HAL
open science

A certified lightweight non-interference Java bytecode verifier

Gilles Barthe, David Pichardie, Tamara Rezk

To cite this version:

Gilles Barthe, David Pichardie, Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. Mathematical Structures in Computer Science, 2013, 23 (5), pp.1032-1081. 10.1017/S0960129512000850 . hal-00915189

HAL Id: hal-00915189

https://inria.hal.science/hal-00915189v1

Submitted on 7 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Under consideration for publication in Math. Struct. in Comp. Science

A Certified Lightweight Non-Interference Java Bytecode Verifier

GILLES BARTHE[1], DAVID PICHARDIE[2] and TAMARA REZK[3]

[1] *IMDEA Software Institute, Spain.*

[2] *INRIA Rennes - Bretagne Atlantique, France.*

[3] *INRIA Sophia Antipolis - Méditerranée, France.*

Received 27 September 2011

Non-interference guarantees the absence of illicit information flow throughout program execution. It 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. We define an information flow type system for a sequential JVM-like language that includes all these programming features, and we prove, in the Coq proof assistant, that it guarantees non-interference. 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 certified information flow type system for such an expressive fragment of the JVM.

Contents

1 Introduction 3

2 Related work 7

2.1 Prior work 7

2.2 Companion works 8

2.3 Other related work 8

3 The $\mathrm{JVM}_{\mathcal{I}}$ submachine 11

3.1 Programs, memory model, and operational semantics 11

3.2 Non-Interference 13

3.3 Informal presentation of the type system 15

3.4 Typing rules 19

3.5 Type system soundness 23

3.6 Computing and verifying the CDR structure 27

3.7 Verifying typability 28

4 $\mathrm{JVM}_{\mathcal{O}}$: The object-oriented extension of $\mathrm{JVM}_{\mathcal{I}}$ 29

4.1 Programs, memory model, and operational semantics 29

4.2 Non-Interference 32

4.3 Typing rules 35

4.4 Type system soundness 38

5 $\mathrm{JVM}_{\mathcal{C}}$: The Method Extension of $\mathrm{JVM}_{\mathcal{O}}$ 39

5.1 Programs, memory model, and operational semantics 39

5.2 Non-Interference 40

5.3 Typing rules 43

5.4 Type system soundness 46

6 $\mathrm{JVM}_{\mathcal{E}}$: The exception-handling extension of $\mathsf{JVM}_{\mathcal{C}}$ 46

6.1 Programs, memory model and operational semantics 47

6.2 Non-Interference 51

6.3 Typing rules 53

6.4 A typable example 55

6.5 Type system soundness 56
7 Machine-checked proof 57
7.1 Motivation and overview 58
7.2 Formal semantics 60
7.3 Formalization of the security condition 63
7.3.1 Finite maps 63
7.3.2 Indistinguishability relations 64
7.4 Soundness proof methodology 64
7.5 Executable checkers 65
8 Conclusion 66
References 68

1. Introduction

The Java security architecture combines static and dynamic mechanisms to ensure that applications are not harmful to other applications or to the runtime environment. In particular, a bytecode verifier statically guarantees that a program is safe, i.e. does not perform arithmetic on references, overflows the stack, or jumps to protected memory locations. A stack inspection mechanism dynamically performs access control verifications. However, the Java security architecture lacks 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.

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 prop-

erties 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 type-based enforcement mechanisms have been developed for advanced programming features such as exceptions, objects (Banerjee and Naumann, 2005), interactions (O'Neill et al., 2006), concurrency (Volpano and Smith, 1998) and distribution (Mantel and Sabelfeld, 2003). In parallel to these fundamental studies, there have been efforts to design and implement information flow type systems for fully-fledged programming languages such as Java (Myers, 1999) and Caml (Pottier and Simonet, 2003). One leading effort towards the development of information-flow aware programming language is Jif (Myers, 1999), which builds upon the decentralized label model and offers a flexible and expressive framework to define information flow policies for Java programs.

The central contribution of this paper is the definition and the proof of soundness of an information flow type system for a significant subset of (sequential) Java bytecode programs including objects, arrays, methods, and exceptions—but excluding e.g. initialization, multi-threading, and garbage collection. The type system builds upon previous works by the authors (Barthe et al., 2004; Barthe and Rezk, 2005) which 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 an object-oriented language that resembles the $\mathrm{JVM}_{\mathcal{O}}$ fragment of this paper—extended with a simplified treatment of exceptions. This paper adopts many of the ideas and techniques of (Barthe and Rezk, 2005), but also improves substantially over it in terms of language coverage, precision of the analysis, and expressiveness of the policy:

— *language coverage:* we provide a treatment of exceptions that is close to Java, and include methods and arrays;

— *precision of the analysis:* we rely on a refined notion of control dependence region that provides a fine-grained treatment of exceptions and make the analysis able to communicate with preliminary analyses to reduce the control flow graph of applications,

— *policy expressiveness:* we adopt arbitrary lattices of security levels instead of two-element lattices.

While these issues have been addressed previously in isolation, their combination yields significant complexity in soundness proofs, making it necessary to machine-check proofs rather than using pen-and-paper arguments. A second contribution of our work is a formalization in the Coq proof assistant of a lightweight information flow verifier that checks whether a program is typable according to our type system. The verifier is compatible with the Java architecture and operates in the fashion of lightweight bytecode verification, i.e. it takes a JVM program with security annotations (and some additional information on the control dependence regions of programs), and checks that the program respects the security policy purported by the annotations.

Contents of the paper We begin with an overview of related work in Section 2. Then, we analyze in turn increasingly complex fragments of the JVM:

— the machine $\mathrm{JVM}_{\mathcal{I}}$, studied in Section 3, 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 $\mathrm{JVM}_{\mathcal{I}}$ are adapted from our earlier work (Barthe et al., 2004);

— the machine $\mathrm{JVM}_{\mathcal{O}}$, studied in Section 4, is an object-oriented extension of $\mathrm{JVM}_{\mathcal{I}}$ which includes features such as dynamic object creation, instance field accesses and

updates, arrays and is expressive enough for compiling intra-procedural statements from (Banerjee and Naumann, 2005). In this section, we define and discuss heap indistinguishability. The main difficulty is to propose a sufficiently fine type system which allows public arrays to handle secret information.

— $\mathrm{JVM}_{\mathcal{C}}$, studied in Section 5, is a procedural extension of $\mathrm{JVM}_{\mathcal{O}}$ 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;

— $\mathrm{JVM}_{\mathcal{E}}$, studied in Section 6, extends $\mathrm{JVM}_{\mathcal{C}}$ 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.

For each fragment, we shall define the syntax and semantics of programs; formulate the security policy and the typing rules; and finally prove soundness of the type system. Section 7 provides additional details on the formal proof developed in Coq.

This paper supersedes (Barthe et al., 2007). The main differences are the incremental presentation of different language fragments, the longer account of the machine-checked formalization, and the addition of several examples.

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. Related work

2.1. *Prior work*

In order to realize our goal of defining a sound information flow type system for (sequential) Java bytecode, we draw influences from several earlier works that address its features in isolation. For example, our approach to deal with unstructured code is inspired from Kobayashi and Shirane (Kobayashi and Shirane, 2002), who defined the first information flow type system for a low level language for a subset of the JVM similar to the $\mathrm{JVM}_{\mathcal{I}}$ machine defined in Section 3. We adopt from their type system the use of: i) control dependence regions; ii) security environments. Similar concepts arise in the work of Agat (Agat, 2000), who studied the possibility of eliminating timing leaks through program transformations. For example, Agat uses control dependence regions (which he calls *contexts*) to detect the instructions whose timing behavior may leak information.

Many ideas of the type system originate from Jif (Myers, 1999), an information-flow aware extension of Java that builds on the decentralized label model. Our type system adopts from this work: i) the form of method signatures, ii) the use of pre-analyses to reduce the control flow graph, iii) the ability of public arrays to handle secret information. Jif supports a rich set of mechanisms for specifying and enforcing expressive and flexible security policies. The richness of the Jif type system also makes it difficult to prove soundness—and there is no fully formal description of the type system.

Banerjee and Naumann (Banerjee and Naumann, 2005) develop a provably sound information flow type system for a fragment of Java with objects and methods. Our type system adopts from (Banerjee and Naumann, 2005): i) the focus on a simpler type system that does not support declassification policies nor label polymorphism, ii) the definition

of heap equivalence, iii) the typing rules for method invocations. Their type system is simpler than ours since they omit language features such as exceptions and arrays.

2.2. *Companion works*

A companion work (Barthe et al., 2006) establishes a formal correspondence between the source type systems of (Banerjee and Naumann, 2005) and ours, in the form of a type-preservation result, showing that the compiler maps typable Java programs to typable bytecode programs. As a result, our certified verifier can be used to deploy in a Foundational Proof Carrying Code architecture any program that type-checks in an extension of the type system of (Banerjee and Naumann, 2005) to exceptions. Section 8 also discusses briefly works towards extending our type system to multi-threading (Barthe et al., 2010; Barthe and Rivas, 2011) and declassification (Barthe et al., 2008).

2.3. *Other related work*

This section provides a short summary of other related work. A more detailed account appears in the third author's thesis (Rezk, 2006).

Java A hypothesis of the works of (Myers, 1999; Banerjee and Naumann, 2005) and of this paper is a semantics in which references are opaque, i.e. the only observations that can be made about a reference are those about the object to which it points. Hedin and Sands (Hedin and Sands, 2006) observed that implementations of the Java Virtual Machine commonly violate this assumption, and that allow references to be cast to an integer; moreover, they exhibited a typable 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 one 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.

Information flow has close connections with slicing and dependence analyses (Abadi et al., 1999), and it is possible to adapt methods from this field to analyze the security of programs. For example, Hammer, Krinke and Snelting (Hammer et al., 2006) have developed an automatic and flow-sensitive information flow analysis for Java 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.

JVM Bieber *et al.* (Bieber et al., 2002) provide an early study of information flow in 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 supported 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.

There are alternative approaches to verify information flow properties of bytecode programs. For example, 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.

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 Bonelli, Compagnoni, and Medel (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.

Flow-sensitive type systems and relational logics The type system presented in this paper is flow-insensitive, in the sense that the security level of a variable is fixed throughout the program execution. While it simplifies the description of the type system and its soundness proof, flow-insensitivity restricts the generality of the type system, and leads to secure programs being rejected. In contrast, flow-sensitive verification methods allow the security level of variables to evolve throughout execution, and makes it possible to type more programs. Examples of flow-sensitive methods include the logic of Banerjee

et al (Amtoft et al., 2006), that allows to verify non-interference for an object-oriented language, using independence assertions inspired from separation logic, the type system of Hunt and Sands (Hunt and Sands, 2006), and the aforementioned analysis of Hammer *et al* (Hammer et al., 2006).

While flow sensitivity adds useful expressiveness for a source language, its role is less prominent in the case of type systems, like ours, that aim at verifying bytecode (or executable code), as there exist SSA-like transformations that transform programs that are accepted by flow-sensitive type systems into programs that are accepted by a flow-insensitive one (Hunt and Sands, 2006).

3. The $\mathrm{JVM}_{\mathcal{I}}$ 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.

3.1. *Programs, memory model, and operational semantics*

Programs A $\mathrm{JVM}_{\mathcal{I}}$ program P is given by a list of instructions, taken from the instruction set of Figure 1. We let the set $\mathcal{X}$ be the set of local variables and we let $\mathcal{V}$ be the set of values, i.e. $\mathcal{V} = \mathbb{Z}$. Each program has a set of program points $\mathcal{PP}$, which is defined as $\{1 \ldots n\}$, where n is the length of the list of instructions of P.

States The set $\mathsf{State}_{\mathcal{I}}$ of $\mathrm{JVM}_{\mathcal{I}}$ 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, and $os \in \mathcal{V}^{\star}$ is an operand stack.

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. 1. Instruction set for $\mathrm{JVM}_{\mathcal{I}}$

Operational semantics The small-step operational semantics of the $\mathrm{JVM}_{\mathcal{I}}$, is given in Figure 2 as a relation $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{I}} \times (\mathsf{State}_{\mathcal{I}} + \mathcal{V})$, and is implicitly parametrised by a program P.

In the figure, $\underline{op}$ denotes the standard interpretation of operation op in the domain of values $\mathcal{V}$. The semantics of each instruction is 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 pushes the result of the binary operation op using these operands. Instruction pop just pops the top of the operand stack. Instruction swap swaps the two top operand stack values. Instruction return ends the execution with the top value of the operand stack. Instruction load x pushes, on top of the operand stack, the value currently found in local variable x. 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 whether 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.

The transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow^+$.

Successor relation It is often convenient to view programs as graphs. The graph representation of programs is given by specifying its entry point—by convention it is always

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

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

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

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

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

Fig. 2. Operational Semantics for $\mathrm{JVM}_{\mathcal{I}}$

1—its exit points and the successor relation between program points. Intuitively, j is a successor of i if performing a one-step execution from a state whose program point is i may lead to a state whose program point is j. Besides, j is a return point if it corresponds to a return instruction. Formally, 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, $i \mapsto i+1$.

One also defines for each program P its set $\mathcal{PP}_r$ of return points, i.e. of programs points with no successor—or equivalently, program points that are mapped to a return instruction. By abuse of notation, we write $i \mapsto$ if $i \in \mathcal{PP}_r$.

3.2. *Non-Interference*

The security policy is given by a lattice $(\mathcal{S}, \leq)$ of security levels, and the policy of the program. In the $\mathrm{JVM}_{\mathcal{I}}$ fragment, the policy of a program P is given by a statement of the form $\vec{k_v} \longrightarrow k_r$, where $\vec{k_v}$ assigns a security level to the each local variables, and k_r

sets a security level of its output. In the sequel, we often view $\vec{k_v}$ as a partial mapping from variables to security levels. The notion of non-interferent program is also defined relative to a security level k_{obs} corresponding to the attacker; essentially, the attacker can observe return values and variables whose level is less than or equal to k_{obs}.

The policy of the program and the security level of the attacker induce a notion of indistinguishability between local variable maps.

Definition 3.1 (Local variables indistinguishability). For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim_{\vec{k_v}, k_{\text{obs}}} \rho'$ if ρ and ρ' have the same domain and $\rho(x) = \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\vec{k_v}(x) \leq k_{\text{obs}}$.

In the remaining of the paper, we shall sometimes omit the subscripts $\vec{k_v}$ and k_{obs} whenever there is no risk of confusion. Next, we define the notion of non-interferent program: first, we define a weak notion of non-interferent program for a fixed attacker level; then, we say that a program is non-interferent iff it is non-interferent for all attacker levels.

Definition 3.2 (Non-interferent JVM$_{\mathcal{I}}$ program). A program P is *non-interferent* w.r.t. policy $\vec{k_v} \longrightarrow k_r$ and attacker level k_{obs}, if either $k_r \not\leq k_{obs}$ or $v_1 = v_2$ for every ρ_1, ρ_2, v_1, v_2 such that $\langle 1, \rho_1, \epsilon \rangle \rightsquigarrow^+ v_1$ and $\langle 1, \rho_2, \epsilon \rangle \rightsquigarrow^+ v_2$ and $\rho_1 \sim_{\vec{k_v}, k_{\text{obs}}} \rho_2$.

Moreover, a program P is *non-interferent* w.r.t. policy $\vec{k_v} \longrightarrow k_r$ iff for all attacker levels k_{obs}, P is non-interferent w.r.t. $\vec{k_v} \longrightarrow k_r$ and k_{obs}.

Our definition of non-interference is termination-insensitive, i.e. does not take into account non-terminating executions of programs. Stronger definitions, that reject programs whose termination behaviour depend on high inputs, have been considered in the literature, but the type systems enforcing such policies tend to impose strong restrictions on loops.

3.3. *Informal presentation of the type system*

This paragraph is devoted to pointing out issues about enforcing non-interference for unstructured programs, and providing an informal account of the solutions.

Like any other information flow type system, our type system must prevent leakages that occur through assigning secret values to public variables (direct flows), or through branching over expressions that depend on secrets, and performing in the branches operations that affect the visible part of the state (indirect flows). Our type system prevents direct flows through stack types, and indirect flows through a combination of control dependence regions and security environment.

Direct flows In a high level language, direct flows are prevented by the typing rule for assignments, which is usually of the form (Volpano and Smith, 1997)

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

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 not less than and not equal to that of x, and thus that there is no illicit flow to x.

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 to store 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

$$\mathsf{load}\ y_H$$

$$\mathsf{store}\ x_L$$

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

Indirect flows In a high level language with structured control flow, typing judgements are of the form $\vdash c : k$. Informally, if a command c is typable then it is non-interfering, and moreover if $\vdash c : H$ then c does not modify any low variable. In such systems, indirect flows are prevented by the typing rules branching statements, which for if-then-else statements is usually of the form (Volpano and Smith, 1997):

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

and thus ensures that the write effects of c_1 and c_2 are not less or equal to the guard of the branching statement.

To prevent illicit flows in a low-level language, the typing rules for branching instructions cannot simply enforce local constraints, i.e. they cannot talk only about the current program point and its successors. Instead, the typing rules must also enforce global constraints that prevent low assignments and updates to occur under high guards. Therefore, the typing rules rely on a graph representation of the program, and an approximation of the scope of branching statements using control dependence regions.

Control dependence regions Our type system assumes that programs are bundled with additional information about their control dependence regions. This assumption is in line with the intended usage of our type checker as a lightweight bytecode verifier and

streamlines the presentation by allowing us to focus on the information flow analysis itself. The information is given in the form of two functions region and jun. The intuition behind regions and junction points is that region(i) includes all program points executing under the guard at i and that jun(i), if it exists, is the sole exit from the region of i; in particular, whenever jun(i) is defined there should be no return instruction in region(i). Figure 3 provides examples of regions of two compiled programs. Note that in the rightmost picture, which corresponds to an if-then-else statement, the branching point i does not belong to its region, whereas in the leftmost picture, which corresponds to a while-do statement, the branching point i does belong to its region.

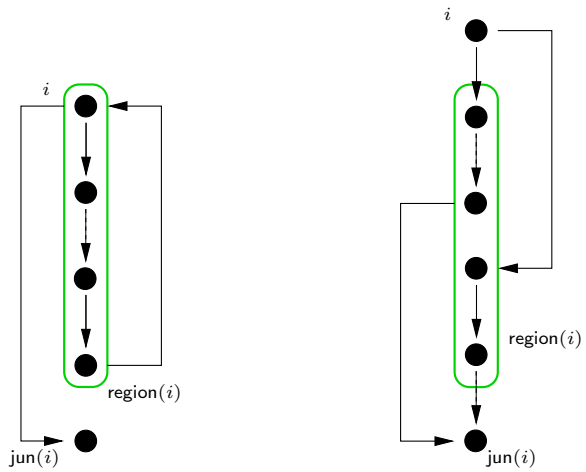


Fig. 3. Example of CDR for a while and an if construct.

The soundness of the type system requires that the functions verify the following properties: any successor of i either belongs to the region of i, or is equal to jun(i) (if defined), and jun(i) is the sole exit to the region of i; in particular if jun(i) is defined there should be no return instruction in region(i).

Definition 3.3 (SAFE CDR structure). A control dependence region (CDR) struc-

ture $(\mathsf{region}, \mathsf{jun})$ given by a total function region and a partial function jun is safe if the following properties hold:

CDR1 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 $k = \mathsf{jun}(i)$;

CDR2 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 $k = \mathsf{jun}(i)$;

CDR3 for all program points i, j, if $j \in \mathsf{region}(i)$ and $j \mapsto$ then $\mathsf{jun}(i)$ is undefined.

Subsection 3.6 provides additional information on computing and checking CDR structures. For the purpose of the soundness of the type system, it is sufficient to know that the program is packaged with a CDR structure that satisfies the above properties.

Security environments The type system is further parametrised by a security environment that attaches a security level to each program point. Informally, the security level of a program point is an upper bound of all the guards under which the program point executes.

The security environment is used in conjunction with the CDR information to prevent implicit flows. This is done in two steps: on the one hand, the typing rule for branching statements enforces that the security environment of a program point is indeed an upper bound of the guard under which it executes; for instance, the rule for ifeq bytecode is of the form:

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

On the other hand, the typing rules for instructions with write effect, e.g. store, 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 `if` $(\mathtt{y_H})$ $\{\mathtt{x_L} = \mathtt{0};\}$ `else` $\{\mathtt{x_L} = \mathtt{1};\}$ is compiled in our low-level language as

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

By requiring that $se(i) \leq \vec{k_v}(x)$, where i is the program point of the store instruction, and by requiring a global constraint on the security environment in the 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.

3.4. *Typing rules*

Our typing rules are of the form:

$$\frac{P[i] = ins \quad constraints}{\vec{k_v} \longrightarrow k_r, \mathsf{region}, se, i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins \quad constraints}{\vec{k_v} \longrightarrow k_r, \mathsf{region}, se, i \vdash st \Rightarrow}$$

where $\vec{k_v} \longrightarrow k_r$ is a policy, $st, st' \in \mathcal{S}^\star$ are stacks of security levels, and *ins* is an instruction found at point i in program P. Our type rules do not record the types of variables: indeed, our type system is flow-insensitive.

$$\frac{P[i] = \mathsf{pop}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{binop}\ op}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{push}\ n}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{swap}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \vec{k_v}(x)}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash st \Rightarrow \left(\vec{k_v}(x) \sqcup se(i)\right) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) \sqcup k \leq k_r}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k :: st \Rightarrow}$$

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

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

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 3.4 (Typable program). A program P is typable w.r.t. a given policy $\vec{k_v} \longrightarrow k_r$, a CDR structure $\mathsf{region} : \mathcal{PP} \to \wp(\mathcal{PP})$, and a security environment $se : \mathcal{PP} \to \mathcal{S}$ if there exists a function $S : \mathcal{PP} \to \mathcal{S}^\star$, called a global typing, 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 $\vec{k_v} \longrightarrow k_r, \mathsf{region}, se, i \vdash S_i \Rightarrow st$ and $st \sqsubseteq S_j$;

2 $i \mapsto$ implies that $\vec{k_v} \longrightarrow k_r, \mathsf{region}, se, i \vdash S_i \Rightarrow$;

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. Two type stacks are in relation only if they have the same size.

It may be helpful to read the definition of typable program from the view of abstract interpretation. Informally, a program P has type S iff S is a post-fixpoint of the system of data flow equations induced by the transfer rules.

Figure 4 presents the typing rules for the $\mathsf{JVM}_{\mathcal{I}}$. We let $\sqcup$ denote the lub of two security levels, and for every $k \in \mathcal{S}$, we let lift_k be the point-wise extension to stack types of $\lambda l.\ k \sqcup l$. All rules are parametrised by a CDR region, a security environment se and a policy $\vec{k_a} \longrightarrow k_r$.

Below we comment on some essential rules:

— The transfer rule for an instruction $\mathsf{push}\ n$ prevents indirect 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` $\mathtt{y_H}$? 0 : 1;, illustrates the need for this constraint:

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

The program is interferent with respect to the policy $(y_H : H) \longrightarrow L$, and hence it should not typable. The typing rule for return instruction rightfully rejects this program because the top of the stack is typed as high when reaching point l_3. Indeed, the instructions $\mathsf{push}\ 0$ and $\mathsf{push}\ 1$ are in the region of the branching instruction $\mathsf{ifeq}\ l_1$ and the security environment se is high at this point.

— 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 prevent 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.

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

In this example, the final value of variable x_L is equal to 0 or 1 and reveals if the value of y_H is 0 or not. So the program is interferent. Our type system rightfully rejects this program. Indeed, the rule for ifeq at point l_1 lifts the operand stack as high and in particular constrains the top of the stack at point l_3 to be a high value; as the rule for store prevents the assignment from high to low, the program is rejected.

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[†] 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 prevents return instructions under

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

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 $\leq k_r$, since it will be observed by the attacker at level k_r. The following example illustrates the need for preventing return instructions in high regions. It corresponds to the source program `if` ($\mathtt{y_H}$) {`return` 0; } `else` {`return` 1; }.

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

This program is interferent w.r.t a policy $\vec{k_v} \longrightarrow L$ because there is a return in a high ifeq and the result will be observed by the attacker. This program is rightfully rejected by the type system: the rule for the ifeq forces the operand stack to be high upon reaching the return instruction, and the return rule prevents the program from returning an observable value in a high security environment.

3.5. *Type system soundness*

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

Theorem 3.5. Let P be a typable $\mathrm{JVM}_{\mathcal{I}}$ program w.r.t. a safe CDR (region, jun) and with a policy $\vec{k_a} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\vec{k_a} \longrightarrow k_r$.

The proof of soundness is based on some assumptions concerning the CDR information, two unwinding lemmas and two lemmas about preserving high contexts.

The unwinding lemmas show that the 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 $\sim$. 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, and to allow proving the step consistent unwinding lemmas—see below—indistinguishability between states must encompass states that have operand stacks of different length.

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.

Definition 3.6 (High operand stack). 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 $st[i] \not\leq k_{\text{obs}}$ for every $1 \leq i \leq n$.

Definition 3.7 (Operand stack indistinguishability). Let $os, os' \in \mathcal{V}^\star$ and $st, st' \in \mathcal{S}^\star$. Then $os : st \sim os' : st'$ is defined inductively as follows:

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

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

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

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.

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

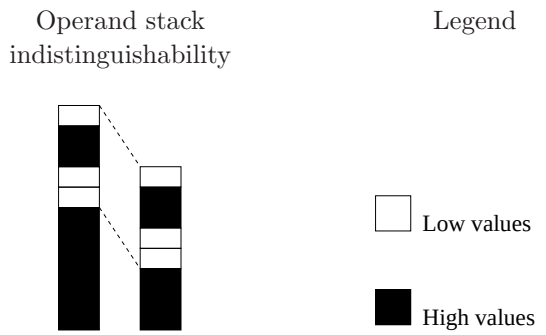


Fig. 5. Operand stack indistinguishability

Definition 3.8 (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 $\langle i, \rho, os\rangle : st \sim \langle i', \rho', os'\rangle : st'$, iff $os : st \sim os' : st'$ and $\rho \sim \rho'$ hold.

Besides, we say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j) \not\leq k_{\text{obs}}$ for all $j \in \mathsf{region}(i)$. A state $\langle i, \rho, os\rangle$ and a stack type st are *high* if $\mathsf{high}(os, st)$ holds.

We now turn to the unwinding lemmas. The lemmas consider a program P that comes equipped with its policy $\vec{k_v} \longrightarrow k_r$, its CDR structure $(\mathsf{region}, \mathsf{jun})$ and security environment se. All are left implicit in the rest of this section. The lemmas state:

— *locally respects:* if $s_1 : st_1 \sim s_2 : st_2$, and $\mathsf{pc}(s_1) = \mathsf{pc}(s_2) = i$, and $s_1 \rightsquigarrow s_1'$, $s_2 \rightsquigarrow s_2'$, $i \vdash st_1 \Rightarrow st_1'$, and $i \vdash st_2 \Rightarrow st_2'$, then $s_1' : st_1' \sim s_2' : st_2'$.

— *step consistent:* if $s_1 : st_1 \sim s_2 : st_2$ and $s_1 \rightsquigarrow s_1'$ and $\mathsf{pc}(s_1) \vdash st_1 \Rightarrow st_1'$, and $se(\mathsf{pc}(s_1)) \not\leq k_{obs}$, and (st_1, s_1) is high, then $s_1' : st_1' \sim s_2 : st_2$.

Both lemmas are proved by a case analysis on the instruction to be executed.

To repeatedly apply unwinding lemmas, we need to have a family of results that deals with preservation of high contexts.

— *high branching:* if $s_1 : st_1 \sim s_2 : st_2$ with $\mathsf{pc}(s_1) = \mathsf{pc}(s_2) = i$ and $\mathsf{pc}(s_1') \neq \mathsf{pc}(s_2')$, if

$s_1 \rightsquigarrow s'_1$, $s_2 \rightsquigarrow s'_2$, $i \vdash st_1 \Rightarrow st'_1$ and $i \vdash st_2 \Rightarrow st'_2$, then (s'_1, st'_1) and (s'_2, st'_2) is high and se is high in region $\mathsf{region}(i)$.

— *high step:* if $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash st \Rightarrow st'$, and $se(\mathsf{pc}(s)) \not\leq k_{obs}$, and (s, st) is high, then (s', st') is high.

These lemmas are proved by case analysis on the instruction being executed, and rely on the CDR properties.

The second family of results deals with monotonicity of indistinguishability.

— *high stack type sub-typing:* if (s, st) is high and $st \sqsubseteq st'$ then (s, st') is high.

— *indistinguishability double monotonicity:* if $s_1 : st_1 \sim s_2 : st_2$, $st_1 \sqsubseteq st$ and $st_2 \sqsubseteq st$ then $s_1 : st \sim s_2 : st$.

— *indistinguishability single monotonicity:* if $s_1 : st_1 \sim s_2 : st_2$, $st_1 \sqsubseteq st'_1$ and (s_1, st_1) is high then $s_1 : st'_1 \sim s_2 : st_2$.

The proof makes use of the unwinding lemmas, the high context lemmas, the monotonicity lemmas and the CDR properties, and proceeds by induction on execution traces. In the induction step[‡] 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 $s_0 : S_{\mathsf{pc}(s_0)} \sim s'_0 : S_{\mathsf{pc}(s'_0)}$ and we want to establish that states s_n and s'_m are indistinguishable:

$$s_n : S_{\mathsf{pc}(s_n)} \sim s'_m : S_{\mathsf{pc}(s'_m)}$$

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

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

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

— If $\mathsf{pc}(s_1) = \mathsf{pc}(s'_1)$ we can apply the *indistinguishability double monotonicity* lemma to establish that $s_1 : S_{\mathsf{pc}(s_1)} \sim s'_1 : S_{\mathsf{pc}(s'_1)}$ 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 (s_1, st) and (s'_1, st') are high. Thanks to the *high stack type sub-typing* lemma it implies that both $(s_1, S_{\mathsf{pc}(s_1)})$ and $(s'_1, S_{\mathsf{pc}(s'_1)})$ are high. By CDR1 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 trace $s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$ we easily show that either there exists k, $1 \leq k \leq n$ such that $k = \mathsf{jun}(i_0)$ and $s_k : S_{\mathsf{pc}(s_k)} \sim s'_0 : S_{i_0}$ (the junction point is reached) or $\mathsf{pc}(s_n) \in \mathsf{region}(i_0)$ and $(s_n, S_{\mathsf{pc}(s_n)})$ is high (the trace stays in the region of i_0). This is proved thanks to CDR2, *high step* lemma and *indistinguishability single monotonicity* 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 CDR3. A similar property holds for trace $s'_1 \rightsquigarrow \cdots \rightsquigarrow s'_m$ and we can group the 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 $s_k : S_{\mathsf{pc}(s_k)} \sim s'_0 : S_{i_0}$ and $s_0 : S_{i_0} \sim s'_{k'} : S_{\mathsf{pc}(s'_{k'})}$. Since $s_0 : S_{i_0} \sim s'_0 : S_{i_0}$ we have by transitivity and symmetry of $\sim$, $s_k : S_{\mathsf{pc}(s_k)} \sim s'_{k'} : S_{\mathsf{pc}(s'_{k'})}$ 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 $(s_n, S_{\mathsf{pc}(s_n)})$ and $(s'_m, S_{\mathsf{pc}(s'_m)})$ are high.

3.6. *Computing and verifying the CDR structure*

The CDR information comes bundled with the code and hence is untrusted. Therefore, it must be verified by a CDR checker; specifically, the CDR checker will ensure that the CDR information complies with the CDR properties—we have shown that these properties are sufficient to guarantee soundness of the type system. The CDR properties can be checked naively in cubic time, and under specific hypotheses, one can design more effective verification methods.

A related issue is how to compute control dependence regions that satisfy the CDR properties. In fact, CDRs are tightly connected to post-dominators, and it is reasonably easy to prove that the CDR properties hold whenever the CDR information is computed using post-dominators. Recall that a program point j post-dominates another program point i, if $i \mapsto^+ j$, i.e. j is reachable from i in a non-zero number of steps, 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 equal to or 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 $i \mapsto^+ j$ and $\mathsf{jun}(i)$ post-dominates j—in particular, $\mathsf{jun}(i)$ is defined; if not, $j \in \mathsf{region}(i)$ iff $i \mapsto^+ j$. Note that, under this construction, $i \in \mathsf{region}(i)$ entails $i \mapsto^+ i$.

3.7. *Verifying typability*

Typability of a program against a policy can be verified via a dataflow analysis (Hankin et al., 2005). More specifically, the checker takes as input a program with its CDR information and with its security annotations, in this case a security environment, a security level for each variable and the result, and a type (i.e. a map from program points to stack type), and checks that the program is typable using a lightweight variant of Kildall's algorithm, see e.g. (Rose, 2003). Since programs are annotated, the checker does not perform a fixpoint computation, but verifies the program in one pass.

We briefly comment on some practical issues regarding using our lightweight information flow checker in a Proof Carrying Code scenario. First, our checker requires programs to carry a significant amount of annotation; however, its role is merely to check that the program is secure, whereas the task of automatically inferring annotations and alleviating

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
	|	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

where $C \in \mathcal{C}$, $f \in \mathcal{F}$ and $t \in \mathcal{T}_J$.

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

the programmer's burden is typically done at source level. Second, the security environment, program type, and control dependence regions can be generated automatically by a certifying compiler; hence, the process of generating an annotated JVM program that is processable by our checker from a typable Java program can be automated. Finally, the information flow checker is "reasonably efficient", in the sense that the CDR information can be computed efficiently, the data flow analysis is performed in a single pass, and the constraints generated by the transfer rules only involve inequalities on security levels.

4. $\text{JVM}_{\mathcal{O}}$: The object-oriented extension of $\text{JVM}_{\mathcal{I}}$

The object-oriented extension of $\text{JVM}_{\mathcal{I}}$, namely $\text{JVM}_{\mathcal{O}}$, includes arrays, instance fields, creation of new instances, and null pointers. On the other hand, it does not feature methods, which are only added in Section 5.

4.1. *Programs, memory model, and operational semantics*

Programs $\text{JVM}_{\mathcal{O}}$ programs are extended $\text{JVM}_{\mathcal{I}}$ programs, equipped with a set $\mathcal{C}$ of class names, a set $\mathcal{F}$ of identifiers representing field names and a set $\mathcal{T}_J$ of Java types (a precise description of these types is not necessary here).

Programs use an extended set of instructions, given in Figure 6.

States The set of $\mathrm{JVM}_{\mathcal{O}}$ 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. By distinguishing between locations and integer values, we can enforce in the semantics of programs that they do not perform pointer arithmetic in $\mathrm{JVM}_{\mathcal{O}}$.

A $\mathrm{JVM}_{\mathcal{O}}$ state is now of the form $\langle i, \rho, os, h\rangle$, where i, ρ, and os are defined as in $\mathrm{JVM}_{\mathcal{I}}$ and h is a heap, that accommodates dynamically created objects and arrays. Heaps are modelled as partial functions $h : \mathcal{L} \rightharpoonup \mathcal{O} + \mathcal{A}$, where the set $\mathcal{O}$ of objects is model-ed as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$, i.e. each object $o \in \mathcal{O}$ possesses a class (written as $\mathsf{class}(o)$) and a partial function to access field values, and the set $\mathcal{A}$ of arrays is model-ed as $\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}) \times \mathcal{PP}$, i.e. each array $a \in \mathcal{A}$ possesses a length (written as $a.\mathsf{length}$), a partial function to access array values and a creation point. We write $o.f$ for 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 relies on an allocator function $\mathsf{fresh} : \mathsf{Heap} \to \mathcal{L}$ that given a heap returns the location for a fresh object, and on a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$ that returns for each class a default object of that class. Function $\mathsf{default}$ is specified according to the standard Java convention[§]: for all defined field $f \in \mathcal{F}$ of a class $C \in \mathcal{C}$,

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

A similar operator $\mathsf{defaultArray} : \mathbb{N} \times \mathcal{T}_J \to (\mathbb{N} \rightharpoonup \mathcal{V})$ models array initialisation.

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 initialised object. The heap is updated with this new object. Instruction $\mathsf{getfield}\ f$ pops

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

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

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

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

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

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

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

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

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

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

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 putfield f uses the top of the stack to update the object associated with the location under the top of the operand stack. Instruction return now returns the top of the operand stack, and the current heap. Instruction newarray t pops a positive integer n from the operand stack to create a new initialised array and pushes the corresponding fresh location on top of the operand stack. The heap is updated with this new array including its length n and its creation point i. Instruction arraylength pops a location l from the operand stack and pushes the length of the corresponding array. Instruction arrayload pops an index j and a location l from the operand stack. The content of the array in location l at index j is fetched and pushed onto the operand stack. Instruction arraystore stores the top of the stack into the j-th element of an array a, where j and a are determined by the second and third values in the stack respectively.

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

4.2. *Non-Interference*

In order to extend the notion of indistinguishability to heaps we follow (Banerjee and Naumann, 2005): we consider that heaps with different locations for "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 low 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, value indistinguishability is defined relative to the bijection β.

For array objects, we extend security levels, with array levels of the form $k[k_c]$. These levels represent the security level of an array, distinguishing the level k_c of the content of the array (which could be itself an array) and the level k of the length of the array and of the reference itself. Hence an array of type $L[H]$ can be only updated in a high context (its content is high) but allocated in any context (its length and the value of its reference are low). We denote by $\mathcal{S}^{\text{ext}}$ the extension of security levels $\mathcal{S}$ with array security levels. The partial order on $\mathcal{S}$ is extended to $\leq^{\text{ext}}$ with the following inductive definition

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

An extended level $k[k_c]$ is considered to be low (written as $k[k_c] \leq k_{\text{obs}}$) if $k \leq k_{\text{obs}}$. More generally, every time we compare an element $k[k_c] \in \mathcal{S}^{\text{ext}}$ with an element $k_0 \in \mathcal{S}$, we just compare k and k_0 w.r.t. the partial order on $\mathcal{S}$ only. Apart from k_{obs} and the elements of the security environments, every previous types (security types of variable and stack

types) are now elements of $\mathcal{S}^{\text{ext}}$. Indistinguishability for $\text{JVM}_{\mathcal{O}}$ is defined relative to a global mapping $\mathsf{ft} : \mathcal{F} \to \mathcal{S}^{\text{ext}}$ that maps fields to security levels (ft will be left implicit in the rest of the paper).

Definition 4.1 (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:

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

Both definitions of operand stack indistinguishability and local variables indistinguishability are now parametrised by β since values on the operand stack and in variables can also be locations.

Indistinguishability for array objects is defined relative to a global mapping $\mathsf{at} : \mathcal{PP} \to \mathcal{S}^{\text{ext}}$ that maps creation point of arrays to security levels for their contents. The mapping at will be left implicit in the rest of the paper. We will abusively denote $\mathsf{at}(a)$ as the level that is associated 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 4.2 (Array indistinguishability). Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable with respect to an attacker level k_{obs} and a function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ if and only if $a_1.\mathsf{length} = a_2.\mathsf{length}$, $\mathsf{at}(a_1) = \mathsf{at}(a_2)$ and moreover, if $\mathsf{at}(a_1) \leq k_{\text{obs}}$ then for all index i such that $0 \leq i < a_1.\mathsf{length}$, $a_1[i] \sim_\beta a_2[i]$.

The definition of object indistinguishability says that two objects are indistinguishable if they have the same class and the values held in their low fields are indistinguishable.

Definition 4.3 (Object indistinguishability). Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to an attacker level k_{obs} and a function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ 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 4.4 (Heap indistinguishability). Two heaps h_1 and h_2 are indistinguishable with respect to an attacker level k_{obs} and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_{k_{\text{obs}},\beta} h_2$, 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 $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_{k_{\text{obs}},\beta} h_2(\beta(l))$ and $h_1(l)$ and $h_2(\beta(l))$ are either two objects or two arrays;

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

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

Definition 4.5 (Non-interferent JVM$_\mathcal{O}$ program). A program P is *non-interferent* w.r.t. its policy $\vec{k_v} \longrightarrow k_r$, if for every attacker level k_{obs} and 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 $\langle 1, \rho_1, \epsilon, h_1\rangle \rightsquigarrow^+ v_1, h_1'$, $\langle 1, \rho_2, \epsilon, h_2\rangle \rightsquigarrow^+ v_2, h_2'$ and $h_1 \sim_{k_{\text{obs}},\beta} h_2$, $\rho_1 \sim_{\vec{k_v},k_{\text{obs}},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1 \sim_{k_{\text{obs}},\beta'} h_2$ and moreover, if $k_r \leq k_{\text{obs}}$ then $v_1 \sim_{\beta'} v_2$.

Here $\beta \subseteq \beta'$ means that $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(\beta')$ and for all locations $l \in \mathsf{dom}(\beta)$, $\beta(l) = \beta'(l)$.

$$\frac{P[i] = \mathsf{new}\ C}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad (se(i) \sqcup k_2) \sqcup^{\mathrm{ext}} k_1 \leq \mathsf{ft}(f) \qquad k_1 \in \mathcal{S}^{\mathrm{ext}} \qquad k_2 \in \mathcal{S}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k_1 :: k_2 :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{getfield}\ f \qquad k \in \mathcal{S}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k :: st \Rightarrow \big((k \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathsf{ft}(f)\big) :: st}$$

$$\frac{P[i] = \mathsf{newarray}\ t \qquad k \in \mathcal{S}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k :: st \Rightarrow k[\mathsf{at}(i)] :: st}$$

$$\frac{P[i] = \mathsf{arraylength} \qquad k \in \mathcal{S} \qquad k_c \in \mathcal{S}^{\mathrm{ext}}}{\vec{k_v} \to k_r, \mathsf{region}, se, i \vdash k[k_c] :: st \Rightarrow k :: st}$$

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

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

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

The definition of non-interference allows for β to be extended, in order to handle objects that are dynamically created during the execution.

4.3. *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. The typing rules we propose for arrays follow Askarov and Sabelfeld (Askarov and Sabelfeld, 2005) who argue that public arrays must be allowed to handle secret information in order to achieve any realistic case study like the mental poker they have programmed in Jif (Myers, 1999).

— The transfer rule for new adds to the stack type the security level of the current program point, which imposes a constraint on the 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 field of the object) 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 may be aliases to low objects.

To illustrate this last point, consider the source program

```
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. The bytecode for this program is:

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

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 a 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 aliasing (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 equal or greater than $\mathsf{ft}(f)$ and the level k of the location (to prevent explicit flows) and equal or greater than $se(i)$ for implicit flows.

— The transfer rule for newarray creates a new security level for the new created array, combining the length level k 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 $(k_1 \sqcup k_2) \sqcup^{\text{ext}} k_c$. The join operation $\sqcup^{\text{ext}} \in \mathcal{S} \times \mathcal{S}^{\text{ext}} \to \mathcal{S}^{\text{ext}}$ is defined by $k' \sqcup^{\text{ext}} k = k' \sqcup k$ when $k, k' \in \mathcal{S}$ and $k' \sqcup^{\text{ext}} k[k_c] = (k' \sqcup k)[k_c]$ when $k, k' \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\text{ext}}$. Here k_1 allows to prevent implicit flows through a high index and k_2 through alias.

The following example illustrates this first point. It corresponds to a source program like

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

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.

load $a_{L[L]}$

load i_H

arrayload

store x_L

In this program, if the low array $\mathtt{a_{L[L]}}$ contains distinct elements at different positions, an attacker could learn the value of $\mathtt{i_H}$ by looking at the result of $\mathtt{a_{L[L]}}[\mathtt{i_H}]$. This program is rejected by our type system because $\mathtt{a_{L[L]}}[\mathtt{i_H}]$ 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 $\mathtt{a}_{\mathtt{H[L]}}[\mathtt{i}_{\mathtt{L}}]$ where $\mathtt{a}_{\mathtt{H[L]}}$ may be either aliased to a low array $\mathtt{a}^{0}_{\mathtt{L[L]}}$ containing only the 0 integer or aliased to a low array $\mathtt{a}^{1}_{\mathtt{L[L]}}$ containing only the integer 1. Hence observing the value of $\mathtt{a}_{\mathtt{H[L]}}[\mathtt{i}_{\mathtt{L}}]$ would allow an attacker to know which of this array is aliased to $\mathtt{a}_{\mathtt{H[L]}}$.

— The transfer rule for arraystore uses the partial order $\leq^{\text{ext}}$ previously defined. It 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 $\mathtt{a}_{\mathtt{L[L]}}[\mathtt{i}_{\mathtt{H}}] = \mathtt{1}$ in a low array $\mathtt{a}_{\mathtt{L[L]}}$ only containing the integer 0 would reveal the value of $\mathtt{i}_{\mathtt{H}}$.

Finally, the constraint between k_3 and k_c prevents modifying low array contents if its reference is high. 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 $\mathtt{a}_{\mathtt{H[L]}}$.

4.4. *Type system soundness*

As for the $\text{JVM}_{\mathcal{I}}$, the type system is sound, in the sense that if a program is typable then it is non-interferent.

Theorem 4.6. Let P be a typable $\text{JVM}_{\mathcal{O}}$ program w.r.t. the safe CDR $(\mathsf{region}, \mathsf{jun})$ and with a signature $\vec{k_v} \longrightarrow k_r$. Then P is non-interferent with respect to the policy associated with $\vec{k_v} \longrightarrow k_r$.

5. $\mathrm{JVM}_{\mathcal{C}}$: 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.

5.1. *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 class inheritance relation. 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 $\mathsf{invokevirtual}\ m_{\mathrm{ID}}$ 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 analyses to rely on an equivalent, so-called mix-step semantics, where method invocation is performed in one big step transition. Thus, a $\mathrm{JVM}_{\mathcal{C}}$ state is defined as in $\mathrm{JVM}_{\mathcal{O}}$.

Operational semantics The mix-step operational semantics for method calls fully evaluates those calls from an initial state to a return value and uses it to continue the current computation. The semantic rules are given in Figure 9. As it can be seen in the first rule, the semantics of instructions is like in $\mathrm{JVM}_{\mathcal{O}}$, except for the new instruction $\mathsf{invokevirtual}$, whose semantics is given by the second rule. The location l is used to resolve the virtual call. Thanks to the class of l and the identifier m_{ID}, a method m' is found in the class hierarchy (through the lookup operator). The transitive closure of $\rightsquigarrow_m$ is then used to

$$\frac{P_m[i] \neq \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \langle i, \rho, os, h\rangle \rightsquigarrow_{\mathrm{JVM}_{\mathcal{O}}} \langle i', \rho', os', h'\rangle}{\langle i, \rho, os, h\rangle \rightsquigarrow_m \langle i', \rho', os', h'\rangle}$$

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

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

obtain the result of the execution of m'. Execution of m' is initialised with location l for the reserved variable *this* and the elements of the operand stack os_1 for the other variables¶.

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.

5.2. *Non-Interference*

Non-interference for a $\mathsf{JVM}_{\mathcal{C}}$ program is given by local policies defined by security signatures for every method and the same global policy mappings, ft and at, introduced for $\mathrm{JVM}_{\mathcal{O}}$.

Method signatures are standard (Myers, 1999; Banerjee and Naumann, 2005) and are of the form $\vec{k_v} \xrightarrow{k_h} k_r$, where $\vec{k_v}$ provides the flow insensitive security level of all method variables (being a parameter or not), 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 the execution of the method.

¶ We assume that all other variables used for local computation in the method are initialised by a default value according to their type.

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 5.1 (Side effect preorder). Two heaps $h_1, h_2 \in \mathsf{Heap}$ are *side effect preordered* with respect to a security level $k \in \mathcal{S}$ (written as $h_1 \preceq_k h_2$) if and only if $\mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2)$ 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 permits to define the notion of *side-effect-safe method*.

Definition 5.2. A method m is *side-effect-safe* with respect to a security level k_h if for all local variables in $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$, all heaps $h, h' \in \mathsf{Heap}$ and value $v \in \mathcal{V}$, $\langle 1, \rho, \epsilon, h\rangle \rightsquigarrow_m^+ v, h'$ imply $h \preceq_{k_h} h'$.

The notion of non-interferent method can be stated using the same indistinguishability relation as in $\mathrm{JVM}_{\mathcal{O}}$. A method m is called *non-interferent for signature* $\vec{k_v} \longrightarrow k_r$ if for any attacker level k_{obs} and any two (normally) terminating executions initiated with indistinguishable arguments according to $\vec{k_v}$ and indistinguishable heaps according to k_{obs} and the global policy ft, the results are indistinguishable by k_r and their heaps are indistinguishable according to the global policy.

Definition 5.3 (Non-interferent $\mathrm{JVM}_{\mathcal{C}}$ method). A method m is *non-interferent* w.r.t. a policy $\vec{k_v} \longrightarrow k_r$, if for every attacker level k_{obs} and 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 $\langle 1, \rho_1, \epsilon, h_1\rangle \rightsquigarrow_m^+ v_1, h_1'$, $\langle 1, \rho_2, \epsilon, h_2\rangle \rightsquigarrow_m^* v_2, h_2'$ and $h_1 \sim_{k_{\mathrm{obs}},\beta} h_2$, $\rho_1 \sim_{\vec{k_v},k_{\mathrm{obs}},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$, $h_1' \sim_{k_{\mathrm{obs}},\beta'} h_2'$ and moreover, if $k_r \leq k_{\mathrm{obs}}$ then $v_1 \sim_{\beta'} v_2$.

Then, a method is secure if it is side-effect safe and non-interferent.

Definition 5.4 (Secure $\mathrm{JVM}_{\mathcal{C}}$ method). A method m is *secure* 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[‖] 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 programs). The set of security signatures of a method m is defined as $\mathrm{Policies}_{\Gamma}(\mathrm{m}) = \{\ \Gamma_{\mathrm{m}}[\mathrm{k}] \mid \mathrm{k} \in \mathcal{S}\ \}$. Note that for coherence, each $\Gamma_m[k]$ should give type k to its local variable *this*. In the rest of the paper Γ will often be left implicit. We use it to define the notion of *secure program*.

Definition 5.5 (Secure $\mathrm{JVM}_{\mathcal{C}}$ program). A program is *secure* with respect to a table of method signatures Γ if for all its method m, m is safe with respect to all policies in $\mathrm{Policies}_{\Gamma}(\mathrm{m})$.

Example 5.6. Let P be a program that includes a method m and a class C with field f. Let m have variables x_H, y_L and a unique security signature $H, L \xrightarrow{H} H$. We assume that $\mathsf{ft}(f) = H$ with respect to the global mapping ft.

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

If the code of m is defined by:

$$
\begin{array}{rl}
 & \mathsf{new}\ C \\
 & \mathsf{store}\ y_L \\
 & \mathsf{load}\ x_H \\
 & \mathsf{ifeq}\ l_1 \\
 & \mathsf{load}\ y_L \\
 & \mathsf{push}\ 1 \\
 & \mathsf{putfield}\ f \\
l_1: & \mathsf{load}\ y_L \\
 & \mathsf{getfield}\ f \\
 & \mathsf{return}
\end{array}
$$

then method m is non-interferent because: starting from equal values for y_L (y_L represents the low part of the state as stated by the security signature), the low part of the memory is not modified at all during the execution of m. There are no assignments to the low fields: this respects the high write effect of the method required by the policy.

5.3. *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 JVM$_{\mathcal{O}}$ typing rules, except for putfield which needs a modification and the virtual call rule which is new. These two rules are given in Figure 10.

Concerning putfield only one constraint is added w.r.t. the previous JVM$_{\mathcal{O}}$ rule. The new constraint $k_h \leq \mathsf{ft}(f)$ restricts modification of fields to those fields whose security level is 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

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

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

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

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 the execution of the called method. The constraint $k_h \leq k'_h$ prevents the called method from updating 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 5.7. 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[††]:

[††] We omit the call of the initializer.

```
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.

Finally, we define typability of programs.

Definition 5.8 (Typable JVM$_{\mathcal{C}}$ program). A JVM$_{\mathcal{C}}$ program is typable w.r.t. safe CDRs $(\mathsf{region}_m, \mathsf{jun}_m)$ and with table of signatures Γ iff all methods m in P are typable w.r.t. $(\mathsf{region}_m, \mathsf{jun}_m)$ and all signatures in $\text{Policies}_\Gamma(\text{m})$.

5.4. *Type system soundness*

As for the $\mathrm{JVM}_{\mathcal{I}}$ and $\mathrm{JVM}_{\mathcal{O}}$, the type system is sound, in the sense that if a program is typable then it is secure.

Theorem 5.9. Let P be a typable $\mathrm{JVM}_{\mathcal{C}}$ program w.r.t. safe CDRs $(\mathsf{region}_m, \mathsf{jun}_m)$ and with table of signatures Γ. Then P is secure with respect to Γ.

6. $\mathrm{JVM}_{\mathcal{E}}$: The exception-handling extension of $\mathsf{JVM}_{\mathcal{C}}$

In this section we show how $\mathsf{JVM}_{\mathcal{C}}$ is extended with an exception handling mechanism. 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 (Myers, 1999).

Exceptions have an enormous impact on the control flow graph of programs, since many instructions become branching instructions. Thus, exceptions move the control flow graph from being an un-labelled directed graph to being a labelled directed graph, where the labels are either Norm (labels that do not correspond to any exception branch) or C (for an exception class C). The CDR analysis is then redefined in a labelled fashion, i.e. $\mathsf{region}(i, C)$ and $\mathsf{jun}(i, C)$.

Curbing this explosion in the control flow graph is essential for maintaining a minimum of precision in the information flow analysis; therefore, our analysis is parametrised by a pre-analysis (PA) that detects branches that will never be taken. The PA analyser may perform 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).

The extension of the type system to multiple exceptions is achieved by a fine-grained definition of control dependence regions that is parametrised by a class-analysis and an exception-analysis (which is part of the PA analyser). 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.

6.1. *Programs, memory model and operational semantics*

Programs The instruction set of the $\mathrm{JVM}_{\mathcal{C}}$ is extended with the bytecode throw. We assume that programs come equipped with a partial function $\mathsf{Handler}_m : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$ 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 point $i \in \mathcal{PP}$ then, if $\mathsf{Handler}_m(i, C) = t$, then the control will be transferred to program point t, and if $\mathsf{Handler}_m(i, C)$ is undefined (written as $\mathsf{Handler}_m(i, C) \uparrow$), the exception is uncaught in method m. In the first case the operand stack is reset to a singleton with the exception object on its top.

States $\mathrm{JVM}_{\mathcal{E}}$ 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 some exception-throwing instructions in $\mathrm{JVM}_{\mathcal{E}}$. (For brevity, we refrain from listing exceptional rules for $\mathsf{getfield}$, $\mathsf{putfield}$, $\mathsf{arraylength}$, $\mathsf{arrayload}$, $\mathsf{arraystore}$). There are three exceptional rules for the virtual call instruction. The first and the second model the cases when 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[‡‡]. 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 modelled by the function RuntimeExceptionHandling. Each instruction which performs an access on a reference (getfield f, putfield f and throw, arraylength, arrayload, arraystore) has 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 parametrised by a tag $\tau \in \{\mathsf{Norm}\} + \mathcal{C}$ to describe the nature of the transition (see the successor relation below). We will sometimes omit the tag τ in the notation $\rightsquigarrow_{m,\tau}$ for clarity.

Successor relation The successor relation is now decorated by an element (called *tag*) in $\{\mathsf{Norm}\} + \mathcal{C}$ in order to reflect the nature of the underlying semantics step: Norm 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 by the class analysis result and successors of an invokevirtual are approximated by the exception analysis result of the called method.

[‡‡] 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 bytecode verified. It is straightforward to show that our instrumented semantics coincides with the standard semantics if the exception analysis is semantically safe.

$$\frac{\langle i, \rho, os, h\rangle \rightsquigarrow_m \langle i', \rho', os', h'\rangle \text{ in } \mathrm{JVM}_{\mathcal{C}} \text{ semantics}}{\langle i, \rho, os, h\rangle \rightsquigarrow_{m,\mathsf{Norm}} \langle i', \rho', os', h'\rangle}$$

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

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

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

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

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

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

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 $\mathrm{JVM}_{\mathcal{E}}$

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

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

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

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

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

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

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

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

CDR properties CDR results are now associated not only with program points but also with tags:

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

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

CDR1: 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 $k = \mathsf{jun}(i, \tau)$;

CDR2: 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 $k = \mathsf{jun}(i, \tau)$;

CDR3: for all program points i, j and tag τ, if $j \in \mathsf{region}(i, \tau)$ and j is a return point then $\mathsf{jun}(i, \tau)$ is undefined;

CDR4: 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 $\mathsf{jun}(i, \tau_2) \in \mathsf{region}(i, \tau_1)$;

CDR5: for all program points i, j and tag τ, if $j \in \mathsf{region}(i, \tau)$ and j is a return point then for all tag τ' such that $\mathsf{jun}(i, \tau')$ is defined, $\mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$.

CDR6: for all program point i and tag τ_1, if $i \mapsto^{\tau_1}$ then for all tag τ_2, $\mathsf{region}(i, \tau_2) \subseteq \mathsf{region}(i, \tau_1)$ and if $\mathsf{jun}(i, \tau_2)$ is defined, $\mathsf{jun}(i, \tau_2) \in \mathsf{region}(i, \tau_1)$.

Junction points uniquely delimits ends of regions. CDR1 expresses that successors of branching points belongs (or ends) the region associated with the same kind as their successor relation. CDR2 says that a successor of a point in a region is either still in the same region or at this end. CDR3 forbids junction points for a region which contains a return point. CDR4 and CDR5 express properties between regions of a same program point but with different tags. CDR4 says that if two differently tagged regions end in distinct points, the junction point of one must belong to the region of the other. CDR5 imposes that the junction point of a region must be within every region which contains

a return point and is decorated with a different tag. CDR6 imposes that a return point i of tag τ_1 has a region $\mathsf{region}(i, \tau_1)$ large enough to contain all the others regions (and the eventual junction points) that are attached to i. CDR6 can be seen as an extension of CDR5 for the case $j = i$. Any region that contains a return point or start at an ending point must contain all other regions.

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

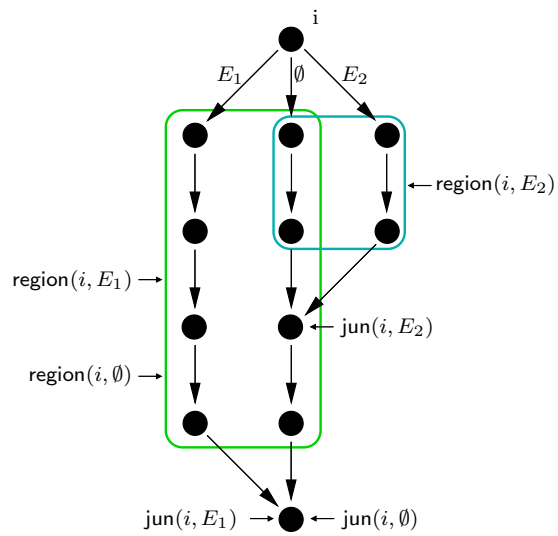


Fig. 13. Example of CDR in $\mathrm{JVM}_{\mathcal{E}}$. Only relevant tags are presented here.

6.2. *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 $\mathsf{JVM}_{\mathcal{C}}$. (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}.) The security level $\vec{k_r}$ (called *output level*) is now a list of security levels of the form $\{\mathsf{Norm} : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, 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.

The notion of *output indistinguishability* is adapted accordingly.

Definition 6.1 (Output indistinguishability). Given an attacker level k_{obs}, a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, an output level $\vec{k_r}$, indistinguishability of two final states in method m is defined by the clauses:

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

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

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

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

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

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 $\text{JVM}_{\mathcal{O}}$ (at page 33).

Definition 6.2 (Non-interferent $\text{JVM}_{\mathcal{E}}$ method). A method m is *non-interferent* w.r.t. a policy $\vec{k_v} \longrightarrow \vec{k_r}$, if for every attacker level k_{obs}, 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 $\langle 1, \rho_1, \epsilon, h_1\rangle \rightsquigarrow_m^+ r_1, h_1'$, $\langle 1, \rho_2, \epsilon, h_2\rangle \rightsquigarrow_m^+ r_2, h_2'$ and $h_1 \sim_{k_{\text{obs}},\beta} h_2$, $\rho_1 \sim_{\vec{k_v},k_{\text{obs}},\beta} \rho_2$, there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$ and $(r_1, h_1') \sim_{k_{\text{obs}},\beta',\vec{k_r}} (r_2, h_2')$.

Like in JVM$_\mathcal{C}$, we impose a *side-effect-safe* (*c.f.* page 41 for a formal definition) on methods. This notion is used when virtual call occurs in a high context in order to enforce that no modification is done on low information during the execution of the called method.

Definition 6.3 (Secure JVM$_\mathcal{E}$ method). A method m is *secure* 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 6.4 (Secure JVM$_\mathcal{E}$ program). A program is *secure* with respect to a table of method signature Γ if for all its method m, m is secure with respect to all policies in Policies$_\Gamma$(m)[§§].

6.3. *Typing rules*

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

The rule for ifeq is updated to flag with Norm the region that it considers. The virtual call needs now three typing rules. The first one corresponds to a normal control flow edge from the call site to its successor in the calling method. It is very similar to the rule in JVM$_\mathcal{C}$ except that invokevirtual is now a branching instruction because of the various exceptions that may be thrown at this point. We rely on the information excAnalysis(m_{ID})

[§§] Policies$_\Gamma$(m) has been defined on page 42.

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

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

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

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

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

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

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

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

to compute the level upper bound k_e of all exceptions that may be thrown by the method. The level k_e and the level k of the receiver object (that may be null and may throw a null pointer exception at runtime) are used to constrain the security environment and the next stack type. The second and third rules are parametrised by any exception e that may be thrown by m_{ID}. The second rule corresponds to the case where the exception is caught at the caller site, while the third rule corresponds to the case where the exception is not caught there. In each of these rules, the level $\vec{k'_r}[e]$ is used to constrain the corresponding region $\mathsf{region}(i, e)$.

Observe that the typing judgement is now parametrised by a tag $\tau \in \{\mathsf{Norm}\} + \mathcal{C}$. 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 6.5 (Typable method). A method m is typable w.r.t. a method signature table Γ, a global field policy ft, a policy sgn, and a CDR $\mathsf{region}_m : \mathcal{PP} \to \wp(\mathcal{PP})$ 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 $i, j \in \mathcal{PP}$, $e \in \{\mathsf{Norm}\} + \mathcal{C}$:

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 $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow$

6.4. *A typable example*

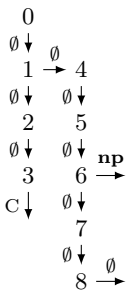
Figure 15 presents an example of a typable method `m`, giving the corresponding source code and the tagged flow graph. A method `m` may throw two kinds of exceptions: an exception of class `C` depending on the value of `x`, and an exception of class **np** depending on the values of `x` and `y`. Normal return depends on `y` because execution terminates normally only if it is not *null*. The method `m` is typable with the policy $m : (\mathit{this} : L,\ x : L,\ y : H) \xrightarrow{H} \{\mathsf{Norm} : H,\ C : L,\ \mathbf{np} : H\}$ with the CDR (given only for branching points), the type stacks and the security environment given in Figure 15.

Figure 16 gives another example[¶¶] where fine grain exception handling is necessary for the code to be typable. Here the update $t_L = 1$ at point 6 is accepted if and only if $se(6)$ is low. This fragment is accepted by our type system since, thanks to the fine grain

¶¶ To keep the example short here we give a compressed version of the compiled code.

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

$0 : \mathsf{load}\ x$
$1 : \mathsf{ifeq}\ 4$
$2 : \mathsf{new}\ C$
$3 : \mathsf{throw}$
$4 : \mathsf{load}\ y$
$5 : \mathsf{push}\ 3$
$6 : \mathsf{putfield}\ f$
$7 : \mathsf{const}\ 1$
$8 : \mathsf{return}$



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

$\mathsf{region}(1, \mathsf{Norm}) = \{2, 3, 4, 5, 6, 7, 8\}$ $\mathsf{jun}(1, \mathsf{Norm})$ undef.

$\mathsf{region}(6, \mathsf{Norm}) = \emptyset$ $\mathsf{jun}(6, \mathsf{Norm}) = 7$ $\mathsf{region}(6, \mathbf{np}) = \{7, 8\}$ $\mathsf{jun}(6, \mathbf{np})$ undef.

Fig. 15. Typable methods at source and bytecode level

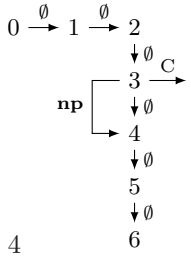
regions, typing rule for virtual call only propagates exception levels $\vec{k_r}[\mathbf{np}] = H$ in the region $\mathsf{region}(3, \mathbf{np})$ (instead of $\mathsf{region}(3, C)$).

6.5. *Type system soundness*

We finish this section by stating the type soundness theorem for $\mathrm{JVM}_{\mathcal{E}}$.

Theorem 6.6. Let P be a $\mathrm{JVM}_{\mathcal{E}}$ typable program w.r.t. safe CDRs $(\mathsf{region}_m, \mathsf{jun}_m)$ and a table of signatures Γ. Then P is secure with respect to Γ.

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



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

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

Fig. 16. Typable fragment with virtual call

7. Machine-checked proof

We have formalized within the Coq proof assistant the information flow type system for the JVM$_{\mathcal{E}}$ fragment, and proved formally its soundness. Moreover, we have formalized executable checkers for the CDR properties and for typability, and proved formally their soundness—in the sense that an annotated program that is accepted by the CDR checker satisfies the CDR properties, and that an annotated program that is accepted by the information flow checker is typable w.r.t. our type system and non-interferent. The statement of soundness of the type system hinges on a formalization of the operational semantics of the JVM, and of the notion of non-interferent program.

This section presents an overview of the proof. We start with a short discussion on the relevance of formal proofs, and some statistics on the development. Then, we describe our formalization of the semantics of the JVM and of the notion of non-interferent program. Further, we discuss our approach for proving unwinding lemmas and for constructing executable checkers. We conclude with an example.

7.1. *Motivation and overview*

Our formalization participates to an increasing trend of using proof assistants for building machine-checked proofs of the metatheory of programming languages (Aydemir et al., 2005). One primary motivation for using proof assistants is that they provide significant help for managing satisfactorily the complexity of type soundness proofs. In our view, the complexity of information flow type systems for full-fledged languages makes machine-checked formalizations extremely important, if not compulsory, for three concommitting reasons. First, the formalization of the operational semantics contains a significant number of rules; for example, the JVM virtual call has 5 different transitions (call on a null reference which generates a null pointer exception that may be caught or not, normal termination of the callee, termination by an exception that may be caught or not in the caller context). Second, the type system contains over 60 rules, and many rules have a large (up to 10) number of premises, see e.g. Figure 14. Third, the proof of non-interference relies on unwinding lemmas that require reasoning about two program executions, leading to a very large number of cases in proofs. Moreover, the proof of correctness of the type system is stratified: one must first prove that the CDR checker is correct (assuming that the pre-annotations are), then prove that the type system is correct (assuming that the pre-annotations and CDR checker are)—we do not provide a means to check the correctness of pre-annotations; this is left for future work. Summarizing, we have proved the following theorem; the second part corresponds to the Theorem 6.6.

Theorem 7.1.

1 CDR and IF can be checked by executable functions.

2 For every annotated program P,

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

where:

— the security condition is formalized by the predicate SAFE;

— the correctness of program annotations is formalized by the predicate PA;

— the CDR properties (given in Section 6) are formalized by the predicate CDR;

— the notion of typable program is formalized by the IF predicate.

Foundational Proof-Carrying Code (Appel, 2001) provides another motivation for machine-checked proofs: a certified checker can be used to reduce the Trusted Computing Base of a security architecture for mobile code. Figure 17 describes how our type system would operate in a Proof-Carrying Code scenario. The left-hand side of the figure corresponds to the code producer, which should produce a certificate in the form of the results of the PA, CDR, and IF analyzers. Our formalization focuses on the right-hand side of the figure, which corresponds to the code consumer:

1 the PA checker verifies that annotations provided by the PA analyzer are correct;

2 the CDR checker verifies that regions provided by the CDR analyzer verify the safe over-approximation properties of Section 6;

3 the IF checker verifies type correctness in the style of lightweight bytecode verification.

One virtue of Foundational Proof Carrying Code is that it yields a significantly simpler Trusted Computed Base: specifically, the Trusted Computing Base is reduced to the Coq type checker and the formal definition of non-interference, as shown in Figure 18—contrast with Figure 17 where formal proofs were not mentioned.

The full Coq development is about 17,000 lines; its main components are: the operational semantics of the JVM, the definition of the type system, and the proof of soundness of the type system. Each of them is a significant formalization in itself; Figure 19 gives indications on the size of the components. The development is available at

`http://www.irisa.fr/celtique/pichardie/ext/iflow/`

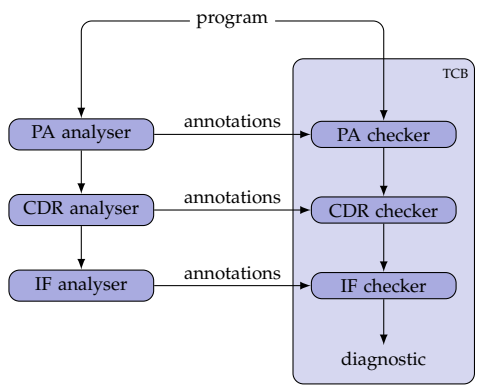


Fig. 17. Information flow analyzer and checker

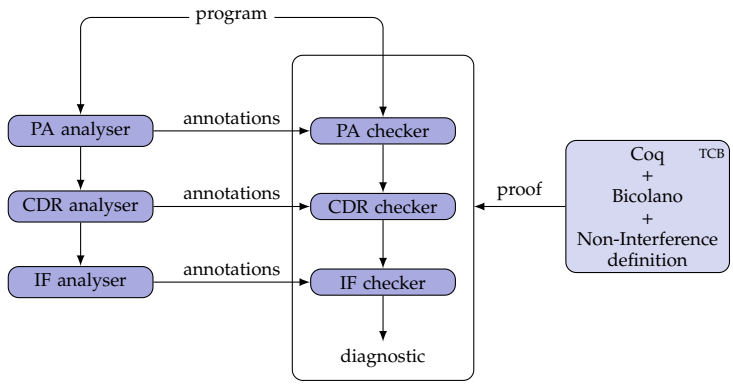


Fig. 18. Information flow analyser and checker with Coq TCB

7.2. *Formal semantics*

The development relies on a formal semantics of the JVM in Coq, called Bicolano, 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, multi-threading, dynamic class loading, garbage collection, 64-bit arithmetic and floats.

	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. 19. Size of the Coq development

The core of Bicolano is a small-step operational semantics which describes the dynamic behavior of a bytecode program according to the JVM specification. The small-step semantics is formalized as an inductively defined relation $\cdot \to \cdot$ between states of the virtual machine, where a state consists of a heap, and a stack frame; Figure 20 presents the small-step semantics for method calls and return. In addition, Bicolano formalizes a mix-step semantics, in which method calls are performed in one step—as in Section 5; in particular, the mix-step semantics for virtual method invocation appears in Figure 9 (page 40). The mix-step semantics is also formalized as an inductively defined relation $\cdot \to \cdot$ between states of the virtual machine, but uses a simplified notion of state in which the stack frame is replaced by a single frame. Both semantics are equivalent, in the sense that the big-step semantics induced by the two semantics coincide; Bicolano formally establishes this equivalence between them. The crux of the proof is a lemma stating that each execution of the JVM to a final value implies the corresponding judgment of the mix-step semantics.

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

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

We briefly comment on the role of the two semantics in our work: the mix-step seman-

tics brings important simplifications in the definition of state indistinguishability and in the soundness proofs, and hence we use it to machine-check type soundness. The small-step semantics serves as a reference formalization, and hence the final theorem is stated using the small-step semantics.

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

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

Fig. 20. Small-step semantics rule for virtual method call

For the purpose of the information-flow type system, we have also developed 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 21 gives an example of instrumented transition. Annotations take the form of flags safe attached to program points where the pre-analyzer predicts that no exception may be thrown here; moreover, the instrumented semantics constrains that exceptions can only be raised at program points which are not annotated as safe. Assuming that the annotations are correct, the mix-step semantics and the instrumented mix-step semantics coincide.

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

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

Fig. 21. Example of annotated semantic rule

7.3. *Formalization of the security condition*

The definition of non-interferent program, and the proof of soundness of the type system, rely on the notion of state indistinguishability. The main issue with the formalization of the latter is the notion of finite bijection used to relate heaps in the two program executions. Instead of parametrizing state indistinguishability by a finite bijection β from A to A, we have found it more convenient to parametrize the definition by a pair of finite functions (β_1, β_2) from natural numbers to A. Informally, the finite functions β_1 and β_2 are determined by the execution of the first and second programs respectively: β_1 is extended when a new low object is created during execution of the first program, and likewise for β_2. This allows us to define an instrumented operational semantics in which the partial function β_i is part of the program state; see next subsection.

7.3.1. *Finite maps* A finite function from `nat` to `A` is modelled by the dependent type:

```
Record ffun (A:Type) : Set := make {
  lookup :> nat -> option A;
  domain_size : nat;
  lookup_domain : forall n, n<domain_size <-> (lookup n<>None)
}.
```

Hence, an element of type `ffun A` is given by three elements: a partial function from natural numbers to `A`, modelled as a type-theoretical function `lookup` from natural numbers to `option A`; a natural number `domain_size` that gives the current size of the function domain and a proof `lookup_domain` that the domain of `lookup` is equal to the set of the numbers smaller than `domain_size`. In order to carry our reasoning, we have built a library that includes operators and lemmas to manipulate and reason about finite maps.

7.3.2. *Indistinguishability relations* We have formalized indistinguishability relations and built a library of basic results to reason about indistinguishability—the library contains more than 100 lemmas.

Indistinguishability relations are defined w.r.t. a pair of finite functions from natural numbers to locations. As in the paper, the formalization defines indistinguishability incrementally for values, operand stacks, local variables, heaps and states. As an example the signature for the heap indistinguishability relation is the following:

```
hp_in (newArT : Method * PC -> L.t') (ft:FieldSignature -> L.t')
      (b b': FFun.t Location) (h h': Heap.t) : Prop
```

Here the parameter `newArT` gives the annotation for array allocation (one array content type at each `newarray` location in the program). `L.t'` is the type of information flow types extended to arrays (see Section 4). The finite functions `b` and `b'` correspond to the partial bijection of the previous sections; the predicate `hp_in` enforces that the two functions satisfy some expected properties, e.g. being bijective.

7.4. *Soundness proof methodology*

The main technical artifact of the soundness proof is a (mix-step) defensive semantics that keeps track of type information and partial bijections, and is particularly useful to reason on well-typed executions.

The defensive semantics manipulate states of the form (in Coq syntax):

```
Inductive state : Type :=
  | intra : IntraNormalState -> TypeStack -> ffun Location -> state
  | ret   : Heap.t -> ReturnVal -> ffun Location -> state.
```

The following Coq code presents the rule corresponding to object allocation.

```
Inductive NormalStep_new (c:ClassName) (m:Method) (sgn:sign) :
   IntraNormalState -> TypeStack -> ffun Location ->
   IntraNormalState -> TypeStack -> ffun Location -> Prop :=
| new : forall h pc pc' s l loc h' st b,

  next m pc = Some pc' ->
  Heap.new h p.(prog) (Heap.LocationObject c) = Some (pair loc h') ->

  NormalStep_new c m sgn
  (pc,(h,s,l)) st b
  (pc',(h',(Ref loc::s),l)) ((se pc)::st) (newb (se pc) b loc).
```

In this example, `(pc,(h,s,l))` and `(pc',(h',(Ref loc::s),l))` represent the (JVM) states before and after executing the instruction, while `st` and `((se pc)::st)` represent the corresponding type stacks and `b` and `(newb (se pc) b loc)` are the partial bijections. The `newb` operator is used to extend the domain of a partial bijection depending on the current security level (given here by `(se pc)`).

7.5. *Executable checkers*

The first item of Theorem 7.1 is proved by formalizing boolean-valued functions $\mathsf{check}_{\mathrm{CDR}}$ and $\mathsf{check}_{\mathrm{IF}}$ that enforce the predicates CDR and IF respectively. The function $\mathsf{check}_{\mathrm{CDR}}$ performs a direct verification of the CDR properties for each method. What is left for future work is to implement a verifier $\mathsf{check}_{\mathrm{PA}}$ that entails PA.

Functions $\mathsf{check}_{\mathrm{CDR}}$ and $\mathsf{check}_{\mathrm{IF}}$ are executable Coq programs that have been successfully extracted into Ocaml. We have tested then on a *Tax Calculation* Java program inspired from the case study proposed by Deng and Smith (Deng and Smith, 2004). The full Java source program is given in Figure 22 with its information flow type annota-

tions given in a *Jif-like* syntax, and safety annotations given in comments. The program computes income taxes from an input array of taxable incomes and marital status. The program takes as argument an array `input` of inputs and a tax table `taxTable`. Then, for each index `i` in the array range, it performs a binary search to find an index `lo` such that

$$\texttt{taxTable}[\texttt{lo}].\texttt{brackets} \leq \texttt{input}[\texttt{i}].\texttt{taxableIncome} < \texttt{taxTable}[\texttt{lo}+1].\texttt{brackets}$$

and updates the output array `out.tax[i]` with the computed tax (`taxTable[lo].married` or `taxTable[lo].single` depending on the marital status) and increment a counter `out.married_nb` or `out.single_nb` to count the whole number of married and single tax returns. The taxable incomes (field `taxableIncome`) and the array content of the income taxes (field `tax`) are given a high security level while other data are low.

We briefly comment on the annotations for runtime exceptions (`NP` means NullPointer, `NAS` means NegativeArraySize, `AOB` means ArrayOutofBounds). Most of these annotations can be easily proved with a simple null pointer analysis that maintains the invariant $\texttt{this} \neq \texttt{null}$. The others require more complex arithmetic reasoning, for example a relational numeric static analysis (Besson et al., 2010).

8. 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 companion work on preservation of information flow types by compilation (Barthe et al., 2006), our results provide a sound basis for end-to-end security solutions for Java-based mobile code. The most immediate direction for further work is to extend the type system to a concurrent fragment of

```
class Output {
    int{L} single_nb;  int{L} married_nb;   int[]{L[H]} tax;

    Output{L}(int nbPeople) {
        single_nb = 0; // no NP exception
        married_nb = 0; // no NP exception
        tax = new int[nbPeople]; // no NP exception, no NAS exception
    }

    void updateMarried{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data; // no NP exception, no AOB exception
        married_nb++; // no NP exception
    }

    void updateSingle{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data; // no NP exception, no AOB exception
        single_nb++; // no NP exception
    }
}

class Input {int{H} taxableIncome;   boolean{L} maritalStatus;}
class Tax {int{L} single;   int{L} married;   int{L} brackets;}

class TaxCalculation {

    Output{L} main{L}(Input[]{L[L]} input, Tax[]{L[L]} taxTable) {
        Output{L} out = new Output(input.length);
        for (int{L} i=0; i < input.length; i++) {
            int{H} lo = 0;
            int{H} hi = taxTable.length;
            try {
              while (lo+1 < hi) {
                int{H} mid = (lo + hi) / 2;
                if (input[i].taxableIncome
                      < taxTable[mid].brackets) //no AOB exception
                  {hi = mid;}
                else {lo = mid;};
              };
            } catch (NullPointerException e){};
            if (input[i].maritalStatus)
                { out.updateMarried(i,taxTable[lo].married);}
            else
                { out.updateSingle(i,taxTable[lo].single);};
        };
        return out;
    }
}
```

Fig. 22. The Tax Calculation program.

the Java Virtual Machine, and to support declassification. As an initial step towards dealing with concurrency, we have proposed a sound information flow type system for a concurrent extension of the JVM (Barthe et al., 2010; Barthe and Rivas, 2011); the extension supports objects, methods, multi-threading and dynamic thread creation, but not exceptions, locks and synchronization primitives. The extension builds upon Russo and Sabelfeld (Russo and Sabelfeld, 2006) idea to constrain the behavior of schedulers so that high branches execute uninterruptedly, thereby avoiding internal timing leaks. In our setting, the idea of secure scheduler is modeled by making the behavior of the scheduler depend on the security environment.

The applicability of the type system could be enhanced significantly by considering more flexible policies that allow some controlled form of information release. In (Barthe et al., 2008), we show in the setting of the $\mathrm{JVM}_{\mathcal{I}}$ language how to adapt our type system so that it provides support for delimited non-disclosure, a specific form of declassification that enables to declassify the value of a variable at a specified program point. Technically, the prime difference between (Barthe et al., 2008) and the current work is that the former considers local policies, i.e. there is a security policy for each program point; it allows the security level of variables to change during execution, so that variables can be declassified. The type system that enforces delimited non-disclosure is built systematically from the baseline type system for non-interference, and we foresee no difficulty in extending the results of (Barthe et al., 2008) to richer fragments of the JVM.

References

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. (1999). A core calculus of dependency. In *Principles of Programming Languages*, pages 147–160. ACM Press.

Agat, J. (2000). *Type Based Techniques for Covert Channel Elimination and Register Allocation.* PhD thesis, Chalmers University of Technology and Gothenburg University.

Amtoft, T., Bandhakavi, S., and Banerjee, A. (2006). A logic for information flow in object-

oriented programs. In Morrisett, G. and Jones, S. P., editors, *Principles of Programming Languages*, pages 91–102. ACM.

Appel, A. W. (2001). Foundational proof-carrying code. In Halpern, J., editor, *Logic in Computer Science*, page 247. IEEE Press. Invited Talk.

Askarov, A. and Sabelfeld, A. (2005). Security-typed languages for implementation of cryptographic protocols: A case study. In *European Symposium On Research In Computer Security*, number 3679 in Lecture Notes in Computer Science. Springer-Verlag.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. (2005). Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*. Springer-Verlag.

Banerjee, A. and Naumann, D. (2005). Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177. Special Issue on Language-Based Security.

Barthe, G., Basu, A., and Rezk, T. (2004). Security types preserving compilation. In Steffen, B. and Levi, G., editors, *Verification, Model Checking and Abstract Interpretation*, number 2934 in Lecture Notes in Computer Science, pages 2–15. Springer-Verlag.

Barthe, G., Cavadini, S., and Rezk, T. (2008). Tractable enforcement of declassification policies. In *IEEE Computer Security Foundations Symposium*. IEEE Press.

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

Barthe, G., Pichardie, D., and Rezk, T. (2007). A certified lightweight non-interference Java bytecode verifier. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 125–140. Springer-Verlag.

Barthe, G. and Rezk, T. (2005). Non-interference for a JVM-like language. In Fähndrich, M., editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press.

Barthe, G., Rezk, T., Russo, A., and Sabelfeld, A. (2010). Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3).

Barthe, G. and Rivas, E. (2011). Static enforcement of information flow policies for a concurrent

jvm-like language. In Bruni, R. and Sassone, V., editors, *Proceedings of TGC'11*, volume xxxx of *Lecture Notes in Computer Science*. Springer.

Bernardeschi, C. and Francesco, N. D. (2002). Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In Cortesi, A., editor, *Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag.

Besson, F., Jensen, T. P., Pichardie, D., and Turpin, T. (2010). Certified result checking for polyhedral analysis of bytecode programs. In Wirsing, M., Hofmann, M., and Rauschmayer, A., editors, *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 253–267. Springer.

Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V., and Zanon, G. (2002). Checking secure interactions of smart card applets: Extended version. *Journal of Computer Security*, 10(4):369–398.

Bonelli, E., Compagnoni, A. B., and Medel, R. (2005). Information flow analysis for a typed assembly language with polymorphic stacks. In Barthe, G., Grégoire, B., Huisman, M., and Lanet, J.-L., editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3956 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag.

Deng, Z. and Smith, G. (2004). Lenient array operations for practical secure information flow. In *CSFW*, pages 115–124. IEEE Computer Society.

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):271–321.

Genaim, S. and Spoto, F. (2005). Information flow analysis for Java bytecode. In Cousot, R., editor, *Verification, Model Checking and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag.

Girard, P. (1999). Which security policy for multiapplication smart cards? In *Workshop on Smart Card Technology*. USENIX Association.

Hammer, C., Krinke, J., and Snelting, G. (2006). Information flow control for java based on path conditions in dependence graphs. In *Symposium on Secure Software Engineering*. IEEE Press.

Hankin, C., Nielson, F., and Nielson, H. R. (2005). *Principles of Program Analysis*. Springer-Verlag. Second Ed.

Hedin, D. and Sands, D. (2006). Noninterference in the presence of non-opaque pointers. In *Computer Security Foundations Workshop*. IEEE Press.

Hunt, S. and Sands, D. (2006). On flow-sensitive security types. In *Principles of Programming Languages*, Charleston, South Carolina, USA. ACM Press.

Kobayashi, N. and Shirane, K. (2002). Type-based information analysis for low-level languages. In *Asian Programming Languages and Systems Symposium*, pages 302–316.

Leroy, X. (2002). Bytecode verification on java smart cards. *Softw., Pract. Exper.*, 32(4):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.

Medel, R., Compagnoni, A. B., and Bonelli, E. (2005). A typed assembly language for non-interference. In *Italian Conference on Theoretical Computer Science*, volume 3701 of *Lecture Notes in Computer Science*, pages 360–374. Springer-Verlag.

Montgomery, M. and Krishna, K. (1999). Secure object sharing in Java Card. In *Workshop on Smart Card Technology*. Usenix.

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):527–568. Expanded version of a paper presented at POPL 1998.

Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–241. ACM Press. Ongoing development at `http://www.cs.cornell.edu/jif/`.

O'Neill, K. R., Clarkson, M. R., and Chong, S. (2006). Information-flow security for interactive programs. In *Computer Security Foundations Workshop, IEEE*, pages 190–201. IEEE Computer Society.

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

Rezk, T. (2006). *Verification of confidentiality policies for mobile code*. PhD thesis, Université de Nice Sophia-Antipolis.

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

Russo, A. and Sabelfeld, A. (2006). Securing interaction between threads and the scheduler. In *Computer Security Foundations Workshop*, pages 177–189. IEEE Press.

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

Volpano, D. and Smith, G. (1997). A type-based approach to program security. In Bidoit, M. and Dauchet, M., editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag.

Volpano, D. and Smith, G. (1998). Probabilistic noninterference in a concurrent language. In *Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts. IEEE Press.

Yu, D. and Islam, N. (2006). A typed assembly language for confidentiality. In *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag.