Non-Interference for a JVM-like Language*

(Extended Abstract)

Gilles Barthe
INRIA Sophia Antipolis, France
Gilles.Barthe@sophia.inria.fr

Tamara Rezk
INRIA Sophia Antipolis, France
Tamara.Rezk@sophia.inria.fr

ABSTRACT

We define an information flow type system for a sequential JVM-like language that includes classes, objects, and exceptions. Furthermore, we show that it enforces non-interference. Our work provides, to our best knowledge, the first analysis that has been shown to guarantee non-interference for a realistic low level language.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Miscellaneous

General Terms

Security, Languages, Verification

Keywords

Confidentiality, Type Systems, Low Level Languages

1. INTRODUCTION

The Java security architecture combines static and dynamic mechanisms to enforce innocuousness of applications; however it lacks of appropriate mechanisms to guarantee stronger properties w.r.t. confidentiality, integrity, or resource control. The purpose of this article is to introduce a mechanism for enforcing confidentiality of applications, in the form of an information flow type system for a representative fragment of the JVM, and to show that the type system guarantees non-interference, a high-level security property that guarantees the absence of illicit information flows during a program execution. In a nutshell, our notion of non-interference assumes variables to be either public (low) or secret (high), and requires that the initial values of secret variables do not influence the final values of public variables.

While information flow type systems have been thoroughly studied in the context of high-level languages, see e.g. [1] for an information flow type system for Java, there is no previous analysis that has been shown to guarantee non-interference for a realistic assembly language. In fact, existing works either prove the correctness of information flow type systems for very simple assembly languages, or define information flow type systems for large fragments of the JVM, but do not establish their correctness. In contrast, our analysis is proven correct, and encompasses some major features of the JVM: objects, exceptions, and method calls. The work builds upon known techniques, especially from [1] and [2], but solves a number of non-trivial difficulties due to the complexity of the language. Due to space constraints, it is not possible to present all difficulties and their solutions in detail, and we omit method calls, except for a brief sketch, focusing on objects and exceptions.

In addition to soundness, the main features of our analysis are its decidability and compatibility w.r.t. bytecode verification (however, our analysis does not assume that programs are accepted by the bytecode verifier, see Section 5.4), and its relative accuracy (like every static analysis, it will reject correct programs; however we have shown that Java programs accepted by the analysis in [1] are compiled into programs that are accepted by our analysis, see Section 6).

Following [17], we opt for an incremental presentation, and analyze fragments of the JVM in layers:

- $\text{JVM}_{\mathcal{I}}$ includes basic operations to manipulate operand stacks as well as conditional and unconditional jumps. The type system and its soundness proof are adapted from our earlier work [2]. One of the main difficulties is to calculate the range of branching statements (we circumvent the problem by defining our type system relative to a control dependence region analysis, with the added benefit of achieving a parametric analysis of varying precision);
- $\text{JVM}_{\mathcal{O}}$ is an object-oriented extension of $\text{JVM}_{\mathcal{I}}$ which includes features dynamic object creation, instance field accesses and updates. The main complication of this layer is to adapt the definition of non-interference so as to keep track of the security levels of the new objects created in the heap. We adopt the solution of [1], which studies information flow for a Java-like language (without exceptions), and define indistinguishability relative to a partial bijection on memory locations;
- $\text{JVM}_{\mathcal{E}}$ extends $\text{JVM}_{\mathcal{O}}$ with exceptions. Our goal here is to show how exceptions can be handled in an information flow analysis for a low-level language. Therefore, we stick to a simple setting with only one kind of exceptions, and show how a sound type system that accommodates exceptions can be obtained simply by

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

TLDI'05, January 10, 2005, Long Beach, California, USA.

instr	::=	prim *op*	primitive operation
	|	push v	push value on top of stack
	|	pop	pop value from top of stack
	|	load x	load value of x on stack
	|	store x	store top of stack in x
	|	ifeq j	conditional jump
	|	goto j	unconditional jump
	|	return	return

where *op* is + or ×, $v \in \mathcal{V}$, $x \in \mathcal{X}$, and $j \in \mathbb{N}$.

Figure 1: INSTRUCTION SET FOR $\text{JVM}_\mathcal{I}$

$$\frac{P[i] = \text{prim } op \qquad op \in \mathbb{O} \qquad n_1 \; \underline{op} \; n_2 = n}{\langle i, \rho, n_1 :: n_2 :: s\rangle \rightsquigarrow \langle i+1, \rho, n :: s\rangle}$$

$$\frac{P[i] = \text{pop}}{\langle i, \rho, v :: s\rangle \rightsquigarrow \langle i+1, \rho, s\rangle}$$

$$\frac{P[i] = \text{load } x}{\langle i, \rho, s\rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: s\rangle}$$

$$\frac{P[i] = \text{store } x}{\langle i, \rho, v :: s\rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, s\rangle} \qquad \frac{P[i] = \text{goto } j}{\langle i, \rho, s\rangle \rightsquigarrow \langle j, \rho, s\rangle}$$

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

$$\frac{P[i] = \text{push } n}{\langle i, \rho, s\rangle \rightsquigarrow \langle i+1, \rho, n :: s\rangle} \qquad \frac{P[i] = \text{return}}{\langle i, \rho, v :: s\rangle \rightsquigarrow v}$$

Figure 2: OPERATIONAL SEMANTICS FOR $\text{JVM}_\mathcal{I}$

a mild adaptation of the control dependence region analysis.[1]

Further language features, decidability issues, and the relationship between our type systems and bytecode verification are briefly discussed in Section 5.

Preliminaries. For every function $f \in A \to B$, $x \in A$ and $v \in B$, we let $f \oplus \{x \mapsto v\}$ denote the unique function f' s.t. $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$.

Further, we let $A^\star$ denote the set of A-stacks for every set A. We use hd and tl and :: and ++ to denote the head and tail and concatenation and cons operations on stacks.

Throughout the paper, we assume given a set $\mathcal{X}$ of local variables, and a partial order $\mathcal{S}$ of security levels. For simplicity, we assume that $\mathcal{S} = \{L, H\}$ with $L \leq H$, where H is the high level for confidential data, and L is the low level for observable data.

2. THE $\text{JVM}_\mathcal{I}$ SUBMACHINE: SYNTAX AND SEMANTICS

In this section, we define an information flow type system for a simple assembly language. Most definitions are adapted from [2].

2.1 Programs, memory model and operational semantics

A $\text{JVM}_\mathcal{I}$ program P is given by its list of instructions, taken from the instruction set of Figure 1, and its set $\mathcal{X}$ of local variables. We let $\mathcal{PP} = \{1 \ldots n\}$ denote the set of program points of P, where n is the length of the instruction list of P.

Then, the set $\mathsf{State}_\mathcal{I}$ of $\text{JVM}_\mathcal{I}$ states is defined as the set of triples $\langle i, \rho, s\rangle$, where $i \in \mathbb{N}$ is the program counter that points to the next instruction to be executed; $\rho \in \mathcal{X} \to \mathcal{V}$ is a mapping from local variables to values, where the set $\mathcal{V}$ of values is defined as $\mathbb{Z}$; $s \in \mathcal{V}^\star$ is an operand stack.

Finally, the operational semantics, which formalizes one-step execution of the $\text{JVM}_\mathcal{I}$, is given in Figure 2 as a relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \mathsf{State}_\mathcal{I} \times (\mathsf{State}_\mathcal{I} + \mathcal{V})$. We let $\rightsquigarrow^\star$ denote the transitive closure of $\rightsquigarrow$, and write $P, \rho \Downarrow v$ as a shorthand for $\langle 1, \rho, \epsilon\rangle \rightsquigarrow^\star v$.

[1]Designing precise information flow type systems for languages with several kinds of exceptions has been studied notably in [13, 15]. Dealing with different kinds of exceptions at the level of the JVM is left for future work, see Section 5.

2.2 Non-Interference

Non-interference is defined relative to a mapping $\mathsf{vt} : \mathcal{X} \to \mathcal{S}$ that expresses the security policy by assigning a security level to local variables[2].

DEFINITION 1 (NON-INTERFERING $\text{JVM}_\mathcal{I}$ PROGRAM).

1. *The indistinguishability relation $v \sim_k v'$ (where $v, v' \in \mathcal{V}$ and $k \in \mathcal{S}$) is defined as $k = H \vee v = v'$. The relation is extended pointwise to maps[3]: for $\rho, \rho' : \mathcal{X} \to \mathcal{V}$, we have $\rho \sim_{\mathsf{vt}} \rho'$ if for all $x \in \mathcal{X}$, $\rho(x) \sim_{\mathsf{vt}(x)} \rho'(x)$.*

2. *A program P is* non-interfering, *written $\mathsf{NI}(P)$, if for every ρ, ρ', v, v', we have that $P, \rho \Downarrow v$, and $P, \rho' \Downarrow v'$, and $\rho \sim \rho'$ imply $v \sim_L v'$, i.e. $v = v'$.*

The above definition assumes that an attacker has the ability to see the initial memory of the program, and its final result. It is a variant of termination-insensitive information flow adapted to the $\text{JVM}_\mathcal{I}$.

2.3 Type System

In this section, we define an information flow type system that guarantees non-interference. The type system is described by an abstract transition relation that operates on annotated programs, i.e. programs with extra security annotations.

2.3.1 Control dependence region

The type system is parameterized by abstract control dependence regions. These regions are used by the type system to prevent implicit flows of information. A control dependence region computes (an over-approximation of) the range of branching instructions. Formally, we define for every program P its set of conditional control points as $\mathcal{PP}^\sharp = \{i \in \mathcal{PP} \mid P[i] = \text{ifeq } j\}$ and assume given two

[2]We assume that the security level of local variables is fixed throughout execution. Such an assumption restricts the possibility of reusing variables throughout execution. However, it should be possible to extend the results of this paper to allow variable reuse.

[3]Strictly speaking, we should write $\sim_{\mathsf{vt}}$, but usually we simply write $\sim$ since there is no risk of confusion.

functions

$$\begin{array}{rcl} \mathsf{region} & : & \mathcal{PP}^\sharp \to \wp(\mathcal{PP}) \\ \mathsf{jun} & : & \mathcal{PP}^\sharp \rightharpoonup \mathcal{PP} \end{array}$$

that respectively compute the control dependence region and the junction point of an instruction at a given program point.

The successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program p is defined by the clauses (we write $i \mapsto j$ instead of $i, j \in \mapsto$):

- 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, which we write $i \mapsto$;
- otherwise, $i \mapsto i+1$.

We let $\mapsto^\star$ be the reflexive and transitive closure of $\mapsto$. Furthermore, we assume the functions to satisfy some minimal properties, which are required in the proof of noninterference. Intuitively these properties state that the successors of a branching instructions i are in its region and a successor instruction that is not in its region is a junction point (i.e. an instruction that in any execution is a successor of i) or a successor of this (unique) junction point.

PROPERTY 1. *[Safe Over Approximation Property - SOAP] Let $i \in \mathcal{PP}^\sharp$.*

- *If $i' \mapsto i''$, and $i' \in \mathsf{region}(i)$ or $i' = i$, then $i'' \in \mathsf{region}(i)$ or $i'' = \mathsf{jun}(i)$;*
- *If $i \mapsto^\star i'$, and $i' \in \mathcal{PP}^\sharp \cap \mathsf{region}(i)$ then $\mathsf{region}(i') \subseteq \mathsf{region}(i)$;*
- *If $\mathsf{jun}(i)$ is defined, then $\mathsf{jun}(i) \notin \mathsf{region}(i)$ and for all i' s.t. $i \mapsto^\star i'$ then $i' \mapsto^\star \mathsf{jun}(i)$ or $\mathsf{jun}(i) \mapsto^\star i'$.*

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

2.3.2 Abstract transition relation

In the sequel, we let $\mathcal{ST}$ be $\mathcal{S}^\star$, and $\mathcal{SE}$ be $\mathcal{PP} \to \mathcal{S}$. Elements of $\mathcal{ST}$ and $\mathcal{SE}$ are called stack types and security environments respectively. The set tstate of *(security) types* is defined as $\mathcal{ST} \times \mathcal{SE}$.

The type system is defined through an abstract transition system, which manipulates pairs st, se where: st is a stack type that records the security level of values in the operand stack, and se is a security environment which provides for each program point an upper bound for the security level of the regions in which it is included. The transfer rules are of the form

$$\frac{constraints}{i \vdash st, se \Rightarrow st', se'} \qquad \frac{constraints}{i \vdash st, se \Rightarrow}$$

and determine typing constraints for instructions to be executed, and their successors, see Figure 3; in particular, they prevent direct flows and also indirect flows by forbidding assignments to low variables in high regions. We briefly comment on the essential rules.

- The transfer rule for an instruction store x prevents direct flows by requiring that the value on top of the operand stack has a security level k such that $k \leq \mathsf{vt}(x)$. By requiring that $se(i) \leq \mathsf{vt}(x)$, the rule together with the rule for ifeq also prevents implicit flows: $se(i)$ will be H if the store instruction is under the influence of a high ifeq.
- The transfer rule for ifeq updates the security environment (for program points in the control dependence region of the ifeq instruction) and the operand stack, so as to prevent implicit flows. The following example illustrates the need for lifting the operand stack.

  EXAMPLE 1.

  ```
  1  push 1
  2  store x_L
  3  load x_L
  4  push 0
  5  load y_H
  6  ifeq 8
  7  store y_H
  8  store x_L
  ```

  This program is interfering because the value of y_H may leak to x_L. This program is rejected by the type system parameterized by $\mathsf{region}(6) = \{7\}$ *thanks to the* ifeq *rule which lifts the operand stack.*

- The transfer rule for return requires $se(i) = L$, so as to avoid that programs return from inside an high ifeq, which could lead to an indirect flow. In addition, the rule requires that the value on top of the operand stack has a security level L, since it will be observed by the attacker. The following example illustrates the need for preventing return instructions in high regions.

  EXAMPLE 2.

  ```
  1  push 3
  2  store x_L
  3  load y_H
  4  ifeq 7
  5  push 2
  6  return
  7  push 4
  8  load x_L
  9  return
  ```

  This program is interfering because of a return *in a high* ifeq. *This program is rejected by the type system (parameterized, for example, by* $\mathsf{region}(4) = \{5, 6, 7, 8, 9\}$*) thanks to the* ifeq *rule which lifts the security environment, and to* return *rule which prevents the program from returning in a high security environment.*

2.3.3 Typability

Although not essential for our purposes, we opt for a polyvariant analysis as in [6] and consider judgments of the form $S \vdash P$ where $S : \mathcal{PP} \to \wp(\mathsf{tstate})$; in the sequel we write S_i instead of $S(i)$. Further, $S \vdash P$ is set to hold iff $(\epsilon, se_1) \in S_1$ for some se_1 and for all $i, j \in \mathcal{PP}$ and all $(st, se) \in S_i$:

1. $i \mapsto j \Rightarrow \exists (st', se') \in S_j.\ i \vdash (st, se) \Rightarrow (st', se')$;
2. $i \not\mapsto$ implies $i \vdash (st, se) \Rightarrow$.

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st, se \Rightarrow st, se} \qquad \frac{P[i] = \mathsf{store}\ x \quad se(i) \sqcup k \leq \mathsf{vt}(x)}{i \vdash k :: st, se \Rightarrow st, se}$$

$$\frac{P[i] = \mathsf{prim}\ op}{i \vdash k_1 :: k_2 :: st, se \Rightarrow k_1 \sqcup k_2 \sqcup se(i) :: st, se}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st, se \Rightarrow (\mathsf{vt}(x) \sqcup se(i)) :: st, se}$$

$$\frac{P[i] = \mathsf{pop}}{i \vdash k_1 :: st, se \Rightarrow st, se}$$

$$\frac{P[i] = \mathsf{ifeq}\ j}{i \vdash k :: st, se \Rightarrow \mathrm{lift}_k(st), \mathrm{lift}_k(se, \mathsf{region}(i))}$$

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash st, se \Rightarrow se(i) :: st, se}$$

$$\frac{P[i] = \mathsf{return} \qquad se(i) \sqcup k = L}{i \vdash k :: st, se \Rightarrow}$$

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

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

Figure 3: TRANSFER RULES FOR INSTRUCTIONS IN $\mathrm{JVM}_{\mathcal{I}}$

The first property requires that every time there is a possible step from i to j then there is also an abstract step from i to j using the transfer rules of the abstract semantics. This is needed to assure that every possible step done by the operational semantics verifies the constraints imposed by the transfer rules of the abstract semantics. Likewise, the second requirement is to assure that constraints for return are verified. We say a program P is typable, written $\vdash P$, if there exists S s.t. $S \vdash P$.

2.3.4 Soundness

The type system is sound, as stated in the proposition below. Note that the proposition does not assume that P passes the standard bytecode verification, but is true a fortiori for programs that do.

PROPOSITION 1. *Let P be a typable* $\mathrm{JVM}_{\mathcal{I}}$ *program. Then P is non-interfering.*

Soundness is established by a general method, whose idea is to define a notion of state indistinguishability that captures some appropriate invariant, and thus is preserved under execution (the latter has different meanings depending whether a "high" instruction or a "low" instruction is executed, the level of the instruction being determined by a security environment). The main difficulty in defining state indistinguishability resides in defining a good notion of operand stack indistinguishability: in order to account for high branching instructions, indistinguishability between states must encompass states that have operand stacks of different length. Indistinguishability between operand stacks is needed to establish the lemmas that claim that during execution indistinguishability of states is invariant. Then, the lemmas are established by a case analysis on the instruction being executed and its semantics, using the region safe over-approximation and inclusion properties.

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

Figure 4: INSTRUCTION SET FOR $\mathrm{JVM}_{\mathcal{O}}$

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

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

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

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

Figure 5: OPERATIONAL SEMANTICS FOR ADDITIONAL $\mathrm{JVM}_{\mathcal{O}}$ INSTRUCTIONS

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

The object-oriented extension $\mathrm{JVM}_{\mathcal{O}}$ of $\mathrm{JVM}_{\mathcal{I}}$ includes instance fields, creation of new instances, and null pointers. However, we do not deal with object constructors—we assume objects are automatically initialized with default values.

3.1 Programs, memory model and operational semantics

Programs are similar to $\mathrm{JVM}_{\mathcal{I}}$ programs, but also come equipped with a set $\mathcal{C}$ of class names, and a set $\mathcal{F}$ of field names. In addition, the set of $\mathrm{JVM}_{\mathcal{O}}$ values is defined as $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$, where $\mathcal{L}$ is an (infinite) set of locations. $\mathrm{JVM}_{\mathcal{O}}$ programs use an extended set of instructions, given in Figure 4. Then, the set $\mathsf{State}_{\mathcal{O}}$ of $\mathrm{JVM}_{\mathcal{O}}$ states is defined as an extension of $\mathrm{JVM}_{\mathcal{I}}$ states $\langle i, \rho, s, h\rangle$, where i, ρ, and s are defined as in $\mathrm{JVM}_{\mathcal{I}}$ and h is a heap, that accommodates dynamically created objects. Heaps are modeled as a partial function $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{F} \rightharpoonup \mathcal{V}$, i.e. as the set of finite functions—i.e. partial functions with finite domains—from $\mathcal{F}$ to $\mathcal{V}$. We let Heap be the set of heaps.

The operational semantics for the new instructions of $\mathrm{JVM}_{\mathcal{O}}$ relies on an allocator function $\mathsf{fresh} : \mathsf{Heap} \times \mathcal{C} \to \mathcal{L}$, and on a function $\mathsf{default} : \mathcal{C} \to \mathcal{O}$. It is given in Figure 5 as a relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \mathsf{State}_{\mathcal{O}} \times (\mathsf{State}_{\mathcal{O}} + (\mathcal{V} \times \mathsf{Heap}))$. We let $\rightsquigarrow^\star$ denote the transitive closure of $\rightsquigarrow$ and write $P, \mu, h \Downarrow v, h'$ as a shorthand for $\langle 1, \mu, \epsilon, h\rangle \rightsquigarrow^\star v, h'$.

3.2 Non-Interference

Non-interference is defined relative to mappings $\mathsf{vt} : \mathcal{X} \to \mathcal{S}$ and $\mathsf{ft} : \mathcal{X} \to \mathcal{S}$ that express the security policy by assigning a security level to local variables and fields respectively. Following [1], we consider that heaps with different allocations of "high" objects (i.e. objects that have been created in a high security environment) may be indistinguishable by an attacker; therefore indistinguishability is defined relative

to a partial bijection β on locations.

DEFINITION 2 (NON-INTERFERING JVM$_{\mathcal{O}}$ PROGRAM).

1. *Value indistinguishability $v \sim_{\beta,l} v'$, where $v, v' \in \mathcal{V}$ and $l \in \mathcal{S}$, is defined by the clauses:*

$$null \sim_{\beta,L} null \qquad v \sim_{\beta,H} v'$$

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

Value indistinguishability is extended pointwise to local variable maps.

2. *Two heaps h_1 and h_2 are indistinguishable (w.r.t. β), written $h_1 \sim_\beta h_2$, if:*
   - *$dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h_2)$;*
   - *for every $o \in dom(\beta)$, we have $dom(h_1(o)) = dom(h_2(\beta(o)))$;*
   - *for every $f \in dom(h_1)(o)$, we have $h_1(o)(f) \sim_{\beta,\mathsf{ft}(f)} h_2(\beta(o))(f)$.*
3. *A program P is non-interfering, written $\mathsf{NI}(P)$, if for every $\mu, \mu' \in \mathcal{X} \to \mathcal{V}$, and $h, h', h_f, h'_f \in \mathsf{Heap}$ and $v, v' \in \mathcal{V}$, and partial bijection β on locations, we have $P, \mu, h \Downarrow v, h_f$ and $P, \mu', h' \Downarrow v', h'_f$ and $\mu \sim_\beta \mu'$ and $h \sim_\beta h'$ imply $h_f \sim_{\beta'} h'_f$ and $v \sim_{\beta',L} v'$ for some partial bijection $\beta' \supseteq \beta$.*

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

3.3 Type System

The type system for the JVM$_{\mathcal{O}}$ is defined similarly to that of the JVM$_{\mathcal{I}}$.

3.3.1 Control dependence region

The successor relation is extended with the clause $i \mapsto i+1$ for all new instructions. The SOAP properties remain as for the JVM$_{\mathcal{I}}$.

3.3.2 Abstract transition relation

The abstract transition system of the JVM$_{\mathcal{O}}$ extends that of the JVM$_{\mathcal{I}}$ with the typing transfer rules of Figure 6.

- The transfer rule for new adds to the stack type the security level of the current program point, which is intuitively the security level of the newly created object.
- The transfer rule for putfield requires that $k_1 \leq \mathsf{ft}(f)$ in order to prevent an explicit flow from a high value to a low field. The constraint $se(i) \leq \mathsf{ft}(f)$ prevents an implicit flow caused by an assignment to a low field in a high security environment. Finally, the constraint $k_2 \leq \mathsf{ft}(f)$ prevents that distinct assignments to low fields are performed due to the class of the object cannot be statically determined. For example, if the target object has been created or loaded in a high environment, then the type system disallows an assignment to a low field. The following example illustrates the need for this last constraint.

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

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

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

Figure 6: ADDITIONAL TRANSFER RULES FOR JVM$_{\mathcal{O}}$

EXAMPLE 3.

```
1  load y_H
2  ifeq 5
3  new C
4  goto 6
5  new D
6  push 1
7  putfield f
```

Program above is interfering if both C and D have a low field f in common. Indeed, the newly created object of class C or the newly created object of class D will have their field f assigned the value 1 depending on the value of y_H. The program is rejected by the type system (parameterized, for example, by $\mathsf{region}(2) = \{3, 4, 5\}$) thanks to the ifeq *rule which lifts the security environment, the* new *rule which treats the new objects as high, and the* putfield *rule which prevents field assignments for high objects in low security environments. Note that pushing the* putfield *instruction within the branches of the* ifeq *would still result in an interfering program, that will also be rejected by the type system for similar reasons.*

3.3.3 Typability

Typability is defined as for the JVM$_{\mathcal{I}}$, and we use the same notation, i.e. $\vdash P$, to denote that P is typable.

3.3.4 Soundness

The type system is sound. The proof is similar to that of Proposition 1.

PROPOSITION 2. *Let P be a typable* JVM$_{\mathcal{I}}$ *program. Then P is non-interfering.*

4. THE EXCEPTION-HANDLING EXTENSION JVM$_{\mathcal{E}}$ OF JVM$_{\mathcal{O}}$

In this section, we introduce JVM$_{\mathcal{E}}$, an extension of JVM$_{\mathcal{O}}$ with an exception handling mechanism. As mentioned in the introduction, the main objective of this section is to illustrate how exceptions are handled in an information flow type system for a low-level language.

4.1 Background and motivation

The purpose of this section is to briefly describe the exception handling mechanism of the JVM, and compare informally the difference between an information flow type system for exceptions at source and bytecode levels.

4.1.1 Exception handling in the JVM

In a Java-like language, catching exceptions is handled with the command try $\{S_1\}$ catch $\{S_2\}$, where (roughly speaking) S_1 is the piece of code that might produce an exception and S_2 is the code that is executed in the case that S_1 actually throws an exception. At the level of the JVM, there is no command equivalent to try...catch; instead, there is an exception table, i.e. a list of tuples to indicate which is the set of instructions that might produce an exception (starting program point and final program point of S_1) and, in case an exception is thrown, which is the set of instructions to execute to handle the exception (starting program point of S_2).

Consider the program P:

$$\mathsf{try}\{\ y_H.f = v\ ;\ y_H.f'' = v;\ \}\mathsf{catch}\{\ this.f' = v'\ \}$$

The corresponding compiled program is shown as in Example 4. Its exception table consists of a single handler $\langle 3, 4, 9\rangle$.

4.1.2 Non-interference with exceptions

At the source code level, information leakage is prevented by considering for commands typing judgments of the form $\vdash c : \tau, \tau'$ cmd, where τ' is a bound on the guards of exceptions (in contrast to typing judgments of the form $\vdash c : \tau$ cmd for the fragment without exceptions [1]).[4] At the bytecode level, such implicit flows are prevented in a much more direct way, namely by modifying the definition of regions so that they account for exceptions.

Let us return to the example above. The source program is interfering because the assignment $y_H.f$ may throw an exception, skipping the assignment to $y_H.f''$, and causing an assignment to $this.f'$. In Example 4, we explain why the compiled program is rejected by our type system.

4.2 Programs, memory model and operational semantics

Programs are similar to JVM$_\mathcal{O}$ models. However, the instruction set of the JVM$_\mathcal{O}$ is extended with the bytecode throw and we assume that $\mathcal{C}$ contains a distinguished class **Throwable**; for the sake of simplicity, our semantics generates a single kind of exception namely a **Throwable** object, as in [8].

Furthermore, we assume that programs come equipped with a list of exception handlers of the form $\langle b, e, t\rangle$, and a partial function $\mathsf{Handler} : \mathcal{PP} \rightharpoonup E$ that selects the appropriate handler for a given program point. Intuitively, if $\mathsf{Handler}(l) = \langle b, e, t\rangle$, then $l \in [b, e)$ and also the control is transfered to t if the execution at l raises an exception. We write $\mathsf{Handler}(l) \uparrow$ if $\mathsf{Handler}(l)$ is not defined, and $\mathsf{Handler}(l) \downarrow$ otherwise.

Then, the memory model for JVM$_\mathcal{E}$ is the same as JVM$_\mathcal{O}$, and we let $\mathsf{State}_\mathcal{E}$ be the set of JVM$_\mathcal{E}$-states (note that heaps may contain **Throwable** objects). Finally, the operational semantics is defined in Figure 7 as a relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \mathsf{State}_\mathcal{E} \times (\mathsf{State}_\mathcal{E} + (\mathcal{V} \times \mathsf{Heap}) + \{\mathbf{Abnormal}\})$.

Note that our semantics does not raise an exception in case the program is stuck due to stack underflow, or illegal field access or update; instead, the execution gets stuck. Stuck executions are not considered in our definition of non-interference, so they are harmless for our results. Furthermore, it is the purpose of bytecode verification to ensure that no such stuck states occur during execution.

[4]We use such a type system in our ongoing work on type-preserving compilation, see Section 6.2.

$$\frac{P[i] = \mathsf{getfield}\ f \qquad o = null \qquad o' = \mathrm{fresh}(h, \mathrm{Throwable}) \qquad \mathsf{Handler}(i) = \langle b, e, t\rangle}{\langle i, \rho, o :: s, h\rangle \rightsquigarrow \langle t, \rho, o :: \epsilon, h \oplus \{o' \mapsto \mathsf{default}_{\mathbf{Throwable}}\}\rangle}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad o = null \qquad o' = \mathrm{fresh}(h, \mathrm{Throwable}) \qquad \mathsf{Handler}(i) = \langle b, e, t\rangle}{\langle i, \rho, v :: o :: s, h\rangle \rightsquigarrow \langle t, \rho, v :: o :: s, h \oplus \{o' \mapsto \mathsf{default}_{\mathbf{Throwable}}\}\rangle}$$

$$\frac{P[i] = \mathsf{throw} \qquad \mathsf{Handler}(i) = \langle b, e, t\rangle}{\langle i, \rho, o.s, h\rangle \rightsquigarrow \langle t, \rho, o :: \epsilon, h\rangle}$$

$$\frac{P[i] = \mathsf{getfield}\ f \qquad o = null \qquad \mathsf{Handler}(i) \uparrow}{\langle i, \rho, o :: s, h\rangle \rightsquigarrow \mathbf{Abnormal}}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad o = null \qquad \mathsf{Handler}(i) \uparrow}{\langle i, \rho, v :: o :: s, h\rangle \rightsquigarrow \mathbf{Abnormal}}$$

$$\frac{P[i] = \mathsf{throw} \qquad \mathsf{Handler}(i) \uparrow}{\langle i, \rho, o.s, h\rangle \rightsquigarrow \mathbf{Abnormal}}$$

Figure 7: Operational Semantics for JVM$_\mathcal{E}$

4.3 Non-Interference

Non-interference and state indistinguishability are defined is a similar way as for the JVM$_\mathcal{O}$, but we need to account for abnormal termination.

Definition 3 (non-interfering JVM$_\mathcal{E}$ program).

1. *Let $\mathcal{R} = (\mathcal{V} \times \mathsf{Heap}) + \{\mathbf{Abnormal}\}$. For $r, r' \in \mathcal{R}$ and partial bijection β on locations, the relation $r \sim_{\beta,L} r'$ is defined by the clauses* $\mathbf{Abnormal} \sim_{\beta,L} \mathbf{Abnormal}$, *and*

$$\frac{v \sim_{\beta,L} v' \qquad h \sim_{\beta,L} h'}{v, h \sim_{\beta,L} v', h'}$$

2. *A program P is non-interfering, written $\mathsf{NI}(P)$, if for every $\mu, \mu' \in \mathcal{X} \to \mathcal{V}$, and $h, h' \in \mathsf{Heap}$ and $v, v' \in \mathcal{V}$, and partial bijection β, we have $P, \mu, h \Downarrow v, h_f$ and $P, \mu', h' \Downarrow v', h'_f$ and $\mu \sim_\beta \mu'$ and $h \sim_\beta h'$ imply $h \sim_{\beta'} h'$ and $v \sim_{\beta',L} v'$ for some partial bijection $\beta' \supseteq \beta$.*

4.4 Type System

4.4.1 Control dependence region

The main difficulty in extending the type system to JVM$_\mathcal{E}$ is to adapt the definition of control dependence regions. Firstly, we define

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

Secondly, the definition of successor is extended with the following clauses:

- $p[i] = \mathsf{throw}$ and $\mathsf{Handler}(i) = \langle b, e, t\rangle$, then $i \mapsto t$; otherwise, $i \mapsto$;
- if $p[i] = \mathsf{putfield}\ f$, or $p[i] = \mathsf{getfield}\ f$ and furthermore $\mathsf{Handler}(i) = \langle b, e, t\rangle$, then $i \mapsto t$.[5]

[5]Note that we still have $i \mapsto i + 1$ if $p[i] = \mathsf{putfield}\ f$, or $p[i] = \mathsf{getfield}\ f$, so putfield and getfield instructions may have one or two successors.

To the exception of these modifications, the SOAP properties remain as for the JVM$_{\mathcal{I}}$.

4.4.2 Abstract transition system

The abstract transition system of the JVM$_{\mathcal{E}}$ extends that of the JVM$_{\mathcal{O}}$ with the typing transfer rules of Figure 8.

- The new transfer rules deal with instructions that can throw an exception in a similar way as an ifeq instruction, in the sense that they lift the program points in the region of the instruction and the operand stack. The justification for the lift is the same as for ifeq: since a branching in the execution is possible, both branches must be considered with the same level of security.

- For instructions which may raise an exception, but for which the exception is handled, we add two rules to transform the security type for the normal case and for the exceptional case, as in the operational semantics. Implicit flows are avoided because the region of an instruction i that can throw an exception must contain both the code for the handler and all successors of i until the junction point (between the code of the exception an the successors of i) is reached. The following example illustrates the need of including in the control dependence region all successors of i until the junction point.

  Example 4.

  ```
  1   load y_H
  2   push v
  3   putfield f
  4   load y_H
  5   push v
  6   putfield f''
  7   push v
  8   return
  9   load this
  10  push v'
  11  putfield f'
  12  goto 7
  ```

  Program above is interfering if f, f', f'' are low, if the exception table is $\langle 3, 4, 9\rangle$. In a normal execution, the field f'' is assigned the value v, while in an exceptional execution that part of the code is never reached and instead the field f' is assigned the value v'. Since f'' is observable and y_H is high, there is an implicit flow from y_H to f''. The type system (parameterized, for example, by $\mathsf{region}(3) = \{4, 5, 6, 9, 10, 11, 12\}$*) rejects this program, since the instruction* putfield f'' *is in the region of* putfield f*, and the transfer rule for the former lifts the security environment to high, and the transfer rule for the latter rejects the field assignment.*

- For instructions which may raise an exception, and for which the exception is not handled, we require that the security environment at the corresponding program point is low, to prevent information leakage from raising within a high region an exception that escapes this region. We also require that the value which throws an exception has a low security level. The following example illustrates the need for this last constraint.

$$\frac{P[i] = \mathsf{putfield}\ f \qquad \mathsf{ft}(f) \geq k_1 \qquad (se(i) \sqcup k_2) = L \qquad \mathsf{Handler}(i)\uparrow}{i \vdash k_1 :: k_2 :: st, se \Rightarrow \mathsf{lift}_{k_2}(st), \mathsf{lift}_{k_2}(se, \mathsf{region}(i))}$$

$$\frac{P[i] = \mathsf{putfield}\ f \qquad \mathsf{ft}(f) \geq k_1 \sqcup se(i) \sqcup k_2 \qquad \mathsf{Handler}(i)\downarrow}{i \vdash k_1 :: k_2 :: st, se \Rightarrow k_2 :: \epsilon, \mathsf{lift}_{k_2}(se, \mathsf{region}(i))}$$

$$\frac{P[i] = \mathsf{getfield}\ f \qquad (se(i) \sqcup k) = L \qquad \mathsf{Handler}(i)\uparrow}{i \vdash k :: st, se \Rightarrow \mathsf{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st), \mathsf{lift}_k(se, \mathsf{region}(i))}$$

$$\frac{P[i] = \mathsf{getfield}\ f \qquad \mathsf{Handler}(i)\downarrow}{i \vdash k :: st, se \Rightarrow k :: \epsilon, \mathsf{lift}_k(se, \mathsf{region}(i))}$$

$$\frac{P[i] = \mathsf{throw} \qquad (se(i) \sqcup k) = L \qquad \mathsf{Handler}(i)\uparrow}{i \vdash k :: st, se \Rightarrow k :: \epsilon, se}$$

$$\frac{P[i] = \mathsf{throw} \qquad \mathsf{Handler}(i)\downarrow}{i \vdash k :: st, se \Rightarrow k :: \epsilon, se}$$

Figure 8: Additional transfer rules for JVM$_{\mathcal{E}}$

Example 5.

```
1   load y_H
2   push v
3   puftield f
```

Program above is interfering because, if y_H is null the assignment to the low field f is never performed. The type system rejects this program, since the instruction putfield f *requires that* $\mathsf{ft}(f) \geq \mathsf{vt}(y_H)$.

4.4.3 Typability

Typability is defined as for the JVM$_{\mathcal{I}}$ and JVM$_{\mathcal{O}}$, and we use the same notation, i.e. $\vdash P$, to denote that P is typable.

4.5 Soundness

Throughout this section, we assume that P is a typable program.

Theorem 1. $\mathsf{NI}(P)$.

In order to prove soundness, we first need to define an appropriate notion of indistinguishability between states. We have already defined indistinguishability between local variable maps and heaps, so we only need to define indistinguishability between operand stacks. We write $\mathsf{high}\ (s, st)$ if s and st have the same length n and $st[i] = H$ for every i, $1 \leq i \leq n$.

Definition 4.

1. *Let $s, s' \in \mathcal{V}^\star$ and $st, st' \in \mathcal{ST}$. Then $s \sim_{\beta,st,st'} s'$ is defined inductively:*

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

$$\frac{s \sim_{st} st' \ s' \qquad v \sim_{\beta,k} v'}{v :: s \sim_{\beta,k::st,k::st'} v' :: s'}$$

2. *Two states $s = \langle i, \rho, os, h\rangle$ and $s' = \langle i', \rho', os', h'\rangle$ are indistinguishable w.r.t. $st, st' \in \mathcal{ST}$, and a partial bijection β, written $s \sim_{\beta,st,st'} s'$ iff $\rho \sim_\beta \rho'$, and $os \sim_{\beta,st,st'} os'$, and $h \sim_\beta h'$.*

Note that indistinguishability between operand stacks does not require that the two operand stacks are of the same length. This more permissive definition of indistinguishability is required for the proof of Lemma 1 to go through.

The proof of soundness relies on two lemmas, each of which is proven by a double case analysis on the instruction to be executed and on its semantics.

LEMMA 1. *[One-Step Noninterference in High-Level Environments] Consider two states* $s = \langle i, \rho, os, h\rangle$ *and* $s' = \langle i', \rho', os', h'\rangle$ *and let* $(st, se) \in S_i$. *Assume that* $s \rightsquigarrow s'$ *and* $\mathsf{high}(os, st)$ *and that* $i \in \mathsf{region}(j)$ *for some* j *s.t. for all* $i' \in \mathsf{region}(j)$ $se(i') = H$ *and* $s \sim_{\mathsf{id},st,st} s$. *Then there exists* $(st', se) \in S_{i'}$ *such that* $\mathsf{high}(os', st')$, *and* $s \sim_{\mathsf{id},st,st'} s'$.

Note that Lemma 1 does not deal with abnormal termination, since the type system prevents typable programs to terminate abnormally in a high level environment.

LEMMA 2. *[One-Step Noninterference in Low-Level Environments] Let* $s_1 = \langle i, \rho_1, os_1, h_1\rangle$ *and* $s_2 = \langle i, \rho_2, os_2, h_2\rangle$, *be states, and let* $(st_1, se), (st_2, se) \in S_i$. *Assume that* $s_1 \rightsquigarrow s_1'$, *and* $s_2 \rightsquigarrow s_2'$, *and* $s_1 \sim_{\beta,st_1,st_2} s_2$ *for some partial bijection* β *on locations. Then one of the following holds:*

- $s_1' = \langle i_1', \rho_1', os_1', h_1'\rangle$ *and* $s_2' = \langle i_2', \rho_2', os_2', h_2'\rangle$, *and there exist* $(st_1', se') \in S_{i_1'}$ *and* $(st_2', se') \in S_{i_2'}$ *s.t.* $s_1' \sim_{\beta',st_1',st_2'} s_2'$, *for some* β' *such that* $\beta' \supseteq \beta$. *Furthermore, one of the following holds:*
  1. $i_1' = i_2'$;
  2. $\mathsf{high}(os_1', st_1')$ *and* $\mathsf{high}(os_2', st_2')$, *and for every* $j \in \mathsf{region}(i)$ $se'(j) = H$.
- $s_1', s_2' \in \mathcal{R}$ *and Then* $s_1' \sim_{\beta,L} s_2'$.

The proof is based on a general method, which has been used in [2], and formalized in the Coq proof assistant [7].

PROOF OF PROPOSITION 1. Consider the two execution paths

$$\begin{array}{l} s_1 \rightsquigarrow s_2 \rightsquigarrow \ \ldots \ \rightsquigarrow s_{n_1} \\ s_1' \rightsquigarrow s_2' \rightsquigarrow \ \ldots \ \rightsquigarrow s_{n_2}' \end{array}$$

where

- $s_1 = \langle 1, \rho, \epsilon\rangle$, and $s_1' = \langle 1, \rho', \epsilon\rangle$;
- $\rho \sim \rho'$;
- $s_{n_1}, s_{n_2} \in \mathcal{R}$.

By invoking Lemma 2 as long as it applies, we conclude for some maximal q that $s_q, s_q' \in \mathcal{R}$, in which case we are done, or that $s_q = \langle i_q, \mu_q, os_q, h_q\rangle$ and $s_q' = \langle i_q', \mu_q', os_q', h_q'\rangle$, and that there exist β' such that $\beta' \supseteq \beta$, and $(st_q, se_q) \in S_{i_q}$ and $(st_q', se_q') \in S_{i_q}$ such that $s_q \sim_{\beta',st_q,st_q'} s_q'$ with $\mathsf{high}\ (os_q, st_q)$ and $\mathsf{high}\ (os_q', st_q')$, and $se_q(i_q) = se_q'(i_q') = H$.

By invoking Lemma 1 as long as it applies on each reduction sequence starting from s_q and s_q', we can conclude that there exist $s_{q_1} = \langle i_{q_1}, \mu_{q_1}, os_{q_1}, h_{q_1}\rangle$ and and there exist $s_{q_2}' = \langle i_{q_2}', \mu_{q_2}', os_{q_2}', h_{q_2}'\rangle$, and $(st_{q_1}, se_{q_1}) \in S_{i_{q_1}}$ and $(st_{q_2}', se_{q_2}') \in S_{i_{q_2}}$ s.t. $s_q \sim_{\mathsf{id},st_q,st_{q_1}} s_{q_1}$ and $s_q' \sim_{\mathsf{id},st_q',st_{q_2}'} s_{q_2}$. Using the SOAP property, one concludes that $i_{q_1} = i_{q_2}'$. Furthermore, by some transitivity result for the indistinguishability relation we have $s_{q_1} \sim_{\beta',st_{q_1},st_{q_2}'} s_{q_2}'$, so we can apply Lemma 2 again. □

5. DISCUSSION

In this section we briefly discuss extensions, limitations and decidability issues of our type system.

5.1 Method calls

Although we do not provide details here, we have extended our analysis to methods, and shown its soundness. The extension is compatible with the modularity of bytecode verification, and assumes that methods come equipped with a security signature

$$\vec{k_1} \xrightarrow{k_3,k_4} k_2$$

that gives the expected security level of its arguments $\vec{k_1}$, of its result k_2, and its effect k_3 on the global store, and the maximal security level of the security environment where an exception not handled in the method might be thrown (note that [1] uses a similar notion of signature, but only considers $\vec{k_1}$, k_2, k_3 are as in [1] as exceptions are not part of their language).

The main difficulties are first to extend state indistinguishability so that it relates states whose frame stacks are of different lengths (which is required to handle high method calls, i.e. method calls that have a high effect on the heap), second to handle information leakages caused by dynamic method dispatch, and third of all, to control information leakage through exceptions that escape from the method in which they are raised and that are thus propagated to other methods. The first point is dealt using similar ideas as for operand stack indistinguishability, and the second point is handled as in [1], and the third point is handled using k_4.

5.2 System exceptions and multiple exceptions

Our operational semantics does not deal with system exceptions such as **OutOfMemory** exceptions, and hence our notion of non-interference does not need to deal with system exceptions. If we assume the heap to be bounded, information leakages may be caused by **OutOfMemory** exceptions, as illustrated by the program (where x is high):

```
load x
ifeq 5
new C
goto 3
return
```

One brutal solution would be to prevent dynamic object creation in high-level environments; another rather brutal solution would be to split the heap into a high-level part, and a low-level part, so as to avoid that object allocation leaks information.

Besides, our operational semantics only considers one kind of exception. It is unimportant for the setting of the JVM$_{\mathcal{E}}$ not to have several kind of exceptions (because putfield and getfield only raise null pointer exceptions), but it is crucial to account for different kinds of exceptions larger fragments of the JVM.

We have not yet attempted to consider the possibility of having several exceptions, since our focus was rather on how to design a sound informal flow type system for a low-level language with a simple exception handling mechanism. However, we believe that it should be possible to adapt existing work, and in particular the work of Myers [13], which we briefly describes below.

In the JIF prototype, developed by Myers [13], the static checker determines for each expression its path label, that collects the different levels of information transmitted by different possible termination paths (normal termination, termination through a return, and termination through different kinds of exception). This fine-grained labeling is appealing when dealing with different kinds of exceptions: throwing and catching exceptions does not necessarily taint subsequent computations.

It should be possible for our type system to allow several types of exceptions by adapting the path labels of [13] to our setting. To this end, it will be necessary to let control dependence regions depend on the kind of exception that a given instruction might throw, and by tracking information about which exceptions can be thrown in our type system.

5.3 Subroutines

Finally, we have extended our type system with subroutines, i.e. jsr and ret instructions. The main complexity here is that subroutines yield a complex successor relation, and that the control dependence regions cannot be computed syntactically any longer(see [6] to further explanation about the complexity of verifying programs with subroutines) . Instead we need to rely on an address type system, whose abstract states consist of a stack of address types, and of a map that assigns to every register an address type, where an address type is defined as $\{\mathbf{Ret}\ j \mid j \in \mathbb{N}^+\} \cup \{\delta\}$. The typing transfer rules are of the form

$$\frac{constraints}{a, i \vdash st, se \Rightarrow st', se'}$$

$$\frac{constraints}{a, i \vdash st, se \Rightarrow}$$

where a is an abstract state.

We have not proved soundness of our type system for two reasons. Firstly, the analysis requires two passes over the program: a first pass to compute the control dependence regions, and another pass for the information flow analysis. Secondly, the type system becomes rather complex, and we believe that machine-checked proofs are required.

5.4 Decidability issues and relation to bytecode verification

Our information flow type system is decidable, and the typability of a program might be decided using Coglio's variant [6] of Kildall's algorithm—alternatively, and as long as we do not deal with polymorphic subroutines, we could adopt a simpler type system and stick to Kildall's original algorithm [10].

PROPOSITION 3. *Let P be a* JVM$_\mathcal{E}$ *program. Then it is decidable whether $\vdash P$ holds,* provided *we impose an upper bound on the length of the operand stack (remark that we impose an upper bound on the length of the operand stack and not the stack of method calls).*

The proof follows from the termination of Coglio's analysis, and from the fact that it characterizes typable programs.

In addition, since the analysis is done methodwise, our information flow type system is compatible with bytecode verification algorithms [11], with which it will need to be combined to handle more complex fragments of the JVM, including fragments that consider different forms of exceptions.

6. CONCLUSION

We have defined an information flow type system that ensures termination-insensitive non-interference for a representative sequential fragment of the JVM. By focusing on one mainstream language that is used for mobile code, we hope to contribute to the applicability of information flow (although, as pointed in [20], there are other important challenges to address).

6.1 Related work

Sabelfeld and Myers [16] provide an excellent survey of the literature on non-interference. We refer to their survey for a more complete overview of related work and only focus on very closely related work; in particular, we do not discuss static analyses of object-oriented programs in any detail.

Non-interference for low-level languages has been studied by Zdancewick and Myers [19] for a λ-calculus with jumps, and by Barthe, Rezk and Basu [2] for a simple assembly language, and more recently by Bonelli, Compagnoni and Medel [5] again for a simple assembly language, using linear continuations for computing control dependence regions.

Besides, Lanet *et al.*, see e.g. [4], develop a method to detect illicit flows for a sequential fragment of the JVM. In a nutshell, they proceed by specifying in the SMV model checker a symbolic transition semantics of the JVM that manipulates security levels, and by verifying that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their approach has been refined by Bernardeschi and De Francesco, see e.g. [3], for a subset of the JVM that includes jumps, subroutines but no method invocation, exceptions, or instructions for heap manipulation. However, these works do not provide a proof of non-interference.

Recently, Genaim and Spoto [9] propose an information flow analysis for Java Bytecode, using abstract interpretation and boolean functions. However, their system does not consider objects.

Non-interference for Java is considered by Banerjee and Naumann for a non-trivial fragment of Java [1]; variants of their results have been machine-checked independently by Strecker [18] and Naumann [14]. The JIF prototype, developed by Myers [13], provides an information flow analysis for a very rich fragment of Java; however, its complexity is such that non-interference for JIF has not been studied formally.

Non-interference for CAML has also been studied by Pottier and Simonet [15]. They provide a non-interference soundness proof for their type system, which encompasses most major features of CAML, including exceptions. In their work, a security level is associated with every exception name as in [13].

Safety type systems for low-level languages have been studied e.g. by Morrisett et al. [12], who formalized idealized versions of a Type Assembly Language (TAL), and proved type safety results.

6.2 Ongoing and future work

Using the type system of this paper (and its extension with method calls), and an extension of the type system of [1] to Middleweight Java with exceptions, we have extended our earlier results on information flow types preserving compilation [2] and derived non-interference for the source language (joint work with D. Naumann). While this result shows that our analysis is sufficiently accurate to ac-

cept a large class of programs, we believe that it can be made more precise by combining it with effect systems that provide additional information about the exceptional behavior of programs, or with assertions. E.g. the program in Example 5 is always rejected because field access could throw an exception; however, we may know from the enclosing context that no exception will be thrown, and an effect system or an assertion could guide the information flow type system into accepting the program.

Using the Coq proof assistant [7], we have also formalized with a general framework to establish non-interference for low-level languages, and instantiated our framework to the $\text{JVM}_{\mathcal{I}}$ (joint work with F. Kammüller). In the future, we would like to cover larger sequential fragments of the JVM, including the $\text{JVM}_{\mathcal{E}}$ and its extension with subroutines, arrays, and method calls.

A more ambitious objective would be to address the concurrent aspects of the JVM; however, most works on information flow type systems for concurrent languages adopt a more restrictive notion of non-interference, and preliminary investigations are required to understand whether one can get a meaningful definition of termination-insensitive non-interference in a concurrent setting, or whether one can adapt our type system to other notions of non-interference that are better fitted for concurrent settings.

Acknowledgments: The authors would like to thank David Naumann, Florian Kammüller, and anonymous reviewers for providing valuable comments on draft versions of this paper.

7. REFERENCES

[1] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 200x. Special Issue on Language-Based Security. To appear.

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

[3] C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15, 2002.

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

[5] E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis, 2004. Manuscript.

[6] A. Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, 2004.

[7] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.4*, February 2003.

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

[9] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. *To appear in VMCAI'05*, Paris, France. January 2005.

[10] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.

[11] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *In ACM Transactions on Programming Languages and Systems*, 21(3):528-569, May 1999.

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

[14] D. Naumann. Machine-checked correctness of a secure information flow analyzer (preliminary report). Technical Report CS-2004-10, Stevens Institute of Technology, March 2003.

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

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

[17] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer-Verlag, 2001.

[18] M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[19] S. Zdancewic and A. Myers. Secure information flow and CPS. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 2001.

[20] S. Zdancewic. Challenges in information flow security. In R. Giacobazzi, editor, *Informal proceedings of PLID'04*, 2004.