Math. Struct. in Comp. Science (2013), *vol.* 23, *pp.* 1032–1081. © Cambridge University Press 2013
doi:10.1017/S0960129512000850 First published online 17 May 2013

A certified lightweight non-interference Java bytecode verifier[†]

GILLES BARTHE[‡], DAVID PICHARDIE[§] and TAMARA REZK[¶]

[‡]*IMDEA Software Institute, Campus Montegancedo,*
28660-Boadilla del Monte, Madrid, Spain
Email: gilles.barthe@imdea.org
[§]*INRIA Rennes – Bretagne Atlantique, Campus de Beaulieu,*
35042 Rennes Cedex France.
Email: David.Pichardie@inria.fr
[¶]*INRIA Sophia Antipolis – Méditerranée,*
2004 Route des Lucioles, BP 93,
06902 Sophia Antipolis Cedex France
Email: Tamara.Rezk@inria.fr

Received 20 December 2010; revised 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 the 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 formalisation is that we have extracted from our proof a certified lightweight bytecode verifier for information flow. Our work provides, to the best of our knowledge, the first sound and certified information flow type system for such an expressive fragment of the JVM.

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, that is, it 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

[†] This work was partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS and French Brittany region project CertLogS.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

concentrates on who accesses sensitive information, but not on 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 properties ensured by security models for mobile code. In contrast to security models based on access control, language-based security focuses on information-flow policies that track how sensitive information is propagated during execution.

Starting from the seminal work Volpano and Smith (1997), type systems have become prominent in approaches to the 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 with 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 on the decentralised label model and offers a flexible and expressive framework for defining information flow policies for Java programs.

The central contribution of this paper is the definition and 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, for example, initialisation, multi-threading and garbage collection. The type system builds on work in Barthe *et al.* (2004) and Barthe and Rezk (2005) by the authors of the current paper, which proposes a sound information flow type system for a simple assembly language that closely resembles the $\mathrm{JVM}_{\mathcal{I}}$ fragment of the current paper, and an object-oriented language that resembles the $\mathrm{JVM}_{\mathcal{O}}$ fragment of the current paper extended with a simplified treatment of exceptions. The current paper adopts many of the ideas and techniques of Barthe and Rezk (2005), but is also a substantial improvent on it in terms of:

— *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 a 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, requiring us to machine-check proofs rather than use pen-and-paper arguments. A second contribution of our work is a formalisation 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 through lightweight bytecode verification, that is, it takes a JVM program with security annotations (and some additional information on the control

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

dependence regions of programs), and checks that the program respects the security policy indicated by the annotations.

1.1. *Contents of the paper*

We begin with an overview of related work in Section 2, and in the following sections analyse increasingly complex fragments of the JVM:

— In Section 3, we study the $\mathrm{JVM}_{\mathscr{I}}$, which 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 also define and discuss operand stack indistinguishability. The definitions and type system for the $\mathrm{JVM}_{\mathscr{I}}$ are adapted from our earlier work Barthe *et al.* (2004).

— In Section 4, we study the $\mathrm{JVM}_{\mathscr{O}}$, which is an object-oriented extension of the $\mathrm{JVM}_{\mathscr{I}}$, which includes features such as dynamic object creation, instance field accesses and updates, and arrays, and is expressive enough for compiling the intra-procedural statements of Banerjee and Naumann (2005). In this section, we also define and discuss heap indistinguishability. The main difficulty is defining a sufficiently fine type system that allows public arrays to handle secret information.

— In Section 5, we study the $\mathrm{JVM}_{\mathscr{C}}$, which is a procedural extension of the $\mathrm{JVM}_{\mathscr{O}}$ with method calls, and is expressive enough to compile the language of Banerjee and Naumann (2005). The main difficulty is handling information leakages caused by dynamic method dispatch.

— In Section 6, we study the $\mathrm{JVM}_{\mathscr{E}}$, which extends the $\mathrm{JVM}_{\mathscr{C}}$ by adding exceptions. The main difficulty is handling information leakages caused by exceptions, especially when they escape the scope of the method in which they are raised.

For each fragment, we: define the syntax and semantics of programs; formulate the security policy and the typing rules; and, finally, prove the soundness of the type system. Section 7 provides additional details of 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 formalisation, and the addition of several examples.

1.2. *Notation and conventions*

For every function $f \in A \to B$, $x \in A$ and $v \in B$, we write $f \oplus \{x \mapsto v\}$ to denote the unique function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$. We also write $A^\star$ to denote the set of A-stacks for every set A. We use hd, tl, $::$ and $++$, respectively, to denote the head, tail, cons and concatenation operations on stacks.

For simplicity, the examples throughout the paper will take as the partial order of security levels $\mathscr{S} = \{L, H\}$ with $L \leqslant H$, where H is the high level used for confidential data and L is the low level used for observable data.

Finally, we will also assume that all methods return a result – this is a harmless departure from Java that allows us to avoid duplicating many definitions. However, this

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

assumption is only made here for the sake of presentation, and the formal proofs do in fact consider both the cases of methods returning a result and methods returning no result.

2. Related work

2.1. *Prior work*

In order to realise our goal of defining a sound information flow type system for (sequential) Java bytecode, we draw on the work in several earlier papers that address its features in isolation. For example, our approach to dealing with unstructured code is inspired by Kobayashi and Shirane (2002), which defined the first information flow type system for a low-level language for a subset of the JVM similar to the $\text{JVM}_{\mathscr{I}}$ machine defined in Section 3. We adopt from their type system the use of:

(i) control dependence regions;
(ii) security environments.

Similar concepts are used in Agat (2000), which 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 behaviour may leak information.

Many ideas of the type system originate from Jif (Myers 1999), which is an information-flow aware extension of Java that builds on the decentralised 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. However, 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 (2005) developed 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) a 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 a number of language features such as exceptions and arrays.

2.2. *Companion papers*

A companion paper, Barthe *et al.* (2006), establishes a formal correspondence between the source type systems of Banerjee and Naumann (2005) and the one we use here in the form of a type-preservation result showing that the compiler maps typable Java programs

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 briefly discusses work we have done in extending our type system to include 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).

2.3.1. *Java.* A hypothesis underlying Myers (1999), Banerjee and Naumann (2005) and the current paper is a semantics in which references are opaque, that is, the only observations that can be made about a reference are those about the object to which it points. Hedin and Sands (2006) observed that implementations of the Java Virtual Machine commonly violate this assumption and allow references to be cast to an integer. Hedin and Sands also exhibited a typable Jif program that does not use declassification but leaks information by 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 is indeed satisfied by many implementations of the JVM. In addition to demonstrating the attack, Hedin and Sands showed 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 it 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 analyse 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.

2.3.2. *JVM.* Bieber *et al.* (2002) provided an early study of information flow in the JVM. Their method consists of 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 starting with Bernardeschi and Francesco (2002), Bernardeschi and co-workers have also suggested the use of abstract interpretation and model-checking techniques to verify secure information.

There are some alternative approaches to verifying the information flow properties of bytecode programs. For example, Genaim and Spoto (2005) showed how to represent

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

information flow for Java bytecode through boolean functions. This representation then allows checking through binary decision diagrams. The analysis is fully automatic and does not require that methods be annotated with security signatures.

2.3.3. *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 developed a typed assembly language (TAL) based on a conventional RISC assembly language and showed that typable programs of System F can be compiled into typable TAL programs (Morrisett *et al.* 1999).

The study of non-interference for typed assembly languages was initiated by Bonelli, Compagnoni and Medel, who developed a sound information flow type system for a simple assembly language called SIFTAL (Bonelli *et al.* 2005). A novelty of SIFTAL is the introduction of pseudo-instructions, which 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 used in the JVM to handle method calls, the stack of continuations is used for flow control within the body of a method. The use of pseudo-instructions allows the formulation of global constraints in the type system, and thus a guarantee of non-interference. A more recent paper by the same authors (Medel *et al.* 2005) and Yu and Islam (2006) both avoid the use of pseudo-instructions. Yu and Islam also consider a richer assembly language and prove type-preserving compilation for an imperative language with procedures.

2.3.4. *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 and co-workers (Amtoft *et al.* 2006), which allows the verification of non-interference for an object-oriented language by using independence assertions inspired from separation logic, the type system of Hunt and Sands (2006) and the aforementioned analysis in Hammer *et al.* (2006).

While flow sensitivity adds useful expressiveness to a source language, its role is less prominent in the case of type systems, like ours, that aim to verify bytecode (or executable code) since there are 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 $\mathrm{JVM}_{\mathscr{I}}$ 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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

$instr$	::=	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 the $\mathrm{JVM}_{\mathcal{I}}$.

3.1. *Programs, memory model and operational semantics*

3.1.1. *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 $\mathcal{V}$ be the set of values, that is, $\mathcal{V} = \mathbb{Z}$. 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.

3.1.2. *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, which 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.

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

In Figure 2, $\underline{op}$ denotes the standard interpretation of operation op in the domain of values $\mathcal{V}$. The semantics of each instruction is standard:

- push c pushes a constant c onto the top of the operand stack.
- binop op pops the two top operands of the stack and pushes the result of the binary operation op using these operands.
- pop just pops the top of the operand stack.
- swap swaps the two top operand stack values.
- return ends the execution with the top value of the operand stack.
- load x pushes the value currently found in local variable x onto the top of the operand stack.
- store x pops the top of the stack and stores it in local variable x.
- 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 continues to the next program point.
- goto j unconditionally jumps to program point j.

The transitive closure of $\leadsto$ is denoted by $\leadsto^{+}$.

3.1.4. *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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

Fig. 2. Operational semantics for the $\mathrm{JVM}_{\mathscr{I}}$.

always 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. Also, j is a return point if it corresponds to a return instruction. Formally, the successor relation $\mapsto \subseteq \mathscr{PP} \times \mathscr{PP}$ 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, $i \mapsto i+1$.

We also define for each program P its set $\mathscr{PP}_r$ of return points, that is, program points with no successor, or, equivalently, program points that are mapped to a return instruction. By abuse of notation, we write $i \mapsto$ if $i \in \mathscr{PP}_r$.

3.2. *Non-interference*

The security policy is given by a lattice $(\mathscr{S}, \leqslant)$ of security levels and the policy of the program. In the $\mathrm{JVM}_{\mathscr{I}}$ 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 rest of the paper, we will often view $\vec{k_v}$ as a partial mapping from variables to security levels. The notion of a 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 $\rho, \rho' : \mathscr{X} \rightharpoonup \mathscr{V}$, we have $\rho \sim_{\vec{k_v}, k_{\mathrm{obs}}} \rho'$ if ρ and ρ' have the same domain and $\rho(x) = \rho'(x)$ for all $x \in \mathsf{dom}(\rho)$ such that $\vec{k_v}(x) \leqslant k_{\mathrm{obs}}$.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

Moreover, a program P is *non-interferent* with respect to policy $\vec{k_v} \longrightarrow k_r$ if and only if for all attacker levels k_{obs}, P is non-interferent with respect to $\vec{k_v} \longrightarrow k_r$ and k_{obs}.

Our definition of non-interference is termination-insensitive, that is, it does not take into account non-terminating executions of programs. Stronger definitions, which reject programs whose termination behaviour depends 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 section points out some problems arising when we try to enforce non-interference for unstructured programs and provides 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 operations in the branches 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 the security environment.

3.3.1. *Direct flows.*

In a high-level language, direct flows are prevented by the typing rule for assignments, which is usually of the form

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

(Volpano and Smith 1997), 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 $k \leqslant \vec{k_v}(x)$ ensures that the value stored in x does not depend on any variable whose security level is not strictly less than that of x, and thus that there is no illicit flow to x.

In a low-level language where intermediate computations are performed using an operand stack, direct information flows are prevented by assigning a security level to each value in the operand stack through a so-called *stack type*, and by rejecting programs that

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

attempt to store a value in a low variable when the top of the stack type is high:

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

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 the 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 $H \leqslant L$ generated by the typing rule for assignment is violated. Likewise, the low-level counterpart

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

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

3.3.2. *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

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

(Volpano and Smith 1997), which ensures that the write effects of c_1 and c_2 are not less than 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 just enforce local constraints, that is, they cannot refer only to the current program point and its successors. Instead, the typing rules must also enforce global constraints that prevent low assignments and updates occurring 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.

3.3.3. *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 use 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 picture on the right, which corresponds to an if-then-else statement, the branching point i does not belong to its region, whereas in the picture on the left, which corresponds to a while-do statement, the branching point i does belong to its region.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

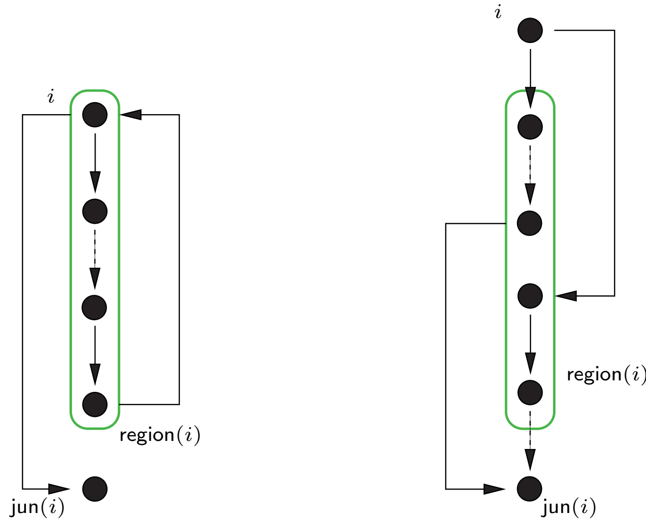

Fig. 3. (Colour online) Example of CDR for a while and an if construct.

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 $\mathsf{jun}(i)$ (if defined), and $\mathsf{jun}(i)$ is the sole exit from the region of i: in particular, if $\mathsf{jun}(i)$ is defined, there should be no return instruction in $\mathsf{region}(i)$.

Definition 3.3 (safe CDR structure). A control dependence region (CDR) structure $(\mathsf{region}, \mathsf{jun})$ 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), we have $k \in \mathsf{region}(i)$ 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.

Section 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.

3.3.4. *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 the requirement 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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

ifeq bytecode is of the form

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

On the other hand, the typing rules for instructions with a write effect, for example, 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

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

The combination of the two rules allows us to prevent indirect flows. For instance, the standard example of indirect flow is

$$\texttt{if}\ (\texttt{y}_{\texttt{H}})\ \{\texttt{x}_{\texttt{L}} = \texttt{0;}\}\ \texttt{else}\ \{\texttt{x}_{\texttt{L}} = \texttt{1;}\},$$

and this 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 $se(i) \leqslant \vec{k_v}(x)$, where i is the program point of the store instruction, and by requiring a global constraint on the security environment for the 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, so the transition for the store instruction cannot be typed.

3.4. *Typing rules*

Our typing rules are of the form

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

where $\vec{k_v} \longrightarrow k_r$ is a policy, $st, st' \in \mathscr{S}^\star$ 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.

Typing rules are used to establish a notion of typability. Following the ideas of 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 with respect to a given policy $\vec{k_v} \longrightarrow k_r$, a CDR structure region : $\mathscr{PP} \to \wp(\mathscr{PP})$ and a security environment $se : \mathscr{PP} \to$

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

Fig. 4. The transfer rules for the instructions in the $\mathrm{JVM}_{\mathscr{I}}$.

$\mathscr{S}$ if there exists a function $S : \mathscr{PP} \to \mathscr{S}^\star$, called a global typing, such that $S_1 = \varepsilon$ (the operand stack is empty at the initial program point 1), and for all $i, j \in \mathscr{PP}$, we have:

(1) $i \mapsto j$ implies that there exists $st \in \mathscr{S}^\star$ 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 the 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 a typable program from the view of abstract interpretation. Informally, a program P has type S if and only if 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 $\mathsf{JVM}_{\mathscr{I}}$. We use $\sqcup$ to denote the lub of two security levels, and for every $k \in \mathscr{S}$, 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 $\vec{k_a} \longrightarrow k_r$.

The following sections comment on some essential rules.

3.4.1. *The* push *n rule.* The transfer rule for a push n instruction prevents indirect flows by requiring that the value pushed onto the 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

$$\texttt{return}\ \mathrm{y_H}\ ?\ 0 : 1;$$

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

illustrates the need for this constraint:

```
      load y_H
l1 :  ifeq l2
      push 0   ⎫
      goto l3  ⎬ region(l1)
l2 :  push 1   ⎭
l3 :  return
```

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

3.4.2. *The* ifeq *rule.* 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, that is, the top of the input stack type. We need 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, nevertheless, it is accepted by a standard bytecode verifier):

```
      push 0
      push 1
      load y_H
l1 :  ifeq l2
      swap     ⎫
      pop      ⎪
      goto l3  ⎬ region(l1)
l2 :  pop      ⎭
l3 :  store x_L
```

In this example, the final value of variable x_L is equal to 0 or 1, and it reveals if the value of y_H is 0 or not. So the program is interferent. Our type system correctly 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. Then, since 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 since 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

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

of the form

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

is sufficient.

3.4.3. *The* return *rule.* The transfer rule for return requires $se(i) \leqslant k_r$, which prevents return instructions under the guard of expressions with a security level greater than k_r. In addition, the rule requires that the value on the top of the operand stack has a security level $\leqslant k_r$ since it will be observed by the attacker at level k_r. The following example, which corresponds to the source program

$$\texttt{if } (\texttt{y}_\texttt{H})\ \{\texttt{return 0;}\}\ \texttt{else}\ \{\texttt{return 1;}\},$$

illustrates why we need to prevent return instructions from appearing in high regions:

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

This program is interferent with respect to 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 correctly 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. If P is a typable JVM$_{\mathcal{I}}$ program with respect to a safe CDR (region, jun) 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 $\leadsto$ 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 us to prove the step-consistent unwinding lemmas below, indistinguishability between states must encompass states that have operand stacks of different length.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

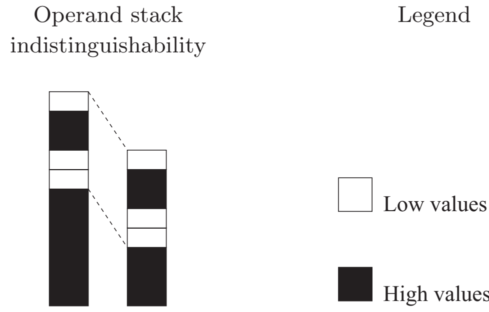

Fig. 5. Operand stack indistinguishability.

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 $st[i] \not\leqslant k_{\mathrm{obs}}$ for every $1 \leqslant i \leqslant n$.

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:

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

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

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

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

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

Definition 3.8 (state indistinguishability). Two states $\langle i, \rho, os \rangle$ and $\langle i', \rho', os' \rangle$ are indistinguishable with respect to $st, st' \in \mathcal{S}^\star$, denoted $\langle i, \rho, os \rangle : st \sim \langle i', \rho', os' \rangle : st'$, if and only if $os : st \sim os' : st'$ and $\rho \sim \rho'$ hold.

We say that the security environment se is high in region $\mathsf{region}(i)$ if $se(j) \not\leqslant k_{\mathrm{obs}}$ 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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 a 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 $se(\mathsf{pc}(s_1)) \not\leqslant k_{obs}$, 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.

In order to apply unwinding lemmas repeatedly, we need a family of results that deals with the 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) are high and se is high in $\mathsf{region}(i)$.
— *high step:*
If $s \rightsquigarrow s'$, and $\mathsf{pc}(s) \vdash st \Rightarrow st'$, and $se(\mathsf{pc}(s)) \not\leqslant k_{obs}$, 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 the 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 proofs make 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 that 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 (the induction hypothesis) and suppose $n > 0$ and $m > 0$. We write $i_0 = \mathsf{pc}(s_0) = \mathsf{pc}(s'_0)$. We first note

[†] The base cases depend on technical properties about return points, which we omit here.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 $i_0 \vdash S_{i_0} \Rightarrow st$, $st \sqsubseteq S_{\mathsf{pc}(s_1)}$, $i_0 \vdash S_{i_0} \Rightarrow st'$, $st' \sqsubseteq S_{\mathsf{pc}(s'_1)}$. Then:

— 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 the 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. From the *high stack type sub-typing* lemma, this 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 $\mathsf{pc}(s_1) = \mathsf{jun}(i_0)$. It is now easy to show by induction on the trace $s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$ that either there exists k, $1 \leqslant k \leqslant n$ 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 using CDR2 and the *high step* and *indistinguishability single monotonicity* lemmas. Note that in the second case, where $\mathsf{pc}(s_n) \in \mathsf{region}(i_0)$, we necessarily have $\mathsf{jun}(i_0)$ undefined by CDR3. A similar property holds for trace $s'_1 \rightsquigarrow \cdots \rightsquigarrow s'_m$ and we can group the different cases into two main cases:

(1) $\mathsf{jun}(i_0)$ is defined and there exists k, k', $1 \leqslant k \leqslant n$ and $1 \leqslant k' \leqslant m$ 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 the transitivity and symmetry of $\sim$, that

$$s_k : S_{\mathsf{pc}(s_k)} \sim s'_{k'} : S_{\mathsf{pc}(s'_{k'})}$$

with $\mathsf{pc}(s_k) = \mathsf{pc}(s'_{k'})$, and we can conclude by the 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 must ensure that the CDR information complies with the CDR properties – we have shown that these properties are sufficient to guarantee the soundness of the type system. The CDR properties can be checked naively in cubic time, and under specific hypotheses, more effective verification methods can be designed.

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 $i \mapsto^+ j$, that is, 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. We then 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 $\mathsf{jun}(i)$, that is, $j \in \mathsf{region}(i)$ if and only if $i \mapsto^+ j$ and $\mathsf{jun}(i)$ post-dominates j – in particular, $\mathsf{jun}(i)$ is defined; otherwise,

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

$j \in \mathsf{region}(i)$ if and only if $i \mapsto^+ j$. Note that, under this construction, $i \in \mathsf{region}(i)$ entails $i \mapsto^+ i$.

3.7. *Verifying typability*

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

The following are a few practical issues connected with the use of our lightweight information flow checker in a Proof Carrying Code scenario:

(1) 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 reducing the programmer's burden is typically done at source level.
(2) The security environment, program type and control dependence regions can be generated automatically by a certifying compiler, so the process of generating an annotated JVM program that is processable by our checker from a typable Java program can be automated.
(3) 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. $\mathrm{JVM}_{\mathscr{O}}$: the object-oriented extension of $\mathrm{JVM}_{\mathscr{I}}$

The object-oriented extension of the $\mathrm{JVM}_{\mathscr{I}}$ is called the $\mathrm{JVM}_{\mathscr{O}}$ and includes arrays, instance fields, the creation of new instances and null pointers. However, it does not include methods, which will be added in Section 5.

4.1. *Programs, memory model and operational semantics*

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

Programs use the extended set of instructions given in Figure 6.

4.1.2. *States.* The set of $\mathrm{JVM}_{\mathscr{O}}$ values is extended to $\mathscr{V} = \mathbb{Z} \cup \mathscr{L} \cup \{null\}$, where $\mathscr{L}$ 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 the requirement that they do not perform pointer arithmetic in $\mathrm{JVM}_{\mathscr{O}}$.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

$$
\begin{array}{lll}
instr ::= \ldots & & \\
\mid & \mathsf{new}\ C & \text{create new object in the heap} \\
\mid & \mathsf{getfield}\ f & \text{load value of field } f \text{ on stack} \\
\mid & \mathsf{putfield}\ f & \text{store top of stack in field } f \\
\mid & \mathsf{newarray}\ t & \text{create new array of element of type } t \text{ in the heap} \\
\mid & \mathsf{arraylength} & \text{get the length of an array} \\
\mid & \mathsf{arrayload} & \text{load value from an array} \\
\mid & \mathsf{arraystore} & \text{store value in array}
\end{array}
$$

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

A $\text{JVM}_{\mathcal{O}}$ state is now of the form $\langle i, \rho, os, h \rangle$, where i, ρ and os are defined as in $\text{JVM}_{\mathcal{I}}$ and h is a heap, which 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 modelled as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$ (that is, each object $o \in \mathcal{O}$ possesses a class, denoted by $\mathsf{class}(o)$, and a partial function to access field values) and the set $\mathcal{A}$ of arrays is modelled as $\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}) \times \mathcal{PP}$ (that is, each array $a \in \mathcal{A}$ possesses a length, denoted by $a.\mathsf{length}$, a partial function to access array values and a creation point). The creation point information is not mandatory for specifying program behaviours, but will be useful when defining array indistinguishability. We write $o.f$ for the access to the value of field f, and use $o \oplus \{f \mapsto v\}$ to denote the update of an object o at field f with a value v (we use $h \oplus \{l \mapsto o\}$ in the same way for heap update) and Heap is the set of heaps.

4.1.3. *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. The function $\mathsf{default}$ is specified according to the standard Java convention[†]: for all defined fields $f \in \mathcal{F}$ of a class $C \in \mathcal{C}$,

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

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

$$
\leadsto \subseteq \mathsf{State}_{\mathcal{O}} \times (\mathsf{State}_{\mathcal{O}} + (\mathcal{V} \times \mathsf{Heap})).
$$

In words:

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

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

- putfield f uses the top of the stack to update the object associated with the location under the top of the operand stack.
- return now returns the top of the operand stack, and the current heap.
- 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 creation point i.
- arraylength pops a location l from the operand stack and pushes the length of the corresponding array.
- 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.
- arraystore stores the top of the stack into the jth element of an array a, where j and a are determined by the second and third values, respectively, in the stack.

4.1.4. *Successor relation.* The successor relation is extended by adding 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 (that is, objects that have been created in a high security environment) cannot be distinguished by an attacker, so indistinguishability is defined relative to a bijection β on (a partial set

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 by adding 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 itself be 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 only be updated in a high context (its content is high) but can be allocated in any context (its length and the value of its reference are low). We use $\mathcal{S}^{\text{ext}}$ to denote the extension of security levels $\mathcal{S}$ to include array security levels. The partial order on $\mathcal{S}$ is extended to $\leqslant^{\text{ext}}$ with the following inductive definition:

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

An extended level $k[k_c]$ is considered to be low (written $k[k_c] \leqslant k_{\text{obs}}$) if $k \leqslant k_{\text{obs}}$. 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 with respect to the partial order on $\mathcal{S}$ only. Apart from k_{obs} and the elements of the security environments, all 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:

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

The definitions of both 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 the creation points of arrays to security levels for their contents. The mapping at will be left implicit in the rest of the paper. We will abusively use $\mathsf{at}(a)$ to denote the level 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 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

$$a_1.\mathsf{length} = a_2.\mathsf{length},$$

and, moreover, if $\mathsf{at}(a_1) \leqslant k_{\text{obs}}$, then for any index i such that $0 \leqslant i < a_1.\mathsf{length}$, we have $a_1[i] \sim_\beta a_2[i]$.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 $\mathsf{ft}(f) \leqslant k_{\mathrm{obs}}$, we have $o_1.f \sim_\beta o_2.f$.

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* (that is, locations that might be reachable from low local variables/fields) of the heaps under consideration.

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 $h_1 \sim_{k_{\mathrm{obs}},\beta} h_2$, 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 $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim_{k_{\mathrm{obs}},\beta} h_2(\beta(l))$ 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 the 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* with respect to 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 $\langle 1, \rho_1, \epsilon, h_1 \rangle \leadsto^+ v_1, h_1'$, $\langle 1, \rho_2, \epsilon, h_2 \rangle \leadsto^+ v_2, h_2'$ 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 $k_r \leqslant k_{\mathrm{obs}}$, then $v_1 \sim_{\beta'} v_2$.

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

4.3. *Typing rules*

The abstract transition system of the JVM$_{\mathcal{O}}$ extends that of the JVM$_{\mathcal{I}}$ by adding the typing transfer rules of Figure 8. The typing rules we propose for arrays follow Askarov and Sabelfeld (2005), which argues that public arrays must be allowed to handle secret information if they are going to be usable for realistic case studies such as the mental poker they have programmed in Jif (Myers 1999).

The following sections comment on the rules.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

4.3.1. *The* new *rule.* 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, 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 be stored in either a high or low variable/field.

4.3.2. *The* putfield *rule.* The transfer rule for putfield requires that $k_1 \leqslant \mathsf{ft}(f)$ (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 $se(i) \leqslant \mathsf{ft}(f)$ prevents an implicit flow caused by an assignment to a low field in a high security environment. Finally, the constraint $k_2 \leqslant \mathsf{ft}(f)$ prevents the modification of low fields of high objects that may be aliases to low objects.

To illustrate the last point, consider the source program

```
C x_L  =  new C();
z_H  =  y_H ? new C()  :  x_L;
z_H.f_L  =  1;
```

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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 be high variables. The bytecode for this program is

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

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 because of the putfield rule, which prevents this type of leak caused by aliasing (with the constraint $k_2 \leqslant \mathsf{ft}(f)$ preventing assignments to low fields from high target objects).

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

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

4.3.5. *The* arraylength *rule.* 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.

4.3.6. *The* arrayload *rule.* The transfer rule for arrayload pushes a security level

$$(k_1 \sqcup k_2) \sqcup^{\mathrm{ext}} k_c$$

onto the top of the stack. The join operation

$$\sqcup^{\mathrm{ext}} \in \mathscr{S} \times \mathscr{S}^{\mathrm{ext}} \to \mathscr{S}^{\mathrm{ext}}$$

is defined by

$$k' \sqcup^{\mathrm{ext}} k = k' \sqcup k$$

when $k, k' \in \mathscr{S}$ and

$$k' \sqcup^{\mathrm{ext}} k[k_c] = (k' \sqcup k)[k_c]$$

when $k, k' \in \mathscr{S}$ and $k_c \in \mathscr{S}^{\mathrm{ext}}$. Here k_1 allows us to prevent implicit flows through a high index and k_2 through an alias.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

The following example, which corresponds to a source program like

```
int x_L = a_L[L][i_H];
```

illustrates the first point:

load $a_{L[L]}$
load i_H
arrayload
store x_L

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

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 $\mathtt{a_{L[L]}[i_H]}$. This program is rejected by our type system because $\mathtt{a_{L[L]}[i_H]}$ receives a type H in the arrayload rule, and storing a high value in a low variable is impossible because of the store rule.

The second point corresponds to an access $\mathtt{a_{H[L]}[i_L]}$, where $\mathtt{a_{H[L]}}$ may either be aliased to a low array $\mathtt{a^0_{L[L]}}$ containing only the 0 integer or aliased to a low array $\mathtt{a^1_{L[L]}}$ containing only the integer 1. This means that observing the value of $\mathtt{a_{H[L]}[i_L]}$ would allow an attacker to know which of these arrays is aliased to $\mathtt{a_{H[L]}}$.

4.3.7. *The* arraystore *rule.* The transfer rule for arraystore uses the partial order $\leqslant^{\mathrm{ext}}$ 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 also constrains k_2 and k_c to prevent an information leak by updating a low array content with a high index. Without it, an assignment of the form $\mathtt{a_{L[L]}[i_H]} = \mathtt{1}$ in a low array $\mathtt{a_{L[L]}}$ only containing the integer 0 would reveal the value of $\mathtt{i_H}$.

Finally, the constraint between k_3 and k_c prevents the modification of low array contents if its reference is high. This is, for example, required if an array $\mathtt{a_{H[L]}}$ may be aliased to two distinct low array $\mathtt{a^0_{L[L]}}$ and $\mathtt{a^1_{L[L]}}$. Observing which of these low arrays is modified as a side effect of the assignment $\mathtt{a_{H[L]}[i_L]} = \mathtt{v_L}$ would allow an attacker to learn which of these arrays is effectively equal to $\mathtt{a_{H[L]}}$.

4.4. *Type system soundness*

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

Theorem 4.6. Let P be a typable JVM$_{\mathscr{O}}$ program with respect to the safe CDR (region, 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. JVM$_{\mathscr{C}}$: the method extension of JVM$_{\mathscr{O}}$

In this section we extend our analysis to include methods. The extension is compatible with bytecode verification in the sense that the analysis is modular.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

$$\frac{P_m[i] \neq \text{invokevirtual } m_{\text{ID}} \qquad \langle i, \rho, os, h\rangle \leadsto_{\text{JVM}_{\mathcal{O}}} \langle i', \rho', os', h'\rangle}{\langle i, \rho, os, h\rangle \leadsto_m \langle i', \rho', os', h'\rangle}$$

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

Fig. 9. Operational Semantics for the $\text{JVM}_{\mathscr{C}}$.

5.1. *Programs, memory model and operational semantics*

5.1.1. *Programs.* Each program comes equipped with a set $\mathscr{M}$ of method names together with a set $\mathscr{C}$ of classes as in the $\text{JVM}_{\mathcal{O}}$. 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 require that all methods return a value. The set of instructions of the $\text{JVM}_{\mathcal{O}}$ is extended by adding the new instruction invokevirtual m_{ID} for calling a virtual method. Here, m_{ID} is a method identifier, which may correspond to several methods in the class hierarchy according to the 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.

5.1.2. *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 $\text{JVM}_{\mathscr{C}}$ state is defined in the same way as for the $\text{JVM}_{\mathcal{O}}$.

5.1.3. *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 can be seen in the first rule, the semantics of instructions is like that for the $\text{JVM}_{\mathcal{O}}$, except for the new instruction invokevirtual, whose semantics is given by the second rule. The location l is used to resolve the virtual call. Given the class of l and the identifier m_{ID}, a method m' is found in the class hierarchy (through the lookup operator). The transitive closure of $\leadsto_m$ is then used to obtain the result of the execution of m'. The 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[†].

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

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

5.2. *Non-interference*

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

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 (whether 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 the analysis modular – it represents a lower bound for the security levels of fields that are affected during the execution of the method.

A method is only allowed to perform field updates on fields whose level is greater than k_h. We will now formally define this notion of a *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 $k \in \mathscr{S}$ (written as $h_1 \preceq_k h_2$) if and only if $\mathsf{dom}(h_1) \subseteq \mathsf{dom}(h_2)$ and for all locations $l \in \mathsf{dom}(h_1)$ and all fields $f \in \mathscr{F}$ such that $k \not\leqslant \mathsf{ft}(f)$, we have $h_1(l).f = h_2(l).f$.

This enables us to define the notion of a *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 $\rho \in \mathscr{X} \rightharpoonup \mathscr{V}$, all heaps $h, h' \in \mathsf{Heap}$ and values $v \in \mathscr{V}$, we have $\langle 1, \rho, \epsilon, h\rangle \rightsquigarrow_m^+ v, h'$ implies $h \preceq_{k_h} h'$.

The notion of a non-interferent method can be stated using the same indistinguishability relation as for the $\mathrm{JVM}_{\mathscr{O}}$. A method m is said to be *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 $\mathrm{JVM}_{\mathscr{C}}$ method). A method m is *non-interferent* with respect to a policy $\vec{k_v}\longrightarrow k_r$ if for every attacker level k_{obs} and every partial function $\beta \in \mathscr{L} \rightharpoonup \mathscr{L}$ and every $\rho_1, \rho_2 \in \mathscr{X} \rightharpoonup \mathscr{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $v_1, v_2 \in \mathscr{V}$ 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 $\beta' \in \mathscr{L} \rightharpoonup \mathscr{L}$ such that $\beta \subseteq \beta'$, $h_1' \sim_{k_{\mathrm{obs}},\beta'} h_2'$ and, moreover, if $k_r \leqslant k_{\mathrm{obs}}$, then $v_1 \sim_{\beta'} v_2$.

A method is secure if it is side-effect safe and non-interferent.

Definition 5.4 (secure $\mathrm{JVM}_{\mathscr{C}}$ method). A method m is *secure* with respect to 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 $\vec{k_v}\longrightarrow k_r$.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

Let Γ be a table of method signatures. This table associates with 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 an object of level k (as in Banerjee and Naumann (2005) for source programs). The set of security signatures of a method m is defined by

$$\mathrm{Policies}_{\Gamma}(\mathrm{m}) = \{\Gamma_{\mathrm{m}}[\mathrm{k}] \mid \mathrm{k} \in \mathcal{S}\}. \tag{1}$$

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. However, we will use it to define the notion of a *secure program.*

Definition 5.5 (secure $\mathrm{JVM}_{\mathscr{C}}$ **program).** A program is *secure* with respect to a table of method signatures Γ if for all its methods m, we have 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.

If the code of m is defined by

```
      new C
      store y_L
      load x_H
      ifeq l_1
      load y_L
      push 1
      putfield f
l_1 : load y_L
      getfield f
      return
```

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

[†] Associating signatures with method identifiers instead of methods allows us to enforce the requirement that method overriding preserves the declared security signatures.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

Fig. 10. The new transfer rules for the instructions of the $\mathrm{JVM}_{\mathscr{C}}$.

For putfield, only one constraint is added comapred with the previous $\mathrm{JVM}_{\mathscr{O}}$ rule. The new constraint $k_h \leqslant \mathsf{ft}(f)$ restricts the 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 $k \leqslant k'_h$ is to avoid invocation of methods with low effect on the heap with high target objects. Two different target objects (in two executions) may mean that the body of the method to be executed is different in each execution. If the effect of the method is low ($k_h \leqslant k_{\mathrm{obs}}$), then low memory is modified differently in each execution, leading to an information leak. The constraint $se(i) \leqslant k'_h$ prevents implicit flows (low assignment in high regions) during the execution of the called method. The constraint $k_h \leqslant k'_h$ 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 $(k'_r \sqcup k \sqcup se(i))$, which is obtained from the signature of m_{ID}, prevents its result from flowing to variables or fields with a lower security level. The security level k prevents flows due to the execution of two distinct methods.

The following example illustrates how object-oriented features can lead to interference – see 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 method m be declared in C and overridden in D, as illustrated by the following source program[†]:

```
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();}
}
```

[†] We omit the call of the initialiser.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

```
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 the high variable y_H. Information about y_H may then be inferred by observing the return value of method m.

Finally, we define the typability of programs.

Definition 5.8 (typable $\mathrm{JVM}_{\mathscr{C}}$ program). A $\mathrm{JVM}_{\mathscr{C}}$ program is typable with respect to safe CDRs $(\mathsf{region}_m, \mathsf{jun}_m)$ and with a table of signatures Γ if and only if all methods m in P are typable with respect to $(\mathsf{region}_m, \mathsf{jun}_m)$ and all signatures in $\mathrm{Policies}_\Gamma(\mathrm{m})$.

5.4. *Type system soundness*

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

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

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

In this section we show how the $\mathsf{JVM}_{\mathscr{C}}$ can be 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 notions of both state indistinguishability and 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 change the control flow graph from being an unlabelled 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 to include the labels, that is, we need to use $\mathsf{region}(i, C)$ 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, so 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 the targets of throw 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 include 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 and exception analyses are in the Trusted Computing Base. Thus, the type system exploits the information of the class analysis and signature of methods (which 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*

6.1.1. *Programs.* The instruction set of the $\mathrm{JVM}_{\mathscr{C}}$ is extended by adding the bytecode throw. We assume that programs come equipped with a partial function $\mathsf{Handler}_m : \mathscr{PP} \times \mathscr{C} \rightharpoonup \mathscr{PP}$, which for each method m selects the appropriate handler for a given program point. If an exception of class $C \in \mathscr{C}$ is thrown at program point $i \in \mathscr{PP}$, then, if $\mathsf{Handler}_m(i, C) = t$, the control will be transferred to program point t, and if $\mathsf{Handler}_m(i, C)$ is undefined (written $\mathsf{Handler}_m(i, C) \uparrow$), 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.

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

6.1.3. *Operational semantics.* Figure 11 shows the semantics of some exception throwing instructions in the $\mathrm{JVM}_{\mathscr{E}}$ (for brevity, we have not included the exception rules for getfield, putfield, arraylength, arrayload and arraystore). There are three exception rules for the virtual call instruction. The first and second model the cases when execution of the called method terminates through 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 the requirement 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 with the null pointer exception. When a native exception **np** is thrown, the catching mechanism is modelled by the function RuntimeExcHandling. Each instruction performing an access on a reference (getfield f, putfield f and throw, arraylength, arrayload, arraystore)

[†] This hypothesis is put directly as a 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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

Fig. 11. New operational semantics rules for the $\mathrm{JVM}_{\mathscr{E}}$.

has a similar semantics. The last two rules deal with the new instruction throw, which throws the exception pointed to by the reference on top of the stack. Transitions are now parametrised by a tag $\tau \in \{\mathsf{Norm}\} + \mathscr{C}$ to describe the nature of the transition (see the successor relation below). We will sometimes omit the tag τ in the notation $\leadsto_{m,\tau}$ for clarity.

6.1.4. *Successor relation.* The successor relation is now decorated by an element (called a *tag*) in $\{\mathsf{Norm}\} + \mathscr{C}$ in order to reflect the nature of the underlying semantics step: Norm for a normal step (as in the $\mathrm{JVM}_{\mathscr{C}}$) and $c \in \mathscr{C}$ 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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

6.1.5. *CDR properties.* The CDR results are now associated not only with program points but also with tags:

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

We call a point i such that there exists $\tau \in \{\mathsf{Norm}\} + \mathscr{C}$ with $i \mapsto^{\tau}$ a *return point*. 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, we have $\mathsf{jun}(i, \tau') \in \mathsf{region}(i, \tau)$.

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

Junction points uniquely delimit the ends of regions.

— CDR1 expresses the requirement that successors of branching points belong to (or end) 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 its end.
— CDR3 forbids junction points for a region that contains a return point.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

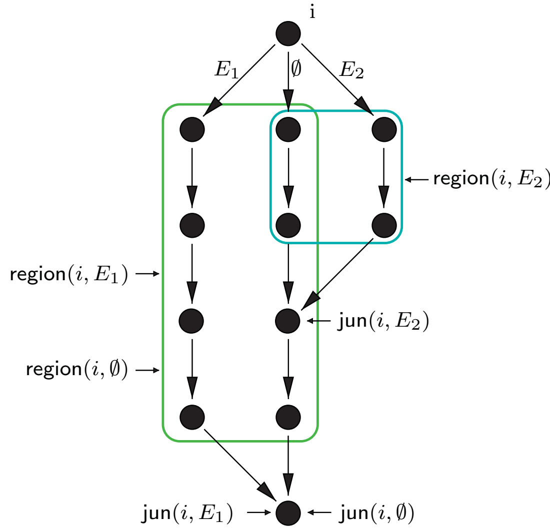

Fig. 13. (Colour online) Example of a CDR in the $\mathrm{JVM}_{\mathscr{E}}$. Only relevant tags are presented here.

— CDR4 and CDR5 express properties between regions of the 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 the requirement that the junction point of a region must be within every region that contains a return point and is decorated with a different tag.

— CDR6 imposes the requirement 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 a safe CDR for an abstract transition system.

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 $\mathsf{JVM}_{\mathscr{C}}$ (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 the *output level*) is now a list of security levels of the form $\{\mathsf{Norm} : k_n, e_1 : k_{e_1}, \dots e_n : k_{e_n}\}$, 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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \qquad (\mathsf{class}(h_1(l_1)) : k) \in \vec{k_r} \qquad k \nleqslant k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \qquad (\mathsf{class}(h_2(l_2)) : k) \in \vec{k_r} \qquad k \nleqslant k_{\text{obs}}}{(v_1, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{\begin{array}{c} h_1 \sim_{k_{\text{obs}},\beta} h_2 \qquad (\mathsf{class}(h_1(l_1)) : k_1) \in \vec{k_r} \qquad (\mathsf{class}(h_2(l_2)) : k_2) \in \vec{k_r} \\ k_1 \nleqslant k_{\text{obs}} \qquad k_2 \nleqslant k_{\text{obs}} \end{array}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

In each case, heaps must be indistinguishable. This definition implies that if indistinguishability outputs are of a different nature (such as a normal value and an 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 a similar nature (two normal values or two exceptions of the same class) they are indistinguishable if the corresponding security level in $\vec{k_r}$ is low.

Both the previous definition and the following one for non-interference rely on indistinguishability definitions already given for the JVM$_{\mathscr{O}}$ (see Definition 4.1).

Definition 6.2 (non-interferent JVM$_{\mathscr{E}}$ method). A method m is *non-interferent* with respect to a policy $\vec{k_v} \longrightarrow \vec{k_r}$ if for every attacker level k_{obs}, every partial function $\beta \in \mathscr{L} \rightharpoonup \mathscr{L}$ and every $\rho_1, \rho_2 \in \mathscr{X} \rightharpoonup \mathscr{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $r_1, r_2 \in \mathscr{V} + \mathscr{L}$ such that

$$\begin{aligned} \langle 1, \rho_1, \epsilon, h_1 \rangle &\leadsto_m^+ r_1, h_1' \\ \langle 1, \rho_2, \epsilon, h_2 \rangle &\leadsto_m^+ r_2, h_2' \\ h_1 &\sim_{k_{\text{obs}},\beta} h_2 \\ \rho_1 &\sim_{\vec{k_v},k_{\text{obs}},\beta} \rho_2, \end{aligned}$$

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

$$(r_1, h_1') \sim_{k_{\text{obs}},\beta',\vec{k_r}} (r_2, h_2').$$

As in the JVM$_{\mathscr{C}}$, we impose a *side-effect-safe* condition (see Definition 5.2 for a formal definition) on methods. This notion is used when a virtual call occurs in a high context in

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

order to enforce the requirement that no low information is modified during the execution of the called method.

Definition 6.3 (secure JVM$_{\mathscr{E}}$ method). A method m is *secure* with respect to 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$_{\mathscr{E}}$ program). A program is *secure* with respect to a table of method signatures Γ if for all its methods m, we have m is secure with respect to all policies in Policies$_\Gamma$(m) – see Equation (1) in Section 5.2 for the definition of Policies$_\Gamma$(m).

6.3. *Typing rules*

The typing rules for the JVM$_{\mathscr{C}}$ are extended (and modified in the case of ifeq and invokevirtual) with the rules given in Figure 14. These rules are concerned with exception-throwing and branching instructions only; the rules for the other instructions are as in the JVM$_{\mathscr{C}}$.

The rule for ifeq is updated to flag with Norm the region that it is concerned with. The virtual call now needs three typing rules. The first corresponds to a normal control flow edge from the call site to its successor in the calling method, and is very similar to the rule in the JVM$_{\mathscr{C}}$, 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}) 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 (which 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 $\mathsf{region}(i, e)$.

Note that the typing judgement is now parametrised by a tag $\tau \in \{\mathsf{Norm}\} + \mathscr{C}$, which 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 a tag requires us to update the notion of a *typable method.*

Definition 6.5 (typable method). A method m is typable with respect to a method signature table Γ, a global field policy ft, a policy sgn and a CDR $\mathsf{region}_m : \mathscr{PP} \to \wp(\mathscr{PP})$ if there exists a security environment $se : \mathscr{PP} \to \mathscr{S}$ and a function $S : \mathscr{PP} \to \mathscr{S}^\star$ such that $S_1 = \varepsilon$ and for all $i, j \in \mathscr{PP}$, we have $e \in \{\mathsf{Norm}\} + \mathscr{C}$:

(1) $i \mapsto^e j$ implies there exists $st \in \mathscr{S}^\star$ 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 $\Gamma, \mathsf{ft}, \mathsf{region}, se, sgn, i \vdash^e S_i \Rightarrow$

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

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

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

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

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

Fig. 14. The transfer rules for the instructions of the $\mathrm{JVM}_{\mathscr{E}}$.

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

$$m : (this : L,\ x : L,\ y : H) \xrightarrow{H} \{\mathsf{Norm} : H,\ C : L,\ \mathbf{np} : H\},$$

with the CDR (given only for branching points), the type stacks and the security environment shown in Figure 15.

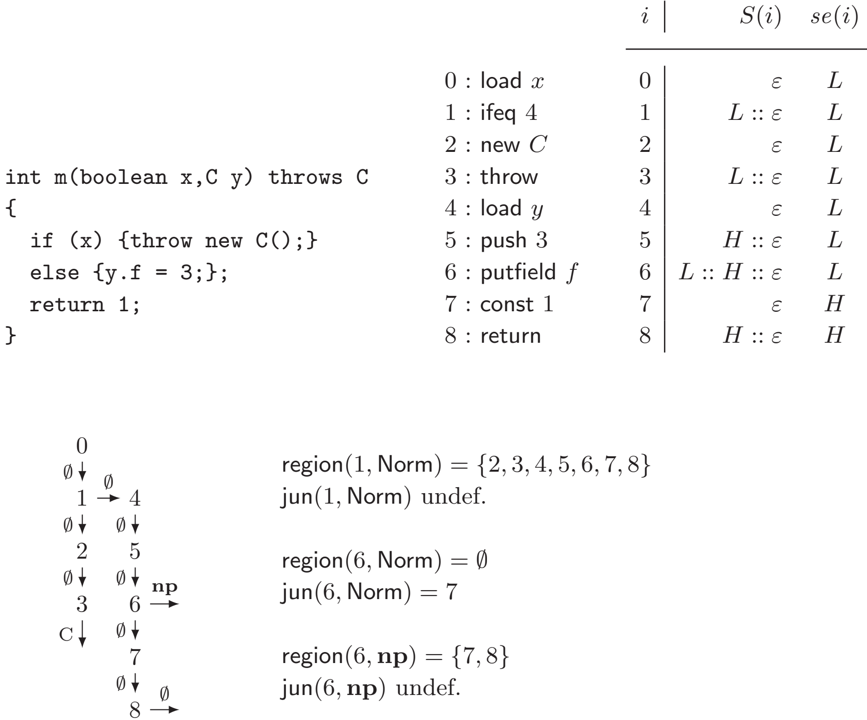

Fig. 15. Typable methods at source and bytecode level.

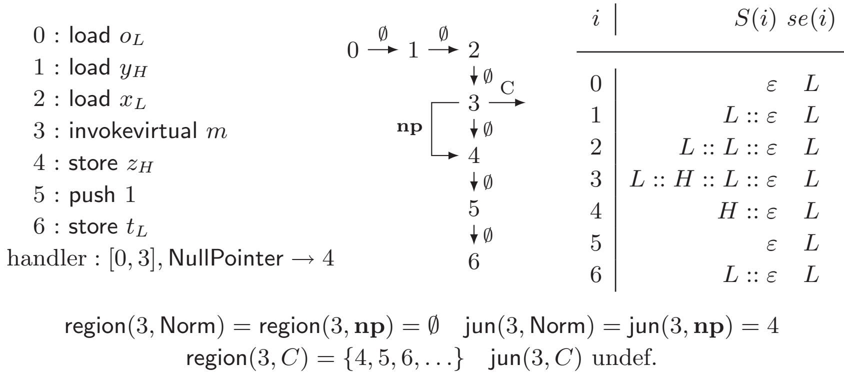

Fig. 16. Typable fragment with virtual call.

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 save space, we have only included a compressed version of the compiled code.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

regions, the typing rule for the 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 conclude this section by stating the type soundness theorem for the $\mathrm{JVM}_{\mathscr{E}}$.

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

7. Machine-checked proof

We have formalised within the Coq proof assistant the information flow type system for the $\mathrm{JVM}_{\mathscr{E}}$ fragment, and proved its soundness formally. Moreover, we have formalised executable checkers for the CDR properties and for typability, and proved their soundness formally (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 with respect to our type system and non-interferent). The statement of soundness of the type system hinges on a formalisation of the operational semantics of the JVM, and of the notion of a non-interferent program.

This section presents an overview of the proof. We start with a short discussion of the relevance of formal proofs, and some statistics related to the development. Following that, we describe our formalisation of the semantics of the JVM and of the notion of a non-interferent program. We then discuss our approach to proving unwinding lemmas and constructing executable checkers. Finally, we conclude with an example.

7.1. *Motivation and overview*

Our formalisation is a contribution to an increasing trend in the use of proof assistants to build 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 in managing the complexity of type soundness proofs. In our view, the complexity of information flow type systems for full-fledged languages makes machine-checked formalisations extremely important, if not compulsory, for three reasons:

(1) The formalisation of the operational semantics contains a significant number of rules; for example, the JVM virtual call has five different transitions (a call on a null reference, which generates a null pointer exception that may be caught or not, normal termination of the callee, and termination by an exception that may or may not be caught in the caller context).
(2) The type system contains over 60 rules, and many rules have a large (up to 10) number of premises – see, for example, Figure 14.
(3) The proof of non-interference relies on unwinding lemmas, which require reasoning about two program executions, which lead to a very large number of cases in proofs. Moreover, the proof of correctness of the type system is stratified: we must first prove

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

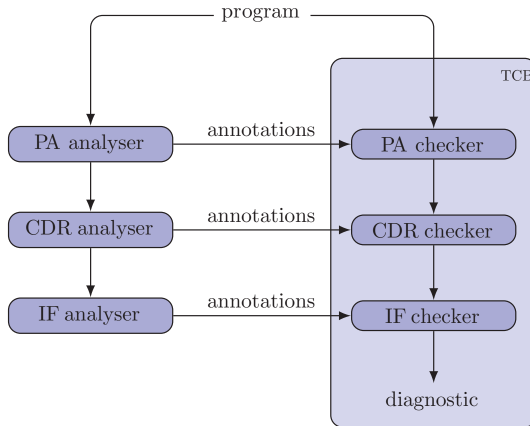

Fig. 17. (Colour online) Information flow analyser and checker.

that the CDR checker is correct (assuming that the pre-annotations are), and then prove that the type system is correct (assuming that the pre-annotations and the CDR checker are – we do not currently provide a means of checking the correctness of pre-annotations; this is left for future work).

Summarising, we have proved the following theorem (the second part corresponds to 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 formalised by the predicate SAFE;
— the correctness of program annotations is formalised by the predicate PA;
— the CDR properties (given in Section 6) are formalised by the predicate CDR;
— the notion of a typable program is formalised by the IF predicate.

Foundational Proof-Carrying Code (Appel 2001) provides another motivation for carrying out 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 analysers. Our formalisation 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 analyser are correct.
(2) The CDR checker verifies that regions provided by the CDR analyser verify the safe over-approximation properties of Section 6.
(3) The IF checker verifies type correctness in the style of lightweight bytecode verification.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

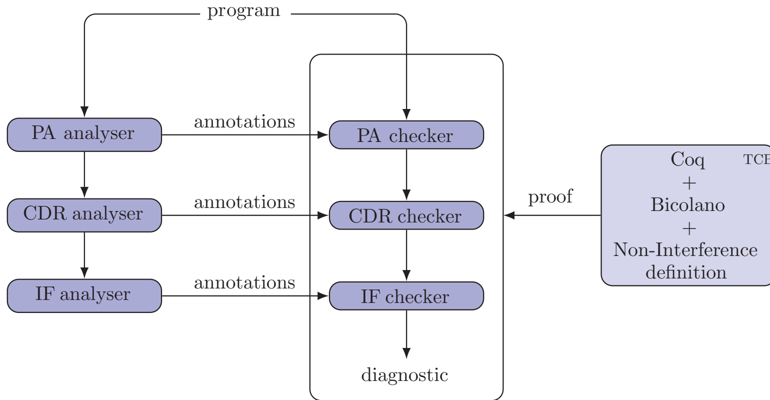

Fig. 18. (Colour online) Information flow analyser and checker with Coq TCB.

	Lines 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.

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 this 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 these is a significant formalisation in itself – Figure 19 gives an indication of the sizes of the components. The development is available at

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

7.2. *Formal semantics*

The development relies on a formal semantics of the JVM in Coq, which is called Bicolano and was developed within the Mobius project to serve as a common basis for the certification of proof carrying code technologies in Coq. Bicolano closely follows the official JVM specification – although some features such as initialisation, subroutines,

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

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

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

multi-threading, dynamic class loading, garbage collection, 64-bit arithmetic and floats are omitted.

The core of Bicolano is a small-step operational semantics that describes the dynamic behaviour of a bytecode program according to the JVM specification. The small-step semantics is formalised 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 formalises 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 is shown in Figure 9 in Section 5.1.3. The mix-step semantics is also formalised as an inductively defined relation $\cdot \to \cdot$ between states of the virtual machine, but using 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:

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

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

The two semantics play different roles in our work. The mix-step semantics brings important simplifications in the definition of state indistinguishability and in the soundness proofs, so we use it to machine-check type soundness. The small-step semantics serves as a reference formalisation, so the final theorem is stated using the small-step semantics.

For the purposes 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 an instrumented transition. Annotations take the form of safe flags attached to program points where the pre-analyser predicts that no exception may be thrown. Moreover, the instrumented semantics imposes constraints that mean exceptions can only be raised at program points that are not annotated as safe. Assuming that the annotations are correct, the mix-step semantics and the instrumented mix-step semantics coincide:

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

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

Fig. 21. Example of an annotated semantic rule.

7.3. *Formalisation of the security condition*

The definition of non-interference for programs and the proof of soundness of the type system both rely on the notion of state indistinguishability. The main issue with the formalisation of the latter is the notion of a finite bijection used to relate heaps in the two program executions. Instead of parametrising state indistinguishability by a finite bijection β from A to A, we have found it more convenient to parametrise the definition using 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 the next section for details.

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`, which is 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 out the 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 formalised the required indistinguishability relations and built a library of basic results to reason about indistinguishability – the library contains more than 100 lemmas.

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

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

Here the parameter `newArT` gives the annotation for array allocation (with 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 the requirement that the two functions satisfy some expected properties, for example, that they are bijective.

7.4. *Soundness proof methodology*

The main technical component of the soundness proof is a (mix-step) defensive semantics that keeps track of type information and partial bijections, and is particularly useful for reasoning about well-typed executions.

The defensive semantics manipulates 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 that depends on the current security level (given here by `(se pc)`).

7.5. *Executable checkers*

The first item of Theorem 7.1 is proved by formalising 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. However, the implementation of a verifier $\mathsf{check}_{\mathrm{PA}}$ that entails PA is left for future work.

The functions $\mathsf{check}_{\mathrm{CDR}}$ and $\mathsf{check}_{\mathrm{IF}}$ are executable Coq programs that have been successfully extracted into Ocaml. We have tested them on a *Tax Calculation* Java program inspired by the case study proposed in Deng and Smith (2004). The full Java

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

source program is given in Figure 22, with its information flow type annotations given in a *Jif-like* syntax and safety annotations given in comments. The program computes income taxes from an input array of taxable income 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

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

and updates the output array `out.tax[i]` with the computed tax, which is either `taxTable[lo].married` or `taxTable[lo].single`, depending on the marital status, and then increments 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.

Most of the annotations for runtime exceptions (`NP` means NullPointer, `NAS` means NegativeArraySize, `AOB` means ArrayOutofBounds) are easy to prove using a simple null pointer analysis that maintains the invariant $\texttt{this} \neq \texttt{null}$. The others require more complex arithmetic reasoning, such as 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 the best of our knowledge, no previous work has provided a sound type system for such an expressive fragment of the sequential JVM. In combination with our companion paper on the preservation of information flow types by compilation (Barthe *et al.* 2006), our results here 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 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). This extension supports objects, methods, multi-threading and dynamic thread creation, but does not include exceptions, locks or synchronisation primitives. The extension builds on the idea in Russo and Sabelfeld (2006) to constrain the behaviour of schedulers so that high branches execute uninterruptedly, thereby avoiding internal timing leaks. In our setting, the idea of a secure scheduler is modelled by making the behaviour 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 $\mathrm{JVM}_{\mathscr{I}}$ language, how to adapt our type system so that it provides support for delimited non-disclosure, which is a specific form of declassification that enables us to declassify the value of a variable at a specified program point. Technically, the main difference between Barthe *et al.* (2008) and the current paper is that the former considers local policies, that is, there is a security policy for each

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

```
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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

program point, which 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 do not foresee any difficulties 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: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 147–160.

Agat, J. (2000) *Type Based Techniques for Covert Channel Elimination and Register Allocation*, Ph.D. 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. (eds.) *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 91–102.

Appel, A. W. (2001) Foundational proof-carrying code. In: Halpern, J. (ed.) *LICS '01 Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science* 247.

Askarov, A. and Sabelfeld, A. (2005) Security-typed languages for implementation of cryptographic protocols: A case study. In: di Vimercati, S. and Syverson, P. and Gollmann, D. (eds.) Proceedings European Symposium On Research In Computer Security – ESORICS 2005. *Springer-Verlag Lecture Notes in Computer Science* **3679** 197–221.

Aydemir, B. E. *et al.* (2005) Mechanized metatheory for the masses: The PoplMark challenge. In: Hurd, J. and Melham, T. (eds.) Theorem Proving in Higher Order Logics – Proceedings 18th International Conference, TPHOLs 2005. *Springer-Verlag Lecture Notes in Computer Science* **3603** 50–65.

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. (eds.) Verification, Model Checking and Abstract Interpretation. *Springer-Verlag Lecture Notes in Computer Science* **2934** 2–15.

Barthe, G., Cavadini, S. and Rezk, T. (2008) Tractable enforcement of declassification policies. In: *Proceedings 21st IEEE Computer Security Foundations Symposium, CSF 2008* 83–97.

Barthe, G., Naumann, D. and Rezk, T. (2006) Deriving an information flow checker and certifying compiler for Java. In: *Proceedings 2006 IEEE Symposium on Security and Privacy* 230–242.

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

Barthe, G. and Rezk, T. (2005) Non-interference for a JVM-like language. In: Fähndrich, M. (ed.) *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation* 103–112.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

Barthe, G., Rezk, T., Russo, A. and Sabelfeld, A. (2010) Security of multithreaded programs by compilation. *ACM Transactions on Information and System Security* **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. (eds.) Trustworthy Global Computing: Revised Selected Papers 6th International Symposium, TGC 2011. *Springer-Verlag Lecture Notes in Computer Science* **7173** 73–78.

Bernardeschi, C. and Francesco, N. D. (2002) Combining Abstract Interpretation and Model Checking for Analysing Security Properties of Java Bytecode. In: Cortesi, A. (ed.) Verification, Model Checking and Abstract Interpretation: Revised Papers Third International Workshop, VMCAI 2002. *Springer-Verlag Lecture Notes in Computer Science* **2294** 1–15.

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. (eds.) Trustworthly Global Computing: Revised Selected Papers 5th International Symposium, TGC 2010. *Springer-Verlag Lecture Notes in Computer Science* **6084** 253–267.

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. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Revised Selected Papers Second International Workshop, CASSIS 2005. *Springer-Verlag Lecture Notes in Computer Science* **3956** 37–56.

Deng, Z. and Smith, G. (2004) Lenient array operations for practical secure information flow. In: Proceedings 17th IEEE Computer Security Foundations Workshop, 2004. *CSFW* 115–124.

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. (ed.) Verification, Model Checking and Abstract Interpretation: Proceedings 6th International Conference, VMCAI 2005. *Springer-Verlag Lecture Notes in Computer Science* **3385** 346–362.

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: *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)* 1–10.

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

Hedin, D. and Sands, D. (2006) Noninterference in the presence of non-opaque pointers. In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)* 217–229.

Hunt, S. and Sands, D. (2006) On flow-sensitive security types. In: *Proceedings POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 79–90

Kobayashi, N. and Shirane, K. (2002) Type-based information analysis for low-level languages. In: *Proceedings of the Third Asian Workshop on Programming Languages and Systems, APLAS'02* 302–316.

Leroy, X. (2002) Bytecode verification on Java smart cards. *Software – Practice and Experience* **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.

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press

Medel, R., Compagnoni, A. B. and Bonelli, E. (2005) A typed assembly language for non-interference. In: Coppo, M., Lodi, E. and Pinna, G. M. (eds.) Theoretical Computer Science: Proceedings 9th Italian Conference, ICTCS 2005. *Springer-Verlag Lecture Notes in Computer Science* **3701** 360–374.

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

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1999) From system F to typed assembly language. In: Pugh, W. (ed.) *ACM Transactions on Programming Languages and Systems (TOPLAS)* **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 *Proceedings 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages – POPL '99* 228–241. (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: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)* 190–201.

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

Rezk, T. (2006) *Verification of confidentiality policies for mobile code*, Ph.D. 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: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)* 177–189.

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. (eds.) Proceedings TAPSOFT '97: Theory and Practice of Software Development – 7th International Joint Conference CAAP/FASE. *Springer-Verlag Lecture Notes in Computer Science* **1214** 607–621.

Volpano, D. and Smith, G. (1998) Probabilistic noninterference in a concurrent language. In: *Procedings 11th IEEE Computer Security Foundations Workshop* 34–43.

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

https://doi.org/10.1017/S0960129512000850 Published online by Cambridge University Press