Type-Based Information Flow Analysis for Bytecode Languages with Variable Object Field Policies

Francisco Bavera
Dept. de Computación, FCEFQyN, UNRC, Argentina and CONICET
pancho@dc.exa.unrc.edu.ar

Eduardo Bonelli
LIFIA, Fac. de Informática, UNLP, Argentina and CONICET
eduardo@lifia.info.unlp.edu.ar

ABSTRACT

Static, type-based information flow analysis techniques targeted at Java and JVM-like code typically assume a global security policy on object fields: all fields are assigned a fixed security level. In essence they are treated as standard variables. However different objects may be created under varying security contexts, particularly for widely used classes such as wrapper or collection classes. This entails an important loss in precision of the analysis. We present a flow-sensitive type system for statically detecting illegal flows of information in a JVM-like language that allows the level of a field to vary at different object creation points. Also, we prove a noninterference result for this language.

1. INTRODUCTION

Information flow analysis (IFA) [12] based on type systems for programming languages [13] has received considerable attention given its potential to enforce secure, end-to-end flow of data at the source code level. Although initial work was developed for imperative languages, most current efforts are geared towards object-oriented and concurrent languages. Somewhat orthogonally, and sparked mainly by the success of Java and the JVM [11], also low-level code formats are being studied. This allows the direct analysis of Java applets and other code distributed over the Internet.

Most extant literature on type-based IFA [1, 2, 5, 6, 4] assumes that the fields of objects are assigned some fixed security label given that, once assigned a value, they do behave similarly to variables in imperative languages. However, some objects such as instances of, say, class Collection or WrapInt (wrapper class for integers), are meant to be used in different contexts. Consider the aforementioned WrapInt class that has a field for storing integers. We illustrate our argument by means of the following example. This program loads the value of x, which we assume to be secret, on the stack, creates a new instance of WrapInt and assigns 0 to its only field f (the possible security levels for f and their consequences are discussed below). It then branches on the secret value of x. Instructions 6 to 8 thus depend on this value and hence the value 1 assigned to field f of the new instance of WrapInt shall be considered secret.

```
1. load x
2. new WrapInt
3. push 0
4. putfield f
5. if 9
6. new WrapInt
7. push 1
8. putfield f
9. return
```

If field f of WrapInt is declared public, then the assignment on line 4 is safe but not the one on line 7. Since this would render the wrapper useless, one could decide to declare f secret. This time both assignments are acceptable, however the public value 0 is unnecessarily coerced to secret with the consequent loss in precision.

In this paper we propose JVM^s ("s" is for "safe"), a core bytecode language together with a type system that performs IFA in a setting where the security level assigned to fields may vary at different points of the program where their host object is created. Two further contributions, less significant although worthy of mention, are that local variables are allowed to be reused with different security levels and also that we enhance precision on how information flow through the stack is handled. Even though variable reuse can be avoided in the analysis by transforming the bytecode so that different uses of a variable become different variables [10], this makes the security framework dependent on yet another component. Moreover, this transformation is not viable in other low-level code such as assembly language where the number of registers is fixed. Regarding the operand stack the extant literature on type-based IFA for bytecode [5, 6, 4] take the approach of modifying the level of the elements of the stack when evaluating branching instructions. In particular, whenever a conditional on a secret expression is evaluated, the security level of *all* the elements in the stack are set to secret. Since Java compilers use the operand stack for evaluation of expressions and thus is empty at the beginning and end of each instruction, this is not a severe limitation. However, this entails that the precision of the analysis depends on the fact that the bytecode is produced by a Java compiler. To sum up, although less significant, these are worthy contributions towards minimizing the assumptions on the code that is to be analysed hence furthering its span of applicability.

Further details. Due to space limitations a description of related work and further details and extensions to this

I	::=	`prim` *op*	primitive operation
	|	`push` *j*	push *j* on stack
	|	`pop`	pop from 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
	|	`new` *C*	create new object in heap
	|	`getfield` *f*	load value of field *f* on stack
	|	`putfield` *f*	store top of stack in field *f*

where *op* is $+$ or $\times$, $x \in \mathbb{X}$, $j \in \mathbb{N}$, and $f \in \mathbb{F}$.

Figure 1: Bytecode instructions

paper may be found elsewhere [8].

2. SYNTAX AND SEMANTICS

A *program* B is a sequence of bytecode instructions (Fig. 1). We write $\mathsf{Dom}(B)$ for the set of *program points* of B and $B(i)$, with $i \in 1..n$ and n the length of B, for the i^{th} instruction of B. Furthermore, x ranges over a set of local variables $\mathbb{X}$, and f over a fixed set of field identifiers $\mathbb{F}$. $\mathbb{C}$ is our universe of class names.

A *value* v is either an integer i, a (heap) location o or the null object *null*. Thus, if $\mathbb{L}$ stands for the set of locations, then the values are $\mathbb{V} = \mathbb{Z} \cup \mathbb{L} \cup \{null\}$. *Machine states* are tuples $\langle i, \alpha, \sigma, \eta \rangle$, where $i \in \mathbb{N}$ is the program counter that points to the next instruction to be executed; α, (local variable array) is a mapping from local variables to values; σ (stack) is an operand stack; and η (heap) is a mapping from locations to objects. *Objects* are modeled as functions assigning values to their fields[1]. A machine state *for* B is one in which the program counter points to an element in $\mathsf{Dom}(B)$. The small-step operational semantics of JVM^s is described as a binary relation $s_1 \longrightarrow s_2$ that states how s_2 is obtained from s_1 by one step of reduction. Sample reduction schemes are depicted below.

$$\frac{B(i) = \texttt{new}\ C \quad o = \mathsf{Fresh}(\eta)}{\langle i, \alpha, \sigma, \eta \rangle \longrightarrow_B \langle i+1, \alpha, o \cdot \sigma, \eta \oplus \{o \mapsto \mathsf{Default}(C)\} \rangle}\ (\textsc{O-New})$$

$$\frac{B(i) = \texttt{putfield}\ f \quad o \in \mathsf{Dom}(\eta) \quad f \in \mathsf{Dom}(\eta(o))}{\langle i, \alpha, v \cdot o \cdot \sigma, \eta \rangle \longrightarrow_B \langle i+1, \alpha, \sigma, \eta \oplus \{o \mapsto \eta(o) \oplus \{f \mapsto v\}\} \rangle}\ (\textsc{O-PtFld})$$

$$\frac{B(i) = \texttt{getfield}\ f \quad o \in \mathsf{Dom}(\eta) \quad \eta(o)(f) = v}{\langle i, \alpha, o \cdot \sigma, \eta \rangle \longrightarrow_B \langle i+1, \alpha, v \cdot \sigma, \eta \rangle}\ (\textsc{O-GtFld})$$

There are also expressions of the form $\langle v, \eta \rangle$ are referred to as *final states*. A non-final state s may either reduce to another non-final state or to a final state. Initial states take the form $\langle 1, \alpha, \epsilon, \eta \rangle$, where ϵ denotes the empty stack. Summing up, $\longrightarrow_B \subseteq$ *State* $\times$ (*State* $+$ ($\mathbb{V} \times$ *Heap*)), where *State* denotes the set of states and *Heap* the set of heaps. We write $\twoheadrightarrow_B$ for the reflexive-transitive closure of $\longrightarrow_B$.

3. TYPE SYSTEM

We assume given a set $\{\mathtt{L}, \mathtt{H}\}$ of *security levels* (l) equipped with $\preceq$, the least partial order satisfying $\mathtt{L} \preceq \mathtt{H}$, and write $\sqcup, \sqcap$ for the induced supremum and infimum. *Security labels* (κ) are expressions of the form $\langle \{a_1, \ldots, a_n\}, l \rangle$ where a_i, $i \in 0..n$, ranges over a given infinite set of *symbolic locations* (motivated shortly). If $n = 0$, then the security label is $\langle \emptyset, l \rangle$. We say a_i (for each $i \in 1..n$) and l are the symbolic locations and level of the security label, respectively. We occasionally write $\langle \_, l \rangle$, or l, when the set of symbolic locations is irrelevant. The ordering on security labels is defined as $\langle R_1, l_1 \rangle \sqsubseteq \langle R_2, l_2 \rangle$ iff $R_1 \subseteq R_2$ and $l_1 \preceq l_2$. By abuse of notation we write $\kappa \sqsubseteq l$ (resp. $\kappa \not\sqsubseteq l$) to indicate that the level of κ is (resp. not) below or equal to l and $\kappa \neq l$ to indicate that the level of κ is different from l. The supremum on security labels is defined as follows: $\langle R_1, l_1 \rangle \sqcup \langle R_2, l_2 \rangle = \langle R_1 \cup R_2, l_1 \sqcup l_2 \rangle$.

A *method* M is an expression of the form $((x_1 : \kappa_1, \ldots, x_n : \kappa_n, \kappa_r), B)$ abbreviated $((\vec{x} : \vec{\kappa}, \kappa_r), B)$, where B is a program referred to as its body and $\kappa_1, \ldots, \kappa_n, \kappa_r$ are the security labels of the formal parameters $x_1, \ldots, x_n$ and the value returned by the method, resp. Methods M are typed by means of a *typing judgement*: $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd M$. Each of $\mathcal{V}, \mathcal{S}, \mathcal{A}$ and $\mathcal{T}$ are called *typing contexts*. Typing contexts supply information needed to type each instruction of M. As such they constitute families of either functions ($\mathcal{V}, \mathcal{S}, \mathcal{T}$) or labels ($\mathcal{A}$) indexed by a finite set of instruction addresses. We write $\mathcal{V}_i$ to refer to the i^{th} element of the family and similarly with the others. The first typing context assigns a *variable array type* V to each instruction (of M). A variable array type is a function assigning security labels to each local variable. Thus $\mathcal{V}_i$ indicates the types of the local variables at instruction i. The $\mathcal{S}$ typing context assigns a *stack type* S to each instruction. A stack type is a function assigning security labels to numbers from 1 to n, where n is assumed to be the length of the stack. $\mathcal{A}$ indicates the level ($\mathtt{L}$ or $\mathtt{H}$) of each instruction. Finally, $\mathcal{T}$ associates a *heap type* T, explained below, to each instruction.

The field of each class is assigned either a (fixed) security label $\langle \emptyset, l \rangle$ or the special symbol $\star$ for declaring the field polymorphic as explained below. This is determined by a function $\mathsf{ft} : \mathbb{F} \to \kappa \cup \{\star\}$. Fields declared polymorphic adopt a level determined by the type system at the point where creation of its host object takes place. The field is given the security level of the context in which the corresponding `new` instruction is executed. Given that multiple instances of the same class may have possibly different security labels for the same fields, we need some bookkeeping. This is achieved by associating with each location a *symbolic location*. Heap types map symbolic locations to expressions of the form $[f_1 : \kappa_1, \ldots, f_n : \kappa_n]$ called *object types*. An object type gives information on the security label of each field. We write $T(a, f)$ for the label associated to field f of the object type $T(a)$.

Two further ingredients are required before formulating the type system. The first is a notion of subtyping for variable array, heap and stack types.

$$\frac{\forall x \in \mathbb{X}. V_1(x) \sqsubseteq V_2(x)}{V_1 \leq V_2} \qquad \frac{\mathsf{Dom}(T_1) \subseteq \mathsf{Dom}(T_2)}{T_1 \leq T_2}$$

$$\frac{\begin{array}{c} |S_1| = |S_2| = n \\ \forall j \in 1..n. \left\{ \begin{array}{ll} S_1(j) \sqsubseteq S_2(j) & \text{if } l \sqsubseteq S_1(j), \\ S_1(j) = S_2(j) & \text{otherwise} \end{array} \right. \end{array}}{S_1 \leq_l S_2}$$

Variable array types are compared pointwise using the ordering on security labels. A heap type T_1 is a subtype of T_2

[1]We assume that the field names of all classes are different.

if the domain of T_1 is included in that of T_2. $S_1 \leq_l S_2$ asserts that stack type S_1 is a subtype of S_2 at level l. Stack types may only be compared if they are of the same size. Furthermore, we allow depth subtyping at position i provided that $S(i)$ is at least l. The other requirement is the availability of control dependence region information [5, 6, 4].

High-level languages have control-flow constructs that explicitly state dependency. Given that such constructs are absent from JVM^s, as is the case in most low-level languages, and that they may be the source of unwanted information flows, our type system requires this information to be supplied. Let $\mathsf{Dom}(B)^\sharp$ denote the set of program points of B where branching instructions occur (i.e. $\mathsf{Dom}(B)^\sharp = \{k \in \mathsf{Dom}(B) \mid B(k) = \mathtt{if}\ i\}$). We assume given two functions ($\wp$ below denotes the powerset operator): (1) region: $\mathsf{Dom}(B)^\sharp \to \wp(\mathsf{Dom}(B))$ and (2) jun: $\mathsf{Dom}(B)^\sharp \to \mathsf{Dom}(B)$.

The first computes the *control dependence region*, an over-approximation of the range of branching instructions and the second the unique junction point of a branching instruction at a given program point. Following previous work [3, 6, 4] we only require some abstract conditions on these functions to hold, referred to as the *safe over approximation property* or *SOAP*. SOAP states how regions and junctions points relate to one another [8].

3.1 Typing Schemes

A set of typing schemes define when a method is to be considered *well-typed*. We assume that this method is valid JVM code in the sense that it passes the bytecode verifier. For example, in an instruction such as $\mathtt{prim}\ +$ we do not check whether the operands are indeed numbers. The typing schemes rely on previously supplied region and jun satisfying the SOAP properties.

Method M is *well-typed* if there exist typing contexts $\mathcal{V}$, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{T}$, such that the type judgement $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd M$ holds. The typing rule defining this judgement is

$$\frac{\begin{array}{c}\forall x_j \in \vec{x}.\mathcal{V}_1(x_j) = \kappa_j \quad \forall x_j \in \vec{x}.\mathcal{T}_1(\kappa_j) \text{ defined} \quad \mathcal{S}_1 = \epsilon \\ \forall i \in \mathsf{Dom}(B).\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathbin{\vec{:}} \kappa, \kappa_r), B)\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd ((x \mathbin{\vec{:}} \kappa, \kappa_r), B)}$$

The variable array type $\mathcal{V}_1$ must provide labels for all parameters and must agree with the ones assigned to each of them by the declaration $x \mathbin{\vec{:}} \kappa$. Also, if the label of a parameter has a nonempty set of symbolic locations, then $\mathcal{T}_1$ does not leave out these references. The stack is assumed to be empty. Finally, all program points should be well-typed under the typing contexts $\mathcal{V}, \mathcal{S}, \mathcal{A}$ and $\mathcal{T}$. A program point i of B is well-typed if the judgement $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathbin{\vec{:}} \kappa, \kappa_r), B)$ holds. This will be the case if it is derivable using the *typing schemes for instructions*, a sample of which are provided in Fig. 2.

T-PRIMOP requests that $\mathcal{S}_i$ have at least two elements on the top and that the top element of $\mathcal{S}_{i+1}$, the stack type of the successor instruction, have at least that of these elements joined with the current program counter level. We write $\mathcal{S}_{i+1}(0)$ for the topmost element of the stack type $\mathcal{S}_{i+1}$ and $\mathcal{S}_{i+1}\backslash 0$ for $\mathcal{S}_{i+1}$ without the topmost element. Subtyping (rather than equality) for variable array, stack and heap types are required given that instruction $i + 1$ may have other predecessors apart from i. T-RET simply requires that the label of the topmost element of the stack does not leak any information. Here κ_r is the label of the result of the method body. Note that $\mathcal{A}(i)$ is not resorted to given that by assumption there is a unique $\mathtt{return}$ instruction and this instruction is executed with $\mathcal{A}(i)$ at low.

In order to type $\mathtt{new}\ C$ a fresh symbolic location is created and the heap type of the successor of i is required to define an object type for it. This object type assigns a security label to each field according to $\mathtt{ft}_{\mathcal{A}(i)}$. For those fields declared polymorphic, the level $\mathcal{A}(i)$ is assigned. T-PTFLD requires that no information about the value written to field f, the reference of the object (l) nor the current security context be leaked. Note that label $\langle R, l\rangle$ may contain multiple symbolic references in R. For example, this would be the case if a new object was created in two different branches of a conditional before merging and then executing the $\mathtt{putfield}$ instruction. The remaining schemes may be understood along similar lines.

Regarding type checking, it can be implemented as a (monovariant) control flow analysis algorithm that explores abstract execution paths [8].

4. NONINTERFERENCE

Noninterference [9] ensures that any two computations that are fired from initial states that differ only in values indistinguishable for an external observer, are equivalent. The principle effort in proving this result is in obtaining an appropriate notion of indistinguishable states. This of course depends on the types of illicit flows that are to be captured. In particular, we have to provide a definition that caters for the features advertised in the the introduction, including unrestricted reuse of local variables and a more precise control of the operand stack.

We begin our discussion with some preliminary notions. Given that in any two runs of a program different heap locations may be allocated, it is convenient to introduce a (partial) bijection β between locations [2]. For example, two runs of $\mathtt{new}\ C$ could allocate different heap locations, say o_1 and o_2, in the heap to the new object. This bijection helps establish that these locations are related by setting $\beta(o_1) = o_2$. Given that JVM^s tracks the types of fields via symbolic locations we also introduce a pair of (partial) bijections between symbolic locations and locations themselves: $(\beta^{\lhd}, \beta^{\rhd})$. In our example, $\beta^{\lhd}(a) = o_1$ and $\beta^{\rhd}(a) = o_2$. Notice that both locations get assigned the same symbolic location given that it arises from the typing derivation of $\mathtt{new}\ C$. The sum of β and $(\beta^{\lhd}, \beta^{\rhd})$ is called a *location bijection set.* Location bijection sets shall thus be required for relating heaps (as explained below).

Definition 1. A *location bijection set* β consists of (1) a (partial) bijection β_{loc} between locations; and (2) a pair of (partial) bijections $(\beta^{\lhd}, \beta^{\rhd})$ between symbolic locations and locations where $\mathsf{Dom}(\beta_{loc}) \subseteq \mathsf{Ran}(\beta^{\lhd})$ and $\mathsf{Ran}(\beta_{loc}) \subseteq \mathsf{Ran}(\beta^{\rhd})$ s.t. for all $o \in \mathsf{Dom}(\beta_{loc})$, $\beta^{\lhd-1}(o) = \beta^{\rhd-1}(\beta(o)))$.

By default, we write $\beta(o)$ to mean $\beta_{loc}(o)$. We define location bijection set β' to be an *extension of* β, written $\beta \subseteq \beta'$, if $\beta_{loc} \subseteq \beta'_{loc}$ and $\mathsf{Ran}(\beta^{\lhd}) \subseteq \mathsf{Ran}(\beta'^{\lhd})$ and $\mathsf{Ran}(\beta^{\rhd}) \subseteq \mathsf{Ran}(\beta'^{\rhd})$. We write id for a location bijection set β such that β_{loc} is the identity on locations.

4.1 Indistinguishability - Definitions

In order to relate states we need to be able to relate each of its components. This includes values, variable arrays, stacks and heaps. Values and stacks are standard; local variable

$$\text{(T-PRIMOP)}\quad \frac{\begin{array}{l} B(i) = \texttt{prim}\ op \\ (\mathcal{A}(i) \sqcup \kappa' \sqcup \kappa'') \sqsubseteq \mathcal{S}_{i+1}(0) \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \kappa' \cdot \kappa'' \cdot (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

$$\text{(T-IF)}\quad \frac{\begin{array}{l} B(i) = \texttt{if}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \kappa \cdot \mathcal{S}_{i+1} = \kappa \cdot \mathcal{S}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1}, \mathcal{T}_{i'} \\ \forall k \in \texttt{region}(i).\kappa \sqsubseteq \mathcal{A}(k) \\ i+1, i' \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

$$\text{(T-STR)}\quad \frac{\begin{array}{l} B(i) = \texttt{store}\ x \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \backslash x \leq \mathcal{V}_{i+1} \backslash x \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{A}(i) \sqsubseteq \mathcal{V}_{i+1}(x) \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

$$\text{(T-NEW)}\quad \frac{\begin{array}{l} B(i) = \texttt{new}\ C \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \mathcal{T}_i \cdot a : [f_1 : \texttt{ft}_{\mathcal{A}(i)}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}(i)}(f_n)] \leq \mathcal{T}_{i+1} \\ \langle \{a\}, \mathcal{A}(i) \rangle \cdot \mathcal{S}_i \leq_{\mathcal{A}(i)} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

$$\text{(T-PTFLD)}\quad \frac{\begin{array}{l} B(i) = \texttt{putfield}\ f \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \kappa' \cdot \langle R, l \rangle \cdot \mathcal{S}_{i+1} \\ \mathcal{A}(i) \sqcup \kappa' \sqcup l \sqsubseteq \bigsqcap_{a \in R} \mathcal{T}_i(a, f), \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

$$\text{(T-GTFLD)}\quad \frac{\begin{array}{l} B(i) = \texttt{getfield}\ f \\ \mathcal{A}(i) \sqsubseteq \mathcal{S}_i(0) = \langle R, l \rangle \\ \kappa = \bigsqcup_{a \in R} \mathcal{T}_i(a, f) \\ l \sqcup \kappa \cdot (\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}(i)} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)}$$

Figure 2: Sample Typing Schemes

arrays requires considering whether they are to related in a low or high-level region given that variable reuse may take place [8]. Here we briefly discuss the case for heaps. In order to compare heaps η_1 and η_2 we verify that all objects allocated in η_1 are indistinguishable with regards to their corresponding objects, as dictated by β, in the other heap (cf. condition 5 of Def. 2). Before comparing objects it is determined whether references o allocated in η_1 and which have a related allocated reference $\beta(o)$ in η_2 are indeed of the same type (cf. condition 4 of Def. 2). Then, they are compared field by field using the security level given by the heap type at the relevant program point (if the instruction is i, then it would be $\mathcal{T}_i(\beta^{\lhd -1}(o), f)$). The remaining conditions in the definition below are sanity checks that are in fact preserved by reduction. Regarding the use of $T_1(\beta^{\lhd -1}(o), f)$ in condition 5, rather than $T_2(\beta^{\rhd -1}(\beta(o)), f)$, the reader should note that the following invariant is seen to hold all along the execution of the program: for every $o \in \mathsf{Dom}(\beta)$, $T_1(\beta^{\lhd -1}(o), f) = T_2(\beta^{\rhd -1}(\beta(o)), f)$.

Definition 2. Let η_1, η_2 be heaps, T_1, T_2 heap types and β a location bijection set. We define η_1 and η_2 to be indistinguishable at T_1, T_2 under β ($\eta_1 \sim_{\beta,(T_1,T_2)} \eta_2$[2]) if:

1. $\mathsf{Dom}(\beta) \subseteq \mathsf{Dom}(\eta_1)$ and $\mathsf{Ran}(\beta) \subseteq \mathsf{Dom}(\eta_2)$,
2. $\mathsf{Ran}(\beta^{\lhd}) \subseteq \mathsf{Dom}(\eta_1)$ and $\mathsf{Ran}(\beta^{\rhd}) \subseteq \mathsf{Dom}(\eta_2)$,
3. $\mathsf{Dom}(\beta^{\lhd}) \subseteq \mathsf{Dom}(T_1)$ and $\mathsf{Dom}(\beta^{\rhd}) \subseteq \mathsf{Dom}(T_2)$,
4. for every $o \in \mathsf{Dom}(\beta)$, $\mathsf{Dom}(\eta_1(o)) = \mathsf{Dom}(\eta_2(\beta(o)))$,
5. for every $o \in \mathsf{Dom}(\beta)$, for every field $f \in \mathsf{Dom}(\eta_1(o))$

$$\eta_1(o, f) \sim_{\beta, T_1(\beta^{\lhd -1}(o), f)} \eta_2(\beta(o), f).$$

4.2 Indistinguishability - Properties

Determining that indistinguishability of values, local variable arrays, stacks and heaps is an equivalence relation requires careful consideration on how location bijection sets are composed. Furthermore, in the presence of variable reuse transitivity of indistinguishability of variable arrays in fact fails unless additional conditions are imposed. These issues are briefly discussed in this section.

In order to state transitivity of indistinguishability for values (or any component of a machine state) location bijection sets must be composed. A location bijection set β is said to be *composable* with another location set γ if for all $o \in \mathsf{Dom}(\beta)$, $\beta^{\rhd -1}(\beta(o)) = \gamma^{\lhd -1}(\beta(o))$. Note that any two location sets may be assumed to be composable w.l.o.g. If β is composable with γ, then we can define their composition $\gamma \circ \beta$ as follows: (1) $(\gamma \circ \beta)_{loc} = \gamma_{loc} \circ \beta_{loc}$[3]; (2) $(\gamma \circ \beta)^{\lhd} = \beta^{\lhd}$ and (3) $(\gamma \circ \beta)^{\rhd} = \gamma^{\rhd}$.

It may be verified that, as defined, $\gamma \circ \beta$ is indeed a location bijection set. Also define the *inverse* of a location bijection set β, denoted $\hat{\beta}$, to be: $\hat{\beta}_{loc} = \beta_{loc}^{-1}$ and $\hat{\beta}^{\lhd} = \beta^{\rhd}$ and $\hat{\beta}^{\rhd} = \beta^{\lhd}$.

Variable reuse allows a public variable to be reused for storing secret information in a high security execution context. Suppose, therefore, that $\alpha_1 \sim_{\beta,(V_1,V_2),\mathtt{H}} \alpha_2$ and also $\alpha_2 \sim_{\gamma,(V_2,V_3),\mathtt{H}} \alpha_3$ where, for some x, $V_1(x) = \mathtt{L}$, $V_2(x) = \mathtt{H}$ and $V_3(x) = \mathtt{L}$. Clearly it is not necessarily the case that $\alpha_1 \sim_{\gamma \circ \beta,(V_1,V_3),\mathtt{H}} \alpha_3$ given that $\alpha_1(x)$ and $\alpha_3(x)$ may differ. We thus require that either V_1 or V_3 have at least the level of V_2 for this variable: $V_2(x) \sqsubseteq V_1(x)$ or $V_2(x) \sqsubseteq V_3(x)$. Of course, it remains to be seen that such a condition can be met when proving noninterference, we defer that discussion to Sec. 4.3.

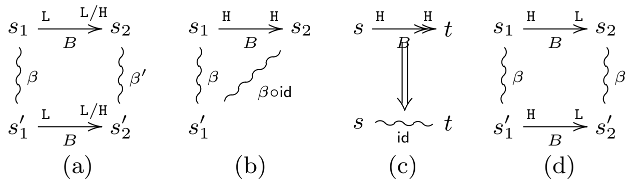
The case of stacks and heaps are dealt with similarly. Together these results determine that machine state indistinguishability too is an equivalence relation.

4.3 Noninterference

Noninterference states that any two terminating runs of a well-typed method starting from indistinguishable initial states produce indistinguishable final states. Let M be a well-typed method $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd ((x \mathrel{\vec{:}} \kappa, \kappa_r), B)$. We say M satisfies *noninterference*, if for every α_1, α_2 variable arrays, v_1, v_2 values, $\eta_1, \eta_2, \eta_1', \eta_2'$ heaps and β a location bijection set: (1) $\langle 1, \alpha_1, \epsilon, \eta_1 \rangle \twoheadrightarrow \langle v_1, \eta_1' \rangle$; (2) $\langle 1, \alpha_2, \epsilon, \eta_2 \rangle \twoheadrightarrow \langle v_2, \eta_2' \rangle$; (3) $\alpha_1 \sim_{\beta,\mathcal{V}_1,\mathtt{L}} \alpha_2$ and (4)$\eta_1 \sim_{\beta,\mathcal{T}_1} \eta_2$, implies $\eta_1' \sim_{\beta',\mathcal{T}_i} \eta_2'$ and $v_1 \sim_{\beta',\kappa_r} v_2$, for some location bijection set $\beta' \supseteq \beta$.

[2]In the case that $T_1 = T_2$ we often write $\eta_1 \sim_{\beta,T_1} \eta_2$.

[3]It is partial functions that are composed: $\mathsf{Dom}(\gamma_{loc} \circ \beta_{loc}) = \{o \in \mathsf{Dom}(\beta_{loc}) \mid \beta_{loc}(o) \in \mathsf{Dom}(\gamma_{loc})\}$.



$s_1 \xrightarrow[B]{\mathtt{L}\quad \mathtt{L/H}} s_2$
β
β'
$s'_1 \xrightarrow[B]{\mathtt{L}\quad \mathtt{L/H}} s'_2$
(a)
$s_1 \xrightarrow[B]{\mathtt{H}\quad \mathtt{H}} s_2$
β
$\beta \circ \mathsf{id}$
s'_1
(b)
$s \xrightarrow[B]{\mathtt{H}\quad \mathtt{H}} t$
$s \sim t$
id
(c)
$s_1 \xrightarrow[B]{\mathtt{H}\quad \mathtt{L}} s_2$
β
β
$s'_1 \xrightarrow[B]{\mathtt{H}\quad \mathtt{L}} s'_2$
(d)

Figure 3: Unwinding lemmas

With the definition of noninterference in place we now formulate the soundness result of our type system.

Proposition 1. M satisfies noninterference.

The proof relies on three *unwinding lemmas* depicted in Fig. 3((a), (b) and (d)). Consider two runs of M, one from s_1 and another from s'_1 as depicted below:

$$\begin{tikzcd}
s_1 \arrow[r, two heads] \arrow[d, squiggly, "\beta"'] \arrow[dr, phantom, "Fig.3(a)"] & s_j \arrow[r, "\mathsf{id}_1"'] \arrow[d, squiggly, "\beta_1"] & \cdots \arrow[r, "\mathsf{id}_{k_1}"'] & s_{h_1} \arrow[r, two heads] \arrow[d, squiggly, "\hat{\beta_3}"'] \arrow[dr, phantom, "Fig.3(d)"] & s_{h_3} \arrow[r] \arrow[d, squiggly, "\beta_2"] & \cdots s_n \arrow[d, squiggly, "\beta_m"] \\
s'_1 \arrow[r, two heads] & s'_j \arrow[r, "\mathsf{id}'_1"'] & \cdots \arrow[r, "\mathsf{id}'_{k_2}"'] & s'_{h_2} \arrow[r, two heads] & s'_{h_4} \arrow[r] & \cdots s'_n
\end{tikzcd}$$

Computation from s_1 and s'_1 may be seen to unwind via Fig. 3(a), in lock-step, until the security context is raised to high at some states s_j and s'_j, resp. At this point, unwinding continues independently in each of the computations starting from s_j and s'_j until each of these reaches a `goto` instruction, say at state s_{h_1} for the first computation and s'_{h_2} for the second, that lowers the security level of the context back to low. Since all values manipulated in these two intermediate computations are high level values, s_j is seen to be high-indistinguishable from s_{h_1} and, likewise, s'_j is seen to be high-indistinguishable from s'_{h_2}, both facts are deduced from Fig. 3(b) (whose proof relies on Fig. 3(c)). This final step requires that we meet the condition on transitivity of variable arrays, as discussed in Sec. 4.2. The only problematic case would be when (1) $s_1 \longrightarrow s_2$ reuses a secret variable with public data and (2) s'_1 declares this variable to be public. Although (2) may of course hold, (1) cannot: any value written to a variable in a high security context must be secret as may be gleaned from the typing rule for `store`.

At this point both computations exit their high level regions and coincide at the unique junction point of these regions. The resulting states s_{h_3} and s'_{h_4} now become low-indistinguishable according to Fig. 3(d).

5. CONCLUSIONS

We have presented a type system for ensuring secure information flow in a JVM-like language that allows instances of a class to have fields with security levels depending on the context in which they were instantiated. This differs over the extant approach of assigning a global fixed security level to a field, thus improving the precision of the analysis as described in the introduction. We are currently experimenting with our implementation on larger examples.

Regarding future work we are developing an extension of our type system that supports method invocation. Support for threads seems to be absent in the literature on IFA for bytecode languages (a recent exception being [7]) and would be welcome. The same applies to declassification. Regarding the latter, it should be mentioned that some characteristics of low level code such as variable reuse enhance the capabilities of an attacker to declassify data. Some preliminary results in this direction have been obtained and shall be presented in forthcoming work.

Ack. To Gilles Barthe for fruitful discussions and the referees for suggestions for improving the paper.

6. REFERENCES

[1] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267. IEEE Computer Society Press, 2002.

[2] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special Issue on Language-Based Security.

[3] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*, 2005.

[4] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer-Verlag, 2007.

[5] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. of TLDI '05*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[6] G. Barthe, T. Rezk, and D. A. Naumann. Deriving an information flow checker and certifying compiler for java. In *S&P*, pages 230–242. IEEE Computer Society, 2006.

[7] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. of the 12th ESORICS*, LNCS. Springer-Verlag, 2007. To appear.

[8] F. Bavera and E. Bonelli. `www.lifia.info.unlp.edu.ar/~eduardo/publications/jvmsLong.pdf`, 2007.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April, 1982.

[10] X. Leroy. Bytecode verification for java smart card. *Software Practice and Experience*, 32:319–340, 2002.

[11] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison Wesley, 1999.

[12] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[13] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621, 1997.