')$. Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

and suppose this derivation is typable with respect to S. Suppose we have a derivation

$$\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots \langle i'_k, \rho'_k, os'_k, h'_k\rangle \overset{(n'_k)}{\rightsquigarrow}_{m,\tau'_k} (r', h')$$

and suppose this derivation is typable with respect to S. Then one of the following case holds:

(1) there exists j, j' with $0 \leq j \leq k$ and $0 \leq j' \leq k'$ such that $i_j = i'_{j'}$ and $\langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_{j'}}, \beta} \langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle$;

(2) $(r, h) \sim_{\vec{k_r}, \beta} (r', h')$

PROOF. If $i_0 = i'_0$ then case 1 trivially holds.
If $i_0 \neq i'_0$ then, using SOAP1 two times, we distinguish four cases:

(1) $i_0 \in \mathsf{region}(i, \tau)$ and $i'_0 \in \mathsf{region}(i, \tau')$. We first invoke Lemma D.0.20 on the derivation $\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots (r, h)$ and the region $\mathsf{region}(i, \tau)$. We hence obtain two cases. In the first case there exists j, with $0 < j \leq k$ such that $i_j = \mathsf{jun}(i, \tau)$ and $\langle i_j, \rho_j, os_j, h_j\rangle \sim_{S_{i_j}, S_{i'_0}, \beta} \langle i'_0, \rho'_0, os'_0, h'_0\rangle$ and we can conclude 1 thanks to Lemma D.0.21. In the second case $k \in \mathsf{region}(i, \tau)$, $\mathsf{highResult}_{\vec{k_r}}(r, h)$ and $h \sim_\beta h'_0$. In this case, we invoke Lemma D.0.20 on the derivation $\langle i'_0, \rho'_0, os'_0, h'_0\rangle \overset{(n'_0)}{\rightsquigarrow}_{m,\tau'_0} \cdots (r', h')$ and the region $\mathsf{region}(i, \tau')$. We again obtain two cases. If there exists j', with $0 < j' \leq k'$ such that $i_{j'} = \mathsf{jun}(i, \tau')$ and $\langle i_{j'}, \rho_{j'}, os_{j'}, h_{j'}\rangle \sim_{S_{i_{j'}}, S_{i_0}, \beta} \langle i_0, \rho_0, os_0, h_0\rangle$, we can conclude we are in case 1 thanks to Lemma D.0.21. In the second case $k' \in \mathsf{region}(i, \tau')$, $\mathsf{highResult}_{\vec{k_r}}(r', h')$ and $h' \sim_\beta h_0$. We hence have

$$\begin{cases} \mathsf{highResult}_{\vec{k_r}}(r, h) \\ \mathsf{highResult}_{\vec{k_r}}(r', h') \\ h' \sim_\beta h_0 \\ h \sim_\beta h'_0 \\ h_0 \sim_\beta h'_0 \end{cases}$$

which implies $(r, h) \sim_{\vec{k_r}, \beta} (r', h')$.

(2) $i_0 = \mathsf{jun}(i, \tau)$ and $i'_0 \in \mathsf{region}(i, \tau')$. We easily conclude by Lemma D.0.21.

(3) $i_0 \in \mathsf{region}(i, \tau)$ and $i'_0 = \mathsf{jun}(i, \tau')$. We easily conclude by Lemma D.0.21.

(4) $i_0 = \mathsf{jun}(i, \tau)$ and $i'_0 = \mathsf{jun}(i, \tau')$. If $\mathsf{jun}(i, \tau) = \mathsf{jun}(i, \tau')$ we are in case 1. If not, SOAP4 applies : $i_0 = \mathsf{jun}(i, \tau) \in \mathsf{region}(i, \tau')$ or $i'_0 = \mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$. In both cases Lemma D.0.21 allows to conclude.

□

We now must prove similar lemmas about high execution for the special cases where a branching occurs in a return point. This first lemma is a corollary of Lemma D.0.19.

LEMMA D.0.23 $\text{JVM}_{\mathcal{G}}$ FINAL STEP CONSISTENT. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i, \rho, os, h\rangle \in \text{State}_{\mathcal{G}}$ a $\text{JVM}_{\mathcal{G}}$ state, $h_0 \in \mathsf{Heap}$ a heap and a stack type $st \in \mathcal{S}^\star$ such that*

$$h \sim_\beta h_0 \qquad se(i) \not\leq k_{\text{obs}} \qquad \mathsf{high}(os, st)$$

(1) If there exists a state $\langle i', \rho', os', h'\rangle \in \text{State}_{\mathcal{G}}$, a tag $\tau \in \{\emptyset\} + \mathcal{C}$ and a stack type $st' \in \mathcal{S}^\star$ such that

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i, \rho', os', h'\rangle \qquad i \vdash^\tau st \Rightarrow st'$$

then

$$\mathsf{high}(os', st') \quad \text{and} \quad h' \sim_\beta h_0$$

(2) If there exists a result $r \in \mathcal{V} + \mathcal{L}$, a heap $h' \in \mathsf{Heap}$ and a tag $\tau \in \{\emptyset\} + \mathcal{C}$ such that

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} r, h' \qquad i \vdash^\tau st \Rightarrow$$

then

$$\mathsf{highResult}_{\vec{k_r}}(r, h') \quad \text{and} \quad h' \sim_\beta h_0$$

PROOF. Similar to the proof of Lemma D.0.19. □

LEMMA D.0.24 ITERATED FINAL STEP CONSISTENT AFTER BRANCHING. *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $\langle i_0, \rho_0, os_0, h_0\rangle \in \text{State}_{\mathcal{G}}$ a $\text{JVM}_{\mathcal{G}}$ states and $h_0' \in \mathsf{Heap}$ a heap such that*

$$h_0 \sim_\beta h_0' \qquad \mathsf{high}(os_0, S_{i_0})$$

Let $i \in \mathcal{PP}$ and $\tau, \tau' \in \{\emptyset\} + \mathcal{C}$ such that

$$i_0 \in \mathsf{region}(i, \tau) \quad \textit{and} \quad i \mapsto^{\tau'}$$

Suppose that se is high in region $\mathsf{region}_m(i, \tau)$. Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0\rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k\rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

and suppose this derivation is typable with respect to S. Then

$$\mathsf{highResult}_{\vec{k_r}}(r, h) \quad \text{and} \quad h \sim_\beta h_0'$$

PROOF. By induction on k.

— $k = 0$ we can directly apply case 2 of Lemma D.0.23 to conclude.

— we suppose the statement is true for a given k and we prove it now for $k+1$. First note that $se(i_0) \not\leq k_{\text{obs}}$, $\mathsf{high}(os_0, S_{i_0})$ and $i_0 \vdash S_{i_0} \Rightarrow st_1$ hold for some st_1 such that $st_1 \sqsubseteq S_{i_1}$, so we have

$$h_1 \sim_\beta h_0' \quad \text{and} \quad \mathsf{high}(os_1, st_1)$$

by case 1 of lemma D.0.23 and by lemma D.0.18. By subtyping lemma E.4.1 we have also:

$$\mathsf{high}(os_1, S_{i_1})$$

Then, using SOAP2, we have $i_1 \in \mathsf{region}(i, \tau)$ or $i_1 = \mathsf{jun}(i, \tau)$.
In the first case, it is sufficient to invoke induction hypothesis on derivation

$$\langle i_1, \rho_1, os_1, h_1\rangle \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \cdots \langle i_{k+1}, \rho_{k+1}, os_{k+1}, h_{k+1}\rangle \overset{(n_{k+1})}{\rightsquigarrow}_{m,\tau_{k+1}} (r, h)$$

to conclude. Let's justify we can use it:

— $i_1 \in \mathsf{region}(i, \tau)$;
— the derivation is on length k and is typable;
— $h_1 \sim_\beta h'_0$ and $\mathsf{high}(os_1, S_1)$ hold
In the second case we have a contradiction because SOAP3 implies that $\mathsf{jun}(i, \tau)$ is undefined.

□

This lemma ends the proof about high executions. We now examine parallel executions of methods, proving first the two remaining lemmas sketched in section 2. Method calls requires to use a notion of *non-interference at order n* to state these lemmas.

Definition D.0.6 *non-interference at order n.* A method m is *non-interferent at order n* with respect to a security signature $\vec{k_v} \longrightarrow \vec{k_r}$, if for every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every states $\langle 1, \rho_1, \epsilon, h_1\rangle, \langle 1, \rho_2, \epsilon, h_2\rangle \in \mathrm{State}_\mathcal{G}$, every heaps $h'_1, h'_2 \in \mathsf{Heap}$, every results $r_1, r_2 \in \mathcal{V} + \mathcal{L}$ and every integer $n_1, n_2 \in \mathbb{N}$ such that

$$\langle i, \rho_1, os_1, h_1\rangle \Downarrow_m^{(n_1)} r_1, h'_1, \quad \langle i, \rho_2, os_2, h_2\rangle \Downarrow_m^{(n_2)} r_2, h'_2, \quad n_1 \leq n \quad n_2 \leq n$$

and

$$\langle 1, \rho_1, \epsilon, h_1\rangle \sim_{\beta,\epsilon,\epsilon} \langle 1, \rho_2, \epsilon, h_2\rangle$$

there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$\beta \subseteq \beta' \quad \text{and} \quad (r_1, h'_1) \sim_{\vec{k_r},\beta'} (r_2, h'_2)$$

Lemma D.0.25 $\mathrm{JVM}_\mathcal{G}$ high branching. *Let n an integer such that all method m' in P are non-interferent at all order k, $k < n$, with respect to all the policies in* $\mathrm{Policies}_\Gamma(\mathrm{m}')$.

Let m a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \mathrm{State}_\mathcal{G}$ two $\mathrm{JVM}_\mathcal{G}$ states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that

$$s_1 \sim_{st_1,st_2,\beta} s_2$$

(1) If there exists two states $\langle i_1, \rho'_1, os'_1, h'_1\rangle, \langle i_2, \rho'_2, os'_2, h'_2\rangle \in \mathrm{State}_\mathcal{G}$ and two stack types $st'_1, st'_2 \in \mathcal{S}^\star$ such that

$$\begin{array}{rr} & i_1 \neq i_2 \\ s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i_1, \rho'_1, os'_1, h'_1\rangle & n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} \langle i_2, \rho'_2, os'_2, h'_2\rangle & n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st'_1 \quad i \vdash^{\tau_2} st_2 \Rightarrow st'_2 & \end{array}$$

then

$$\begin{array}{l} \mathsf{high}(os'_1, st'_1) \quad \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_1) \\ \mathsf{high}(os'_2, st'_2) \quad \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_2) \end{array}$$

(2) If there exists a state $\langle i_1, \rho'_1, os'_1, h'_1\rangle \in \mathrm{State}_\mathcal{G}$, a final result $(r_2, h'_2) \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and a stack type $st'_1 \in \mathcal{S}^\star$ such that

$$\begin{array}{rr} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i_1, \rho'_1, os'_1, h'_1\rangle & n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h'_2) & n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st'_1 \quad i \vdash^{\tau_2} st_2 \Rightarrow & \end{array}$$

then

$$\mathsf{high}(os'_1, st'_1) \quad \textit{and} \quad \textit{se is high in } \mathsf{region}(i, \tau_1)$$

PROOF. By case analysis on $P_m[i]$. We only consider branching instructions here.

Case:. $P_m[i] = \mathsf{ifeq}\ j$. By semantics, $\tau_1 = \emptyset$, $\tau_2 = \emptyset$, $s_1 = \langle i, \rho_1, n_1 :: os_1, h_1\rangle$ and $s_2 = \langle i, \rho_2, n_2 :: os_2, h_2\rangle$. Since $i_1 \neq i_2$, we have necessarily $n_1 \neq n_2$. By typability, $st_1 = k_1 :: st_1''$, $st_2 = k_2 :: st_2''$, $st_1' = \mathrm{lift}_{k_1} st_1''$ and $st_2' = \mathrm{lift}_{k_2} st_2''$. Since $n_1 :: os_1 \sim_{k_1::st_1'', k_2::st_2'', \beta} n_2 :: os_2$, and $n_1 \neq n_2$ we deduce that $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$. The first consequence is $\mathsf{high}(os_1, \mathrm{lift}_{k_1} st_1'')$ and $\mathsf{high}(os_2, \mathrm{lift}_{k_2} st_2'')$. By typability, $\forall j' \in \mathsf{region}(i, \emptyset),\ k_1 \leq se(j')$ and hence se is high in region $\mathsf{region}(i, \emptyset)$.

Case:. $P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}}$. There are several kinds of branching for method calls. s_1 is necessarily of the form $\langle i, \rho_1, os_1 :: v_1 :: os_1', h_1\rangle$ and s_2 of the form $\langle i, \rho_2, os_2 :: v_2 :: os_2', h_2\rangle$ with $v_1, v_2 \in \mathcal{L} \cup \{null\}$. Furthermore, $st_1 = st_1' :: k_1 :: st_1''$ and $st_2 = st_2' :: k_2 :: st_2''$. We then make a first distinction according if one of v_1 or v_2 is equal to *null*.

Case 1: v_1 or v_2 is equal to *null*. There is only a branching if the other value is not null. We only make the case $v_1 = null$ and $v_2 = l_2 \in \mathcal{L}$, the other case is done by symmetry.
By hypothesis

$$os_1 :: null :: os_1' \sim_{st_1'::k_1::st_1'', st_2'::k_2::st_2'', \beta} os_2 :: l_2 :: os_2'$$

and $\mathsf{length}(os_1) = \mathsf{length}(st_1) = \mathsf{length}(os_2) = \mathsf{length}(st_2)$, so necessarily $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$. We deduce then $\mathsf{high}(os_1', (k_1 \sqcup \vec{k_r'}[\mathbf{np}]) :: \epsilon)$ (if $\mathsf{Handler}_m(i, \mathbf{np}) = t$) and that se is high in region $\mathsf{region}(i, \emptyset)$ thanks to the typability constraint $\forall j \in \mathsf{region}(i, \emptyset),\ (k_1 \sqcup \vec{k_r'}[\mathbf{np}]) \leq se(j)$ (which occurs in both typing rules, if $\mathbf{np}$ is caught or not).

We then examine the transition $\langle i, \rho_2, os_2 :: l_2 :: os_2', h_2\rangle \overset{(n_2)}{\rightsquigarrow}_m \cdots$. There are three cases: normal termination of the called method, termination by an exception which can be caught or uncaught in m. In all cases the corresponding typing rules enforce that all element of the next stack type (if there is any one) are greater or equal to k_2 and for all point j in $\mathsf{region}(i, \tau_2)$ $k_2 \leq se(i)$. Hence we are done since $k_2 \not\leq k_{\mathrm{obs}}$.

Case 2: both v_1 and v_2 are in $\mathcal{L}$ (we note them l_1 and l_2 from now). By hypothesis

$$os_1 :: l_1 :: os_1' \sim_{st_1'::k_1::st_1'', st_2'::k_2::st_2'', \beta} os_2 :: l_2 :: os_2'$$

Since $\mathsf{length}(os_1) = \mathsf{length}(st_1) = \mathsf{length}(os_2) = \mathsf{length}(st_2)$, we have by case analysis on lemme E.1.4:
— either $k_1 \not\leq k_{\mathrm{obs}}$ and $k_2 \not\leq k_{\mathrm{obs}}$. In this case we make a similar reasoning as before . There are three cases for the first execution and three others for the second but in all cases the corresponding typing rules enforce that all element of the next stack type (if there is any one) are greater or equal to k_1 (or k_2) and for all point j in $\mathsf{region}(i, \tau_1)$ (or $\mathsf{region}(i, \tau_2)$) $k_1 \leq se(i)$ (or $k_2 \leq se(i)$).
— or $k_1 = k_2$ and $l_1 \sim_\beta l_2$. Since $h_1 \sim_\beta h_2$ we deduce $\mathsf{class}(h_1(l_1)) = \mathsf{class}(h_2(l_2))$ and as a consequence the same method m' is called in both execution. $\Gamma_{m_{\mathrm{ID}}}[k_1]$ and $\Gamma_{m_{\mathrm{ID}}}[k_2]$ are then equals to a same signature $\vec{k_v'} \xrightarrow{k_h'} \vec{k_r'}$. Again, there are three cases for each execution but each times we have

$$\{this \mapsto l_1, \vec{x} \mapsto os_1\} \sim_{\vec{k_v'}, \beta} \{this \mapsto l_2, \vec{x} \mapsto os_2\}$$

thanks to Lemma E.1.5 and typability hypotheses

$$k_1 \leq \vec{k_v'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \vec{k_v'}[i+1]$$
$$k_2 \leq \vec{k_v'}[0] \qquad \forall i \in [0, \mathsf{length}(st_2) - 1],\ st_2[i] \leq \vec{k_v'}[i+1]$$

Now, since m' is non-interferent at order $\max(n_1', n_2') < n$ (with $n' + 1 = n_1$ and

$n_2' + 1 = n_2$) and

$$\langle 1, \{this \mapsto l_1, \vec{x} \mapsto os_1\}, \epsilon, h_1\rangle \Downarrow_{m'}^{(n_1')} r_1, h_1'$$

and

$$\langle 1, \{this \mapsto l_2, \vec{x} \mapsto os_2\}, \epsilon, h_2\rangle \Downarrow_{m'}^{(n_2')} r_2, h_2'$$

we deduce that

$$(r_1, h_1') \sim_{\beta', \vec{k_r'}} (r_2, h_2') \tag{3}$$

for some β' such that $\beta \subseteq \beta'$. The nature of r_1 and r_2 depends on the kind of transition that occurs:

— $\tau_1 = \emptyset$ and $\tau_2 = e \in \mathcal{C}$: e is caught or uncaught in m but in both cases, (3) gives $\vec{k_r'}[e] \not\leq k_{\text{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \emptyset),\ k_1 \sqcup k_e \leq se(j)$ with $k_e = \bigsqcup \left\{ \vec{k_r'}[e_0] \mid e_0 \in \mathsf{excAnalysis}(m_{\text{ID}}) \right\}$ and $\forall j \in \mathsf{region}(i, e),\ k_2 \sqcup \vec{k_r'}[e] \leq se(j)$. Hence se is high in $\mathsf{region}(i, \emptyset)$ and in $\mathsf{region}(i, e)$ since $\vec{k_r'}[e] \not\leq k_{\text{obs}}$ and by the same way $k_e \not\leq k_{\text{obs}}$.

— $\tau_1 = e_1 \in \mathcal{C}$ and $\tau_2 = e_2 \in \mathcal{C}$: branching only occurs if $e_1 \neq e_2$. But since $(\langle l_1'\rangle, h_1') \sim_{\beta', \vec{k_r'}} (\langle l_2'\rangle, h_2')$ with $e_j = \mathsf{class}(h_j'(l_j'))$, $j \in \{1, 2\}$, we have necessarily $\vec{k_r}[e_1] \not\leq k_{\text{obs}}$ and $\vec{k_r}[e_2] \not\leq k_{\text{obs}}$.

By typability, $\forall j \in \mathsf{region}(i, e_1),\ k_1 \sqcup \vec{k_r'}[e_1] \leq se(j)$ and $\forall j \in \mathsf{region}(i, e_2),\ k_2 \sqcup \vec{k_r'}[e_2] \leq se(j)$ so se is high in region $\mathsf{region}(i, e_1)$ and $\mathsf{region}(i, e_2)$.

Let $j \in \{1, 2\}$, If e_j is uncaught there is nothing to prove. If e_j is caught we must establish that $\mathsf{high}(l_j' :: \epsilon, (k_j \sqcup \vec{k_r'}[e_j]) :: \varepsilon)$. This is easy since $\vec{k_r}[e_j] \not\leq k_{\text{obs}}$.

— in the others cases, either there is no branching or the case is symmetric to a previous one.

Case:. $P_m[i] = \mathsf{getfield}\ f$. Branching only occurs if s_1 is of the form $\langle i, \rho_1, null :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, l_2 :: os_2, h_2\rangle$ (or in the symmetric case). Hence $\tau_1 = \mathbf{np}$ and $\tau_2 = \emptyset$ and by typability, $st_1 = k_1 :: st_1''$ and $st_2 = k_2 :: st_2''$.

Since $null :: os_1 \sim_{k_1 :: st_1'', k_2 :: st_2'', \beta} l_2 :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \mathbf{np}),\ k_1 \leq se(j)$ and $\forall j \in \mathsf{region}(i, \emptyset),\ k_2 \leq se(j)$. Hence se is high in region $\mathsf{region}(i, \mathbf{np})$ and $\mathsf{region}(i, \emptyset)$.

Concerning operand stacks, we have by typability $st_2' = \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st_2'')$ and since $k_2 \not\leq k_{\text{obs}}$ we deduce that $\mathsf{high}(os_2', \mathrm{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st_2''))$.

In the first transition there is something to prove only if $\mathbf{np}$ is caught. We must then establish that $\mathsf{high}(l_1' :: \epsilon, (k_1 \sqcup se(i)) :: \epsilon)$. Since $k_1 \not\leq k_{\text{obs}}$ we are done.

Case:. $P_m[i] = \mathsf{putfield}\ f$. Branching only occurs if s_1 is of the form $\langle i, \rho_1, v_1 :: null :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, v_2 :: l_2 :: os_2, h_2\rangle$ (or in the symmetric case). Hence $\tau_1 = \mathbf{np}$ and $\tau_2 = \emptyset$ and by typability, $st_1 = k_1 :: k_1' :: st_1''$ and $st_2 = k_2 :: k_2' :: st_2''$.

Since $v_1 :: null :: os_1 \sim_{k_1 :: k_1' :: st_1'', k_2 :: k_2' :: st_2'', \beta} v_2 :: l_2 :: os_2$, we have necessarily $k_1' \not\leq k_{\text{obs}}$ and $k_2' \not\leq k_{\text{obs}}$. By typability, $\forall j \in \mathsf{region}(i, \mathbf{np}),\ k_1' \leq se(j)$ and $\forall j \in \mathsf{region}(i, \emptyset),\ k_2' \leq se(j)$. Hence se is high in region $\mathsf{region}(i, \mathbf{np})$ and $\mathsf{region}(i, \emptyset)$.

Concerning operand stacks, we have by typability $st_2' = \mathrm{lift}_{k_2'}(st_2'')$ and since $k_2' \not\leq k_{\text{obs}}$ we deduce that $\mathsf{high}(os_2, \mathrm{lift}_{k_2'}(:: st_2''))$.

In the first transition there is something to prove only if $\mathbf{np}$ is caught. We must then establish that $\mathsf{high}(l_1' :: \epsilon, ('k_1 \sqcup se(i)) :: \epsilon)$. Since $k_1' \not\leq k_{\text{obs}}$ we are done.

Case:. $P_m[i] = \mathsf{throw}$. s_1 is of the form $\langle i, \rho_1, v_1 :: os_1, h_1\rangle$ and s_2 of the form $\langle i, \rho_2, v_2 :: os_2, h_2\rangle$ with v_1 and v_2 in $\mathcal{L} \cup \{\mathbf{np}\}$. By typability, $st_1 = k_1 :: st_1''$ and $st_2 = k_2 :: st_2''$.

We then make a first distinction according if one of v_1 or v_2 is equal to *null*. In both cases we show that $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$.

Case 1: v_1 or v_2 is equal to *null*. There is only a branching if the other value is not null. We only make the case $v_1 = null$ and $v_2 = l_2 \in \mathcal{L}$, the other case is done by symmetry.
By hypothesis,

$$null :: os_1 \sim_{k_1::st_1'',k_2::st_2'',\beta} l_2 :: os_2$$

Hence we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$.

Case 2: both v_1 and v_2 are in $\mathcal{L}$ (we note them l_1 and l_2 from now). There is a branching only if $h_1(l_1)$ and $h_2(l_2)$ are of distinct classes. This has for consequence that $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$, since by hypothesis

$$l_1 :: os_1 \sim_{k_1::st_1'',k_2::st_2'',\beta} l_2 :: os_2$$

Now, by typability we have $\forall j \in \mathsf{region}(i,\tau_1),\ k_1 \leq se(j)$ and $\forall j \in \mathsf{region}(i,\tau_2),\ k_2 \leq se(j)$. Hence se is high in region $\mathsf{region}(i,\tau_1)$ and $\mathsf{region}(i,\tau_2)$.

Concerning operand stacks, for $i \in \{1,2\}$ we have something to prove only if τ_i is caught. We must then establish that $\mathsf{high}(l_i :: \epsilon, (k_i \sqcup se(i)) :: \epsilon)$. Since $k_i \not\leq k_{\text{obs}}$ we are done.

□

LEMMA D.0.26 $\text{JVM}_{\mathcal{G}}$ LOCALLY RESPECT. *Let n an integer such that all method m' in P are non-interferent at all order k, $k < n$, with respect to all the policies in* $\text{Policies}_\Gamma(\text{m}')$.

Let m a method in P, $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ a partial function, $s_1, s_2 \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ states at the same program point i and two stack types $st_1, st_2 \in \mathcal{S}^\star$ such that $s_1 \sim_{st_1,st_2,\beta} s_2$.

(1) *If there exists two states $s_1', s_2' \in \text{State}_{\mathcal{G}}$ and two stack types $st_1', st_2' \in \mathcal{S}^\star$ such that*

$$\begin{gathered} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} s_1', \quad n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} s_2', \quad n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st_1' \quad i \vdash^{\tau_2} st_2 \Rightarrow st_2' \end{gathered}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$s_1' \sim_{st_1',st_2',\beta} s_2' \quad \textit{and} \quad \beta \subseteq \beta'$$

(2) *If there exists a state $\langle i_1', \rho_1', os_1', h_1' \rangle \in \text{State}_{\mathcal{G}}$, a final result $(r_2, h_2') \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ and a stack types $st_1', \in \mathcal{S}^\star$ such that*

$$\begin{gathered} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} \langle i_1', \rho_1', os_1', h_1' \rangle \quad n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h_2') \quad n_2 \leq n \\ i \vdash^{\tau_1} st_1 \Rightarrow st_1' \quad i \vdash^{\tau_2} st_2 \Rightarrow \end{gathered}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$h_1' \sim_{\beta'} h_2', \quad \mathsf{highResult}_{\vec{k_r}}(r_2, h_2') \quad \textit{and} \quad \beta \subseteq \beta'$$

(3) *If there exists two final results $(r_1, h_1'), (r_2, h_2') \in (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap}$ such that*

$$\begin{gathered} s_1 \overset{(n_1)}{\rightsquigarrow}_{m,\tau_1} (r_1, h_1') \quad n_1 \leq n \\ s_2 \overset{(n_2)}{\rightsquigarrow}_{m,\tau_2} (r_2, h_2') \quad n_2 \leq n \\ i \vdash st_1 \Rightarrow \quad i \vdash st_2 \Rightarrow \end{gathered}$$

then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$(r_1, h_1') \sim_{\vec{k_r},\beta'} (r_2, h_2') \quad \textit{and} \quad \beta \subseteq \beta'$$

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

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

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, os_2, h_2\rangle$, and $s_1' = \langle i+1, \rho_1, n :: os_1, h_1\rangle$, and $s_2' = \langle i+1, \rho_2, n :: os_2, h_2\rangle$. By typability, $st_1' = se(i) :: st_1$ and $st_2' = se(i) :: st_2$.

We take $\beta' = \beta$. Local variable and heaps are not modified so indistinguishability properties on them still hold. For operand stacks, by hypothesis we have $os_1 \sim_{st_1,st_2,\beta} os_2$. Hence and since $n \sim_\beta n$ holds, we have $n :: os_1 \sim_{se(i)::st_1,se(i)::st_2,\beta} n :: os_2$, thanks to Lemma E.1.2.

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

By semantics, $s_1 = \langle i, \rho_1, n_1 :: n_2 :: os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, n_1' :: n_2' :: os_2, h_2\rangle$ and $s_1' = \langle i+1, \rho_1, n :: os_1, h_1\rangle$, and $s_2' = \langle i+1, \rho_2, n' :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: k_2 :: st$, $st_1' = k_1 \sqcup k_2 \sqcup se(i) :: st$, $st_2 = k_1' :: k_2' :: st'$ and $st_2' = k_1' \sqcup k_2' \sqcup se(i) :: st'$.

We take $\beta' = \beta$. Local variable and heaps are not modified so indistinguishability properties on them still hold. For operand stacks, we make two cases:

— $k_1 \sqcup k_2 \leq k_{\text{obs}}$. In this case, since $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$, Lemma E.1.4 and hypothesis $n_1 :: n_2 :: os_1 \sim_{\beta,k_1::k_2::st,k_1'::k_2'::st'} n_1' :: n_2' :: os_2$ give $k_1 = k_1'$, $k_2 = k_2'$, $n_1 = n_1'$, $n_2 = n_2'$ and $os_1 \sim_{st,st'} os_2$. Hence $n = n'$ and $k_1 \sqcup k_2 \sqcup se(i) = k_1' \sqcup k_2' \sqcup se(i)$ so $n :: os_1 \sim_{k_1' \sqcup k_2' \sqcup se(i)::st',k_1' \sqcup k_2' \sqcup se(i)::st',\beta} n' :: os_2$ by Lemma E.1.2.

— $k_1 \sqcup k_2 \not\leq k_{\text{obs}}$. Hence $k_1 \not\leq k_{\text{obs}}$ or $k_2 \not\leq k_{\text{obs}}$ and thanks to Lemma E.1.4, $k_1' \not\leq k_{\text{obs}}$ or $k_2' \not\leq k_{\text{obs}}$. In both cases $k_1' \sqcup k_2' \not\leq k_{\text{obs}}$. We hence conclude that $k_1 \sqcup k_2 \sqcup se(i) \not\leq k_{\text{obs}}$ and $k_1' \sqcup k_2' \sqcup se(i) \not\leq k_{\text{obs}}$. Since $os_1 \sim_{st,st'} os_2$, we finally obtain $n :: os_1 \sim_{k_1' \sqcup k_2' \sqcup se(i)::st',k_1' \sqcup k_2' \sqcup se(i)::st',\beta} n' :: os_2$.

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

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, os_2, h_2\rangle$, and $s_1' = \langle i+1, \rho_1, \rho_1(x) :: os_1, h_1\rangle$, and $s_2' = \langle i+1, \rho_2, \rho_2(x) :: os_2, h_2\rangle$. By typability, $st_1' = \vec{k_v}_m(x) \sqcup se(i) :: st_1$ and $st_2' = \vec{k_v}_m(x) \sqcup se(i) :: st_2$.

We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability properties on them still hold.

For operand stacks, by hypothesis we have $os_1 \sim_{st_1,st_2,\beta} os_2$ and also by hypothesis of variable indistinguishability we have $\rho_1(x) \sim_\beta \rho_2(x)$ if $\vec{k_v}(x) \leq k_{\text{obs}}$. We hence conclude by Lemma E.1.1.

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

By semantics, $s_1 = \langle i, \rho_1, v :: os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, v' :: os_2, h_2\rangle$, and $s_1' = \langle i+1, \rho_1 \oplus \{x \mapsto v\}, os_1, h_1\rangle$, and $s_2' = \langle i+1, \rho_2 \oplus \{x \mapsto v'\}, os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st_1'$ and $st_2 = k_2 :: st_2'$.

We take $\beta' = \beta$. Heaps are not modified so indistinguishability properties on them still hold.

For local variables we have to check for every variable y that $\rho_1 \oplus \{x \mapsto v\}(y) \sim_\beta \rho_2 \oplus \{x \mapsto v'\}(y)$ whenever $\vec{k_v}(y) \leq k_{\text{obs}}$. This is valid by hypothesis for every variable except x. For x we must prove that $v \sim_\beta v'$ whenever $\vec{k_v}(x) \leq k_{\text{obs}}$. We then make a case analysis using Lemma E.1.4 on hypothesis $v :: os_1 \sim_{k_1::st_1',k_2::st_2',\beta} v' :: os_2$.

— In first case $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v \sim_\beta v'$. This last hypothesis allows us to conclude our proof of $\rho_1 \sim_\beta \rho_2$.

— In second case $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$. By typability, we have $k_1 \leq \vec{k_v}(x)$ and $k_2 \leq \vec{k_v}(x)$. Hence $\vec{k_v}(x) \not\leq k_{\text{obs}}$ and we can conclude our proof of $\rho_1 \sim_\beta \rho_2$.

For operand stacks, $os_1 \sim_{st_1',st_2',\beta} os_2$ easily come from Lemma E.1.4.

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

By semantics, $s_1 = \langle i, \rho_1, n :: os_1, h_1\rangle$, $s_2 = \langle i, \rho_2, n' :: os_2, h_2\rangle$, $s'_1 = \langle i'_1, \rho_1, os_1, h_1\rangle$ (where i'_1 can be $i+1$ or j) and $s'_2 = \langle i'_2, \rho_2, os_2, h_2\rangle$ (where i'_2 can be $i+1$ or j). By typability, $st_1 = k :: st$, $st_2 = k' :: st'$, $st'_1 = \mathrm{lift}_k(st)$ and $st'_2 = \mathrm{lift}_{k'}(st')$.

We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability properties on them still hold.

Operand stack indistinguishability $os_1 \sim_{st,st',\beta} os_2$ finally holds because of hypothesis $n :: os_1 \sim_{k::st,k'::st',\beta} n' :: os_2$ and Lemma E.1.4.

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

This case is trivial. We take $\beta' = \beta$. Neither local variables, operand stacks or heaps are modified so indistinguishability properties on them still hold.

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

By semantics, $s_1 = \langle i, \rho_1, v :: os_1, h_1\rangle$, and $s_2 = \langle i, \rho_2, v' :: os_2, h_2\rangle$, and $(r_1, h'_1) = (v, h_1)$, and $(r_2, h'_2) = (v', h_2)$. By typability, st_1 is of the form $k :: st$ and st_2 is of the form $k' :: st'$.

We have to prove $(v, h_1) \sim_{\beta,\vec{k_r}} (v', h_2)$, that is, $h_1 \sim_\beta h_2$ and $\vec{k_r}[n] \leq k_{\mathrm{obs}} \Rightarrow v \sim_\beta v'$.

Heap indistinguishability holds by hypothesis. We then make a case analysis on hypothesis $v :: os_1 \sim_{k::st,k'::st',\beta} v' :: os_2$ using Lemma E.1.4.

— In first case $k = k'$, $k \leq k_{\mathrm{obs}}$ and $v \sim_\beta v'$. This last hypothesis allows us to prove $\vec{k_r}[n] \leq k_{\mathrm{obs}} \Rightarrow v \sim_\beta v'$.

— In second case $k \leq k_{\mathrm{obs}}$ and $k' \leq k_{\mathrm{obs}}$. Hence $\vec{k_r}[n] \not\leq k_{\mathrm{obs}}$ since by typability, $k \leq \vec{k_r}[n]$ and $k' \leq \vec{k_r}[n]$. So $\vec{k_r}[n] \leq k_{\mathrm{obs}} \Rightarrow v \sim_\beta v'$ is trivially true.

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

By semantics, $s_1 = \langle i, \rho_1, os_1, h_1\rangle$, $s_2 = \langle i, \rho_2, os_2, h_2\rangle$, $s'_1 = \langle i+1, \rho_1, l :: os_1, h_1 \oplus\{l \mapsto \mathsf{default}_C\}\rangle$, and $s'_2 = \langle i+1, \rho_2, l' :: os_2, h_2 \oplus\{l' \mapsto \mathsf{default}_C\}\rangle$, where $l = \mathrm{fresh}(h_1)$, and $l' = \mathrm{fresh}(h_2)$. By typability, $st'_1 = se(i) :: st_1$ and $st'_2 = se(i) :: st_2$.

We first choose β' according to a case analysis on $se(i)$. If $se(i) \leq k_{\mathrm{obs}}$, we choose $\beta' = \beta \oplus\{l \mapsto l'\}$. If $se(i) \not\leq k_{\mathrm{obs}}$, we choose $\beta' = \beta$. In both case $\beta \subseteq \beta'$ thanks to Lemma E.2.5.

Local variables are not modified so indistinguishability properties on them still hold for β', thanks to Lemma E.3.2.

For operand stack indistinguishability, $os_1 \sim_{\beta,st_1,st_2} os_2$ and since $\beta \subseteq \beta'$, $os_1 \sim_{\beta',st_1,st_2} os_2$ also holds, thanks to Lemma E.3.3. If $se(i) \not\leq k_{\mathrm{obs}}$, we obtain by definition of operand stack indistinguishability, $l :: os_1 \sim_{\beta',se(i)::st_1,se(i)::st_2} l' :: os_2$. If $se(i) \leq k_{\mathrm{obs}}$, we only have to prove $l \sim_{\beta'} l'$ to conclude. This follow easily from definition of β' since $\beta'(l) = l'$.

Heap indistinguishability comes from Lemma E.2.5 when $se(i) \leq k_{\mathrm{obs}}$ and from Lemma E.2.6 when $se(i) \not\leq k_{\mathrm{obs}}$.

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

By semantics, $s_1 = \langle i_1, \rho_1, v :: l :: os_1, h_1\rangle$ and $s_2 = \langle i_2, \rho_2, v' :: l' :: os_2, h_2\rangle$, but there are several options to consider for the successors of s_1 and s_2:

— There are no exceptions ($\tau_1 = \emptyset$ and $\tau_2 = \emptyset$) and

$$s'_1 = \langle i_1 + 1, \rho_1, os_1, h_1 \oplus\{l \mapsto h_1(l) \oplus\{f \mapsto v\}\}\rangle$$
$$s'_2 = \langle i_1 + 1, \rho_2, os_2, h_2 \oplus\{l' \mapsto h_2(l') \oplus\{f \mapsto v'\}\}\rangle$$

By typability, $st_1 = k_1 :: k'_1 :: st$, $st_2 = k_2 :: k'_2 :: st'$, $st'_1 = \mathsf{lift}_{k'_1} st$ and $st'_2 = \mathsf{lift}_{k'_2} st'$.

We take $\beta' = \beta$.

By hypothesis and semantics (variables do not change) we have indistinguishability of variables $\rho_1 \sim_\beta \rho_2$.

We know by hypothesis that $v :: l :: os_1 \sim_{\beta,st_1,st_2} v' :: l' :: os_2$ and hence, by Lemma E.1.4, $os_1 \sim_{\beta,st'_1,st'_2} os_2$. We then make a case analysis using Lemma E.1.4.
— either $k'_1 = k'_2$, $k'_1 \leq k_{\text{obs}}$ and $l \sim_\beta l'$. In this case we conclude operand stack indistinguishability $os_1 \sim_{\beta,\mathsf{lift}_{k'_1} st,\mathsf{lift}_{k'_2} st'} os_2$ by Lemma E.1.3
— or $k'_1 \not\leq k_{\text{obs}}$ and $k'_2 \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(os_1, \mathsf{lift}_{k'_1} st)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k'_2} st')$, and finally conclude about operand stack indistinguishability.
To check indistinguishability of heaps we remark that h_1 is only updated in location l and field f, and similarly h_2 is only updated in location l' and field f. Hence, if $\mathsf{ft}(f) \not\leq k_{\text{obs}}$ heap indistinguishability still holds. If $\mathsf{ft}(f) \leq k_{\text{obs}}$ we have $k_1 \leq k_{\text{obs}}$ and $k_2 \leq k_{\text{obs}}$ since by typability, the constraint $k_1 \leq \mathsf{ft}(f)$ and $k_2 \leq \mathsf{ft}(f)$ hold. Hence the Lemma E.1.4 gives us $v \sim_\beta v'$ and $l \sim_\beta l'$. This allows us to conclude about heap indistinguishability.

— One execution is normal ($\tau_2 = \emptyset$) and there is a null pointer exception in the other execution ($\tau_1 = \mathbf{np}$) , there is a handler t in m, and $s'_1 = \langle t, \rho_1, l'' :: \epsilon, h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$ and $s'_2 = \langle i_1 + 1, \rho_2, os_2, h_2 \oplus\{o \mapsto h_2(l') \oplus\{f \mapsto v'\}\}\rangle$. By typability, $st_1 = k_1 :: k'_1 :: st$, $st_2 = k_2 :: k'_2 :: st'$, $st'_2 = \mathsf{lift}_{k'_2} st'$ and $st'_1 = k'_1 \sqcup se(i) :: \epsilon$.
By hypothesis and semantics (variables do not change) we have indistinguishability of variables $\rho_1 \sim_\beta \rho_2$.
Because of $v :: null :: os_1 \sim_{k_1::k'_1::st,k_2::k'_2::st',\beta} v' :: l' :: os_2$ and Lemma E.1.4 we have necessarily $k'_1 \not\leq k_{\text{obs}}$ and $k'_2 \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(l'' :: \epsilon, k'_1 \sqcup se(i) :: \epsilon)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k'_2} st'_2)$, and finally conclude about operand stack indistinguishability $l'' :: \epsilon \sim_{\beta,k'_1 \sqcup se(i)::\epsilon,\mathsf{lift}_{k'_2} st'} os_2$.
To check indistinguishability of

$$h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_\beta h_2 \oplus\{l' \mapsto h_2(l') \oplus\{f \mapsto v'\}\}$$

we first invoke Lemma E.2.6 (l'' is a fresh location for h_1) to establish

$$h_1 \oplus\{l'' \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_\beta h_2$$

By typability, $k'_1 \leq \mathsf{ft}(f)$ so $\mathsf{ft}(f) \not\leq k_{\text{obs}}$. This allows to conclude about heap indistinguishability.

— One execution is normal ($\tau_2 = \emptyset$) and there is a null pointer exception ($\tau_1 = \mathbf{np}$), but no handler for it in m, and $s'_1 = \langle\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathbf{default}_{\mathbf{np}}\}\rangle$; and s'_2 is equal to $\langle i_1 + 1, \rho_2, os_2, h_2 \oplus\{o \mapsto h_2(l') \oplus\{f \mapsto v'\}\}\rangle$. By typability, $st_1 = k_1 :: k'_1 :: st$, $st_2 = k_2 :: k'_2 :: st'$, $st'_2 = \mathsf{lift}_{k'_2} st'$.
We take $\beta' = \beta$.
Heap indistinguishability holds for the same reason as in the previous case.
Because of $v :: null :: os_1 \sim_{k_1::k'_1::st,k_2::k'_2::st',\beta} v' :: l' :: os_2$ and Lemma E.1.4 we have necessarily $k'_1 \not\leq k_{\text{obs}}$ and $k'_2 \not\leq k_{\text{obs}}$. By typability, we have the constraint $k'_1 \leq \vec{k_r}[\mathbf{np}]$. Hence $\vec{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$ and $\mathsf{highResult}_{\vec{k_r}}(\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathbf{default}_{\mathbf{np}}\})$ hold.

— There are two null pointer exceptions due to putfield and there is no handler for them in m, s'_1 is equal to $\langle\langle l_1\rangle, h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$; and s'_2 is equal to $\langle\langle l_2\rangle, h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$. By typability, $st_1 = k_1 :: k'_1 :: st$ and $st_2 = k_2 :: k'_2 :: st'$.
We take $\beta =' \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 gives us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
We then make case analysis on $\vec{k_r}[\mathbf{np}]$.
— if $\vec{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$, we have easily $(\langle l_1\rangle, h'_1) \sim_{\vec{k_r},\beta'} (\langle l_2\rangle, h'_2)$.
— if $\vec{k_r}[\mathbf{np}] \leq k_{\text{obs}}$, we have $(\langle l_1\rangle, h'_1) \sim_{\vec{k_r},\beta'} (\langle l_2\rangle, h'_2)$ thanks to $l_1 \sim_{\beta'} l_2$ (since $\beta'(l_1) = l_2$).

— There are two null pointer exceptions and there is a handler for null pointer exception and program point i, namely t, s'_1 is equal to $\langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus\{l_1 \mapsto \mathsf{default_{np}}\}\rangle$ and $s'_2 = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus\{l_2 \mapsto \mathsf{default_{np}}\}\rangle$, $st_1 = k_1 :: k'_1 :: st$, $st_2 = k_2 :: k'_2 :: st'$, $st'_1 = k'_1 \sqcup se(i) :: \epsilon$, and $st'_2 = k'_2 \sqcup se(i) :: \epsilon$.
We take $\beta =' \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 gives us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default_{np}}\}$.
Local variables are not modified, so by Lemma E.3.2, local variable indistinguishability hold.
We finally have to prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k'_1 \sqcup se(i)::\epsilon, k'_2 \sqcup se(i)::\epsilon, \beta'} l_2 :: \epsilon$. We make a case analysis on hypothesis $v :: null :: os_1 \sim_{\beta, st_1, st_2} v' :: null :: os_2$ with the help of Lemma E.1.4:
— either $k'_1 = k'_2$ and $k'_1 \leq k_{\text{obs}}$. In this case, since $l_1 \sim_{\beta'} l_2$ and $k'_1 \sqcup se(i) = k'_2 \sqcup se(i)$ we are done.
— or $k'_1 \not\leq k_{\text{obs}}$ and $k'_2 \not\leq k_{\text{obs}}$ and operand stack indistinguishability trivially holds.

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

By semantics, $s_1 = \langle i_1, \rho_1, l :: os_1, h_1\rangle$ and $s_2 = \langle i_2, \rho_2, l' :: os_2, h_2\rangle$ and there are several options to consider successors of s_1 and s_2,

— There are no exceptions and $s'_1 = \langle i_1 + 1, \rho_1, v :: os_1, h_1\rangle$ and $s'_2 = \langle i_1 + 1, \rho_2, v' :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st'_1 = \text{lift}_{k_1}((\mathsf{ft}(f) \sqcup se(i)) :: st)$ and $st'_2 = \text{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.
We take $\beta' = \beta$. Local variables and heaps are not modified so indistinguishability still hold for them.
By hypothesis, $v :: os_1 \sim_{k_1::st, k_2::st', \beta} v' :: os_2$ so $os_1 \sim_{st, st', \beta} os_2$ holds too. We then make a case analysis using Lemma E.1.4.
— either $k_1 = k_2$ and $k_1 \leq k_{\text{obs}}$. In this case we conclude operand stack indistinguishability $os_1 \sim_{\mathsf{lift}_{k_1} st, \mathsf{lift}_{k_2} st', \beta} os_2$ by Lemma E.1.3,
— or $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We can hence claim that $\mathsf{high}(os_1, \mathsf{lift}_{k_1} st)$ and $\mathsf{high}(os_2, \mathsf{lift}_{k_2} st')$, and finally conclude about operand stack indistinguishability.

— One execution is normal and there is a null pointer exception in the other execution, there is a handler t in m, and $s'_1 = \langle t, \rho_1, l'' :: \epsilon, h_1 \oplus\{l'' \mapsto \mathbf{default_{np}}\}\rangle$ and s'_2 is equal to $\langle i_1{+}1, \rho_2, v' :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st'_1 = k_1 \sqcup se(i) :: \epsilon$ and $st'_2 = \text{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.
We take $\beta' = \beta$. Local variables are not modified so indistinguishability still hold for them.
By hypothesis $null :: os_1 \sim_{k_1::st, k_2::st', \beta} l' :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. It follows that $\mathsf{high}(l'' :: \epsilon, k_1 \sqcup se(i) :: \epsilon)$ and $\mathsf{high}(v' :: os_2, \text{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st'))$.
Heap indistinguishability hold by Lemma E.2.6.

— One execution is normal and there is a null pointer exception, but no handler for it in m, $s'_1 = \langle\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathsf{default_{np}}\}\rangle$ and $s'_2 = \langle i_1 + 1, \rho_2, v :: os_2, h_2\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, and $st'_2 = \text{lift}_{k_2}((\mathsf{ft}(f) \sqcup se(i)) :: st')$.
Heap indistinguishability holds by Lemma E.2.6.
By hypothesis $null :: os_1 \sim_{k_1::st, k_2::st', \beta} l' :: os_2$, we have necessarily $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. By typability, $k_1 \leq \vec{k_r}[\mathbf{np}]$. Hence we deduce that $\vec{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$ and $\mathsf{highResult}_{\vec{k_r}}(\langle l''\rangle, h_1 \oplus\{l'' \mapsto \mathsf{default_{np}}\})$ holds.

— Two null pointer exceptions and no handler for them in i,

$$s'_1 = \langle\langle l_1\rangle, h_1 \oplus\{l_1 \mapsto \mathsf{default_{np}}\}\rangle \text{ and } s'_2 = \langle\langle l_2\rangle, h_2 \oplus\{l_2 \mapsto \mathsf{default_{np}}\}\rangle$$

By typability, $st_1 = k_1 :: st$ and $st_2 = k_2 :: st'$.
We take $\beta =' \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 gives us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default_{np}}\}$.

We then make case analysis on $\vec{k_r}[\mathbf{np}]$.
— if $\vec{k_r}[\mathbf{np}] \not\leq k_{\text{obs}}$, we have easily $(\langle l_1\rangle, h_1') \sim_{\vec{k_r},\beta'} (\langle l_2\rangle, h_2')$.
— if $\vec{k_r}[\mathbf{np}] \leq k_{\text{obs}}$, we have $(\langle l_1\rangle, h_1') \sim_{\vec{k_r},\beta'} (\langle l_2\rangle, h_2')$ thanks to $l_1 \sim_{\beta'} l_2$ (since $\beta'(l_1) = l_2$).

— Two exceptions with handler: $s_1' = \langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$ and $s_2' = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}\rangle$. By typability, $st_1 = k_1 :: st$, $st_2 = k_2 :: st'$, $st_1' = k_1 :: \epsilon$ and $st_2' = k_2 :: \epsilon$.
We take $\beta =' \beta \oplus\{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 gives us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus\{l_1 \mapsto \mathsf{default}_{\mathbf{np}}\} \sim_{\beta'} h_2 \oplus\{l_2 \mapsto \mathsf{default}_{\mathbf{np}}\}$.
Local variables are not modified, so by Lemma E.3.2, local variable indistinguishability hold.
We finally have to prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k_1::\epsilon,k_2::\epsilon,\beta'} l_2 :: \epsilon$. We make a case analysis on hypothesis $null :: os_1 \sim_{\beta,st_1,st_2} null :: os_2$ with the help of Lemma E.1.4:
— either $k_1 = k_2$ and $k_1 \leq k_{\text{obs}}$. In this case, since $l_1 \sim_{\beta'} l_2$ and $k_1 = k_2$ so we are done.
— or $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$ and operand stack indistinguishability trivially holds.

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

By semantics, $s_1 = \langle i, \rho_1, v_1 :: os_1, h_1\rangle$ and $s_2 = \langle i, \rho_2, v_2 :: os_2, h_2\rangle$ with v_1 and v_2 in $\mathcal{L} \cup \{null\}$. By typability, $st_1 = k_1 :: st_1''$ and $st_2 = k_2 :: st_2''$. There is a successor state in the current method execution if and only if the thrown exception is caught. In this case st_j' ($j \in \{1, 2\}$) is of the form $st_j' = k_j \sqcup se(i) :: \epsilon$.

By hypothesis, $v_1 :: os_1 \sim_{k_1::st_1'',k_2::st_2'',\beta} v_2 :: os_2$. We then make a case analysis using Lemma E.1.4:
— If $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v_1 \sim_\beta v_2$, there are two cases to consider.
— $v_1 = v_2 = null$: this case is exactly the same as for $\mathsf{getfied}$ instruction when both operand stacks of s_1 and s_2 have a null pointer on top of there stacks.
— $v_1 = l_1$, $v_2 = l_2$ and $l_1 \sim_\beta l_2$: since $h_1 \sim_\beta h_2$ we deduce that $h_1(l_1)$ and $h_2(l_2)$ have the same class. Hence we are either in case 1 of the lemma statement (when the exception is caught) or in case 3 (when the exception is uncaught). In both cases we choose $\beta' = \beta$.
— if the exception is caught : by semantics, $s_1' = \langle t, \rho_1, l :: \epsilon, h_1\rangle$ and $s_2' = \langle t, \rho_2, l :: \epsilon, h_2\rangle$. Heaps and local variables are not modified so there is only something to prove for operand stacks, that is: $l_1 :: \epsilon \sim_{k_1 \sqcup se(i)::\epsilon,k_2 \sqcup se(i)::\epsilon,\beta} l_2 :: \epsilon$. This is true since $k_1 \sqcup se(i) = k_2 \sqcup se(i)$and $v_1 \sim_\beta v_2$.
— if the exception is uncaught, we must establish $(\langle l_1\rangle, h_1) \sim_{\vec{k_r},\beta} (\langle l_2\rangle, h_2)$. The current hypotheses contains $h_1 \sim_\beta h_2$ and $l_1 \sim_\beta l_2$ so output indistinguishability holds easily.

— If $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We choose $\beta' = \beta$. In each execution thrown exception is either caught or uncaught. We now examine the different cases (symmetric cases are skipped).
— Both exceptions are caught: each s_j' ($j \in \{1, 2\}$) is then of the form $s_j' = \langle t_j, \rho_j, l_j' :: \epsilon, h_j'\rangle$ with $h_j' = h_j$ (if $v_j \neq null$) or $h_j' = h_j \oplus\{l_j' \mapsto \mathbf{default}_{\mathbf{np}}\}$ (if $v_j \neq null$ and $l_j' = \text{fresh}(h_j)$).
Local variables are not modified so indistinguishability still hold for them.
Each st_j' is of the form $k_j \sqcup se(i) :: \epsilon$ and $k_j \not\leq k_{\text{obs}}$ so operand stack indistinguishability holds since we have $\mathsf{high}(l_1' :: \epsilon, k_1 \sqcup se(i) :: \epsilon)$ for each j.
Concerning heaps, $h_1' \sim_\beta h_2'$ holds even if one of the h_j' is of the form $h_j \oplus\{l_j' \mapsto \mathbf{default}_{\mathbf{np}}\}$ thanks to Lemma E.2.2.
— exception is caught in the first execution but not in the second: s_1' is then of the form $s_1' = \langle t, \rho_1, l_1' :: \epsilon, h_1'\rangle$ and s_2' of the form $(\langle l_2'\rangle, h_2')$ with $h_j' = h_j$ (if $v_j \neq null$) or $h_j' = h_j \oplus\{l_j' \mapsto \mathsf{default}_{\mathbf{np}}\}$ (if $v_j \neq null$ and $l_j' = \text{fresh}(h_j)$), $j \in \{1, 2\}$.

$h'_1 \sim_\beta h'_2$ holds even if one of the h'_j is of the form $h_j \oplus \{l'_j \mapsto \mathbf{default_{np}}\}$ by Lemma E.2.2.
We finally have to establish $\mathsf{highResult}_{\vec{k_r}}(l'_2, h'_2)$. By hypothesis, $\tau_2 = \mathsf{class}(h'_2(l'_2))$, $k_2 \leq \vec{k_r}[\tau_2]$ and $k_2 \not\leq k_{\text{obs}}$ so we are done.
— exception is uncaught in both executions: each s'_j ($j \in \{1,2\}$) is then of the form $(\langle l'_j \rangle, h'_j)$ with $h'_j = h_j$ (if $v_j \neq null$) or $h'_j = h_j \oplus \{l'_j \mapsto \mathbf{default_{np}}\}$ (if $v_j \neq null$ and $l'_j = \text{fresh}(h_j)$).
We must prove that $(\langle l'_1 \rangle, h'_1) \sim_{\beta', \vec{k'_r}} (\langle l'_2 \rangle, h'_2)$. It is sufficient to prove

$$h'_1 \sim_\beta h'_2 \qquad \vec{k_r}[\mathsf{class}(h'_1(l'_1))] \not\leq k_{\text{obs}} \qquad \vec{k_r}[\mathsf{class}(h'_2(l'_2))] \not\leq k_{\text{obs}}$$

$h'_1 \sim_\beta h'_2$ holds even if one of the h'_j is of the form $h_j \oplus \{l'_j \mapsto \mathbf{default_{np}}\}$ thanks to Lemma E.2.2. $\vec{k_r}[\mathsf{class}(h'_j(l'_j))]$ holds for each j since, by semantics $\tau_j = \mathsf{class}(h'_j(l'_j))$ and by typability, $k_j \leq \vec{k_r}[\tau_j]$ and $k_j \not\leq k_{\text{obs}}$.

Case:. $P_m[i] = \mathsf{invokevirtual}\ m_{\text{ID}}$. By semantics, each s_j is of the form $\langle i, \rho_j, os_j :: v_j :: os'_j, h_j \rangle$ with $v_j \in \mathcal{L} \cup \{null\}$. By typability, each st_j is of the form $st''_j :: k_j :: st'''_j$ with $\mathsf{length}(os_j) = \mathsf{length}(st_j)$.
By hypothesis,

$$os_1 :: v_1 :: os'_1 \sim_{st''_1 :: k_1 :: st'''_1, st''_2 :: k_2 :: st'''_2, \beta} os_2 :: v_2 :: os'_2$$

We then make a case analysis using Lemma E.1.4:
— If $k_1 = k_2$, $k_1 \leq k_{\text{obs}}$ and $v_1 \sim_\beta v_2$, there are two cases to consider.
— $v_1 = v_2 = null$: a null pointer exception is thrown. We must then examine if this exception is caught or not.
— If the exception is caught, $s'_1 = \langle t, \rho_1, l_1 :: \epsilon, h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\} \rangle$ and $s'_2 = \langle t, \rho_2, l_2 :: \epsilon, h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\} \rangle$. By typability, $st'_1 = st'_2 = (k_1 \sqcup \vec{k'_r}[\mathbf{np}]) :: \varepsilon$.
We take $\beta =' \beta \oplus \{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}$.
Local variables are not modified, so by Lemma E.3.2, local variable indistinguishability holds.
Finally, we prove operand stack indistinguishability $l_1 :: \epsilon \sim_{k_1 \sqcup \vec{k'_r}[\mathbf{np}] :: \epsilon, k_2 \sqcup \vec{k'_r}[\mathbf{np}] :: \epsilon, \beta'} l_2 :: \epsilon$ since $l_1 \sim_{\beta'} l_2$ and $k_1 = k_2$.
— If the exception is uncaught, $s'_1 = (\langle l_1 \rangle, h_1)$ and $s'_2 = (\langle l_2 \rangle, h_2)$.
We take $\beta' = \beta \oplus \{l_1 \mapsto l_2\}$. By semantics, l_1 and l_2 are fresh locations. Hence Lemma E.2.5 give us $\beta \subseteq \beta'$ and heap indistinguishability $h_1 \oplus \{l_1 \mapsto \mathsf{default_{np}}\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default_{np}}\}$.
We conclude on output indistinguishability $(\langle l_1 \rangle, h_1) \sim_{\vec{k_r}, \beta'} (\langle l_2 \rangle, h_2)$ using previous heap indistinguishability and the fact $l_1 \sim_{\beta'} l_2$.
— $v_1 = l_1$, $v_2 = l_2$ and $l_1 \sim_\beta l_2$: since $h_1 \sim_\beta h_2$ we deduce that $h_1(l_1)$ and $h_2(l_2)$ have the same class. We deduce that the same method m' is called in both executions. As in the proof of the previous lemma we then invoke non-interference at some order k, $k < n$ to establish that outputs of each executions of m' are indistinguishable for some β' such that $\beta \subseteq \beta'$:

$$(r_1, h'_1) \sim_{\beta', \vec{k'_r}} (r_2, h'_2) \tag{4}$$

There are then two cases to consider for each executions of the called method (normal termination of the called method or termination by an exception). We examine now these different cases, skipping symmetric cases. Heaps indistinguishable is not mentioned since it follows from (4).
— If both executions terminate normally $r_1 = v_1 \in \mathcal{V}$ and $r_2 = v_2 \in \mathcal{V}$. (4) gives $\vec{k'_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_{\beta'} v_2$. Indistinguishability of local variables is obtained using Lemma E.3.2. Operand stack indistinguishability $v_1 :: os'_1 \sim_{(\vec{k'_r}[n] \sqcup se(i)) :: st_1, (\vec{k'_r}[n] \sqcup se(i)) :: st_2, \beta'}$

$v_2 :: os_2'$ is finally obtained thanks to $\vec{k_r'}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_{\beta'} v_2$ and $os_1' \sim_{st_1,st_2,\beta'} os_2'$ (obtained using Lemma E.3.3).
— If both executions terminate with an exception e_1 and e_2, $r_1 = l_1' \in \mathcal{L}$ and $r_2 = l_2' \in \mathcal{L}$. (4) gives us two more cases:
In first case $\vec{k_r'}[e_1] \not\leq k_{\text{obs}}$ and $\vec{k_r'}[e_2] \not\leq k_{\text{obs}}$. If both exceptions are caught in m, $l_1' :: \epsilon \sim_{(k_1 \sqcup \vec{k_r'}[e_1])::\varepsilon,(k_2 \sqcup \vec{k_r'}[e_2])::\varepsilon,\beta'} l_2' :: \epsilon$ holds since stacks are high. If e_1 is caught and e_2 is uncaught, $\mathsf{highResult}_{\vec{k_r}}(l_2', h_2')$ holds thanks to typability constraint $k_2 \sqcup se(i) \sqcup \vec{k_r'}[e_2] \leq \vec{k_r}[e_2]$. If both exceptions are uncaught in m, $(\langle l_1' \rangle, h_1') \sim_{\vec{k_r},\beta'} (\langle l_2' \rangle, h_2')$ holds thanks to typability constraints $k_1 \sqcup se(i) \sqcup \vec{k_r'}[e_1] \leq \vec{k_r}[e_1]$ and $k_2 \sqcup se(i) \sqcup \vec{k_r'}[e_2] \leq \vec{k_r}[e_2]$.
In second case $e_1 = e_2$ and $l_1' \sim_{\beta'} l_2'$. If e_1 is caught, $l_1' :: \epsilon \sim_{(k_1 \sqcup \vec{k_r'}[e_1])::\varepsilon,(k_2 \sqcup \vec{k_r'}[e_2])::\varepsilon,\beta'} l_2' :: \epsilon$ holds since $(k_1 \sqcup \vec{k_r'}[e_1]) = (k_2 \sqcup \vec{k_r'}[e_2])$ and $l_1' \sim_{\beta'} l_2'$. If e_1 is uncaught, $(\langle l_1' \rangle, h_1') \sim_{\vec{k_r},\beta'} (\langle l_2' \rangle, h_2')$ holds since $l_1' \sim_{\beta'} l_2'$.

— $k_1 \not\leq k_{\text{obs}}$ and $k_2 \not\leq k_{\text{obs}}$. We choose $\beta' = \beta$. Each h_j' is either of the form $h_j \oplus \{l_j' \mapsto \mathsf{default_{np}}\}$ (if the virtual call is done on a null pointer) or is equal to the final heap obtained after execution of the called method m_j'. Since methods are supposed side-effect safe we know that $h_j \preceq h_j'$. In every cases $h_1' \sim_{\beta'} h_2'$ holds thanks to Lemma E.2.2 or Lemma E.2.4 (side effect levels are high by typability).
We then make a case analysis according to the nature (final or not) of each execution. Heaps indistinguishability is no more mentioned since already proved.
— if both executions go on, by typability all elements in st_1' (respectively st_2') are greater than k_1 (respectively k_2) and hence are high. Operand stack indistinguishability follows easily. Local variables indistinguishability is immediate since local variables are not modified.
— if one execution goes on but the other terminates with an uncaught exception e we must prove that $\vec{k_r}[e] \not\leq k_{\text{obs}}$. This is done using typability constraint $k \sqcup se(i) \sqcup \vec{k_r'}[e] \leq \vec{k_r}[e]$ with k equals to k_1 or k_2.
— If both executions terminate with some uncaught exception e_1 and e_2, output indistinguishability holds using typability constraint $k_1 \sqcup se(i) \sqcup \vec{k_r'}[e_1] \leq \vec{k_r}[e_1]$ and $k_2 \sqcup se(i) \sqcup \vec{k_r'}[e_2] \leq \vec{k_r}[e_2]$.

□

Lemma D.0.27 Non-interference induction step. *Let n an integer such that all method in P are non-interferent at all order k, $k < n$.*

Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, os_0, h_0 \rangle, \langle i_0', \rho_0', os_0', h_0' \rangle \in \text{State}_{\mathcal{G}}$ two $\text{JVM}_{\mathcal{G}}$ statessuch that $\langle i_0, \rho_0, os_0, h_0 \rangle \sim_{S_{i_0},S_{i_0'},\beta} \langle i_0', \rho_0', os_0', h_0' \rangle$, and $i_0 = i_0'$. Suppose we have a derivation

$$\langle i_0, \rho_0, os_0, h_0 \rangle \overset{(n_0)}{\rightsquigarrow}_{m,\tau_0} \cdots \langle i_k, \rho_k, os_k, h_k \rangle \overset{(n_k)}{\rightsquigarrow}_{m,\tau_k} (r, h)$$

with $n_0 + \cdots + n_k \leq n$ and suppose this derivation is typable with respect to S. Suppose we have a derivation

$$\langle i_0', \rho_0', os_0', h_0' \rangle \overset{(n_0')}{\rightsquigarrow}_{m,\tau_0'} \cdots \langle i_k', \rho_k', os_k', h_k' \rangle \overset{(n_k')}{\rightsquigarrow}_{m,\tau_k'} (r', h')$$

with $n_0' + \cdots + n_k' \leq n$ and suppose this derivation is typable with respect to S. Then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$(r, h) \sim_{\vec{k_r},\beta'} (r', h') \quad \textit{and} \quad \beta \subseteq \beta'$$

Proof. By induction on $\max(k, k')$. Suppose the statement is true for derivation of length strictly lower than $\max(k, k')$ and prove it for the current lengths.

There are four cases:

(1) $k = k' = 0$: the case 3 of Lemma D.0.26 allows us a direct conclusion.

(2) $k > 0$ and $k' = 0$: we are in the case 2 of Lemma D.0.26 so there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that

$$h_1 \sim_{\beta'} h', \quad \mathsf{highResult}_{\vec{k_r}}(r', h') \quad \text{and} \quad \beta \subseteq \beta'$$

Thanks to case 2 of lemma D.0.25 and sub-typing lemma E.4.1 we have

$$\mathsf{high}(os_1, S_{i_1}) \quad \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau_1)$$

where τ_1 verifies $i_0 \mapsto^{\tau_1} i_1$. SOAP2 gives $i_1 \in \mathsf{region}(i_0, \tau_1)$ or $i_1 = \mathsf{jun}(i_0, \tau_1)$ but $\mathsf{jun}(i_0, \tau_1)$ is undefined thanks to SOAP3. We can hence apply Lemma D.0.24 to conclude that

$$\mathsf{highResult}_{\vec{k_r}}(r', h') \quad \text{and} \quad h' \sim_{\beta} h'_1$$

Using the other hypotheses

$$h_1 \sim_{\beta'} h'_1, \quad h_1 \sim_{\beta'} h', \quad \mathsf{highResult}_{\vec{k_r}}(r', h')$$

we can easily conclude.

(3) $k = 0$ and $k' > 0$: symmetric version of the previous case.

(4) $k > 0$ and $k' > 0$: We make two cases:
Case 1:. $i_1 = i'_1$. The induction hypothesis directly applies.
Case 2:. $i_1 \neq i'_1$. Let call st_1, st'_1 the stack types such that

$$i_o \vdash^{\tau_0} S_{i_0} \Rightarrow st_1, \quad st_1 \sqsubseteq S_{i_1}, \quad i'_o \vdash^{\tau'_0} S_{i'_0} \Rightarrow st'_1, \quad st'_1 \sqsubseteq S_{i'_1}$$

The case 1 of lemma D.0.25 can be invoked to deduce (with the help of lemma E.4.1)

$$\mathsf{high}(os_1, st_1) \quad \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau)$$
$$\mathsf{high}(os'_1, st'_1) \quad \text{and} \quad se \text{ is high in } \mathsf{region}(i_0, \tau')$$

where τ, τ' verify $i_0 \mapsto^{\tau} i_1$ and $i'_0 \mapsto^{\tau'} i'_1$.
Thanks to lemma E.4.1 we have

$$\mathsf{high}(os_1, S_{i_1}) \quad \text{and} \quad \mathsf{high}(os'_1, S'_{i_1})$$

We are furthermore in case 1 of Lemma D.0.26 so there exists β', $\beta \subseteq \beta'$ such that

$$\langle i_1, \rho_1, os_1, h_1 \rangle \sim_{st_1, st'_1, \beta'} \langle i'_1, \rho'_1, os'_1, h'_1 \rangle$$

Using Lemma E.4.3 two times we have

$$\langle i_1, \rho_1, os_1, h_1 \rangle \sim_{S_{i_1}, S_{i'_1}, \beta'} \langle i'_1, \rho'_1, os'_1, h'_1 \rangle$$

This allows us to invoke Lemma D.0.22 which gives us two cases:

(a) There exists j, j' with $1 \leq j \leq k$ and $1 \leq j' \leq k'$ such that $i_j = i'_{j'}$ and $\langle i_j, \rho_j, os_j, h_j \rangle \sim_{S_{i_j}, S_{i'_{j'}}, \beta} \langle i'_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle$. We can use the induction hypothesis on $\langle i_j, \rho_j, os_j, h_j \rangle \overset{(n_j)}{\rightsquigarrow}_{m, \tau_j} (r, h)$ and $\langle i_{j'}, \rho_{j'}, os_{j'}, h_{j'} \rangle \overset{(n_{j'})}{\rightsquigarrow}_{m, \tau'_j} (r', h')$ to conclude.

(b) $(r, h) \sim_{\vec{k_r}, \beta'} (r', h')$ and we can directly conclude.

□

Proof of Theorem 7.5.1. We show by induction on an integer n that all method in P are non-interferent at order n. We use an induction scheme of the form

$$(\forall n,\ (\forall k, k < n \Rightarrow \mathcal{P}(k)) \ \Rightarrow\ \mathcal{P}(n)) \quad \Longrightarrow \quad \forall n,\ \mathcal{P}(n)$$

The proof is then a direct application of Lemma D.0.27.

□

E. AUXILIARIES LEMMAS

All free variables are implicitly universally quantified. Naming convention allows to infer kind of each variables.

E.1 Operand stack

LEMMA E.1.1. *If* $os \sim_{st,st',\beta} os'$ *and* $k \leq k_{\text{obs}} \Rightarrow v \sim_\beta v'$ *then* $v :: os \sim_{k::st,k::st',\beta} v' :: os'$

LEMMA E.1.2. $os \sim_{st,st',\beta} os'$ *and* $v \sim_\beta v'$ *implies* $v :: os \sim_{k::st,k::st',\beta} v' :: os'$

LEMMA E.1.3. $os \sim_{st,st',\beta} os'$ *then* $\mathsf{lift}_k\, os \sim_{st,st',\beta} \mathsf{lift}_k\, os'$.

LEMMA E.1.4. $v :: os \sim_{k::st,k'::st',\beta} v' :: os'$
implies $os \sim_{st,st',\beta} os'$ *and* $v \sim_\beta v'$ *and one of the following cases:*

— *either* $k = k'$, $k \leq k_{\text{obs}}$ *and* $v \sim_\beta v'$,
— *or* $k \not\leq k_{\text{obs}}$ *and* $k' \not\leq k_{\text{obs}}$.

LEMMA E.1.5. *If* $\mathsf{length}(st_1) = \mathsf{length}(st_2)$ *and*

$$os_1 :: l_1 :: os'_1 \sim_{st_1::k_1::st'_1,st_2::k_2::st'_2,\beta} os_2 :: l_2 :: os'_2$$

If $k_1 \leq \vec{k'_v}[0]$ *and* $k_2 \leq \vec{k'_v}[0]$, *if*

$$\forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \vec{k'_v}[i+1]$$

and

$$\forall i \in [0, \mathsf{length}(st_2) - 1],\ st_1[i] \leq \vec{k'_v}[i+1]$$

then

$$\{this \mapsto l_1, \vec{x} \mapsto os_1\} \sim_{\vec{k'_v},\beta} \{this \mapsto l_2, \vec{x} \mapsto os_2\}$$

E.2 Heap

LEMMA E.2.1. *Let* $k \in \mathcal{S}$ *a security level, for all heap* $h \in \mathsf{Heap}$ *and object* $o \in \mathcal{O}$,

$$h \preceq_k h \oplus \{\text{fresh}(h) \mapsto o\}$$

LEMMA E.2.2. $h \sim_\beta h_0$ *and* $l = \text{fresh}(h)$ *implies* $h \oplus \{l \mapsto o\} \sim_\beta h_0$

LEMMA E.2.3. $h \sim_\beta h_0$ *and* $\mathsf{ft}(f) \not\leq k_{\text{obs}}$ *implies* $h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\} \sim_\beta h_0$

LEMMA E.2.4. $h \sim_\beta h_0$, $k \not\leq k_{\text{obs}}$ *and* $h \preceq_k h'$ *implies* $h' \sim_\beta h_0$

LEMMA E.2.5. *If* $h_1 \sim_\beta h_2$, $l_1 = \text{fresh}(h_1)$ *and* $l_2 = \text{fresh}(h_2)$, *then* $\beta' = \beta \oplus \{l_1 \mapsto l_2\}$ *verifies:*

— $\beta \subseteq \beta'$,
— $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\} \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}$

PROOF. By hypothesis, β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$, $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$, $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$ and for every $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_\beta h_2(\beta(l))$.

We remark first that $l_1 \notin \mathsf{dom}(\beta)$, since $l_1 \notin \mathsf{dom}(h_1)$ and $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$. Similarly, we have $l_2 \notin \mathsf{rng}(\beta)$, since $l_2 \notin \mathsf{dom}(h_2)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$.

From $l_1 \notin \mathsf{dom}(\beta)$, we deduce that $\beta \subseteq \beta'$.

Since β is a bijection between $\mathsf{dom}(\beta)$ and $\mathsf{rng}(\beta)$, since $l_1 \notin \mathsf{dom}(\beta)$ and $l_2 \notin \mathsf{rng}(\beta)$, we deduce β' is a bijection between $\mathsf{dom}(\beta')$ and $\mathsf{rng}(\beta')$.

We finally prove that for all $l \in \mathsf{dom}(\beta')$, $h_1 \oplus \{l_1 \mapsto \mathsf{default}_C\}(l) \sim_{\beta'} h_2 \oplus \{l_2 \mapsto \mathsf{default}_C\}(\beta'(l))$. If $l \neq l$ the result trivially holds by hypothesis. If $l = l$ we just have to prove that

$\mathsf{default}_C \sim_{\beta'} \mathsf{default}_C$. This is true since for all fields $f \in \mathsf{dom}(\mathsf{default}_C)$, $\mathsf{default}_C.f$ is equal to 0 or *null* (elements on which $\sim_{\beta'}$ is reflexive).

□

LEMMA E.2.6. *If $h_1 \sim_\beta h_2$, if $l_1 = \mathrm{fresh}(h_1)$ and $l_2 = \mathrm{fresh}(h_2)$ then the following properties hold*

— $h_1 \oplus\{l_1 \mapsto \mathsf{default}_C\} \sim_\beta h_2$
— $h_1 \sim_\beta h_2 \oplus\{l_2 \mapsto \mathsf{default}_C\}$
— $h_1 \oplus\{l_1 \mapsto \mathsf{default}_C\} \sim_\beta h_2 \oplus\{l_2 \mapsto \mathsf{default}_C\}$

E.3 Extension of β

LEMMA E.3.1. *If $v_1 \sim_\beta v_2$ and $\beta \subseteq \beta'$ then $v_1 \sim_{\beta'} v_2$.*

LEMMA E.3.2. *If $\rho_1 \sim_\beta \rho_2$ and $\beta \subseteq \beta'$ then $\rho_1 \sim_{\beta'} \rho_2$.*

LEMMA E.3.3. *If $os_1 \sim_{st_1,st_2,\beta} os_2$ and $\beta \subseteq \beta'$ then $os_1 \sim_{st_1,st_2,\beta'} os_2$.*

E.4 Indistinguishability monotony

LEMMA E.4.1 HIGH STACK TYPE SUB-TYPING. *Let $st_1, st_2 \in \mathcal{S}^\star$ be two stack types such that $st_1 \sqsubseteq st_2$, if $os \in \mathcal{V}^\star$ is an operand stack such that $\mathsf{high}(os, st_1)$ then $\mathsf{high}(os, st_2)$.*

LEMMA E.4.2 INDISTINGUISHABILITY DOUBLE MONOTONY. *Let $st, st_1, st_2 \in \mathcal{S}^\star$ be three stack types such that $st_1 \sqsubseteq st$ and $st_2 \sqsubseteq st$, if $os_1, os_2 \in \mathcal{V}^\star$ are two operand stacks such that $os_1 \sim_{st_1,st_2} os_2$ then $os_1 \sim_{st,st} os_2$.*

LEMMA E.4.3 INDISTINGUISHABILITY SINGLE MONOTONY. *Let $st_0, st_1, st_2 \in \mathcal{S}^\star$ be three stack types such that $st_1 \sqsubseteq st_2$, if $os, os_0 \in \mathcal{V}^\star$ are two operand stacks such that $os \sim_{st_1,st_0} os_0$ and $\mathsf{high}(os, st_1)$ then $os \sim_{st_2,st_0} os_0$.*