

# Java bytecode verification for secure information flow \*

Marco Avvenuti , Cinzia Bernardeschi and Nicoletta De Francesco

Dipartimento di Ingegneria dell'Informazione

Università di Pisa

Via Diotisalvi 2, I-56126 Pisa, Italy

{m.avvenuti,c.bernardeschi,n.defrancesco}@iet.unipi.it

## ABSTRACT

Security of Java programs is important as they can be executed in different platforms. This paper addresses the problem of secure information flow for Java bytecode. In information flow analysis one wishes to check if high security data can ever propagate to low security observers. We propose a static analysis similar to the type-level abstract interpretation used for standard bytecode verification. Instead of types, our technique works with secrecy levels assigned to classes, methods' parameters and returned values, and handles implicit information flows. A verification tool based on the proposed technique is under development. Using the tool, programs downloaded from untrusted hosts can be checked locally prior to executing them.

## Categories and Subject Descriptors

F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*mechanical verification, specification techniques*; D.2.4 [Software Engineering]: Software/Programs Verification—*formal methods, validation*

## General Terms

Languages, Security, Theory, Verification

## Keywords

Data protection, information flow, Java bytecode, static analysis, verification tool

## 1. INTRODUCTION

The Java run-time system allows programs compiled into the Java Virtual Machine bytecode language (JVML) [18] to be dynamically downloaded. There are situations where the secrecy of sensitive data stored on the host where the bytecode executes is at risk. For example, assume that the

downloaded code has access to the user's private data in order to compute some information. If the program also communicates over the Internet, the private data could be leaked. Users have the option of preventing information release by using the access control mechanism. However, this can be in some cases unsuitable, since useful programs generally need access to private files and to communication resources in order to perform their tasks.

To address this problem, we can adopt a security policy that grants access to private data based on the program's needs, but, at same time, we should check if high security data can ever propagate to low security observers. To define the privacy of data and to control the allowed information flow occurring during program execution, we can use a multi-level security model. System's resources, besides being characterized by access rights, are also assigned a secrecy level. The secrecy levels are partially ordered in a lattice, and it is required that information at a given secrecy level does not flow to lower levels. This property is called *secure information flow* and has been addressed in many papers [10, 2, 4, 12, 19, 17, 1, 20, 14, 5].

For example, a file can have read access right, but it can be required that the information it holds is not transferred into a public file. In this case, the data read from the file are private, i.e. data are considered to have a high secrecy level. A secrecy level can be assigned to the input and the output of a program, according to the files taken as input and produced as output by the program itself: the input/output corresponding to a file containing private information will be assigned a high secrecy level, while the input/output corresponding to a public file will be assigned a low secrecy level. The program has secure information flow if every output having a given secrecy level does not depend on any input at higher secrecy levels.

This paper describes a technique that helps to protect sensitive data on the host by detecting illegal information flows in Java bytecode. Previous work in bytecode verification for secure information flow has based on model checking to perform the verification [7, 9]. Our approach strives to analyzing the code directly. A program is checked before execution, applying a technique similar to the type-level abstract interpretation used for verifying the bytecode correctness. Here, the algorithm adopted by the standard bytecode Verifier is applied to the domain of secrecy levels instead of types.

\*This work was supported by Fondazione Cassa di Risparmio di Pisa, Italy (N. PR02-182).

Secure information flow verification is performed on a per-method basis, assigning secrecy levels to classes, methods' parameters and returned values. As a value used inside the scope of a conditional branch depends on the condition tested by the instruction, a main point is the handling of implicit information flows due to conditional branches. A fixpoint iteration is applied: the fixpoint shows, for each instruction, the least upper bound of secrecy levels every value in a register or in the operand stack depends on. Illegal information flows may occur with instructions performing instance variable update, method invocation or return.

A prototype tool based on the technique presented in the paper is under development. The tool can be added to the host's protection resources. The bytecode should be subject to this static verification before being executed. Since the verification is performed by the executing machine itself, the downloading of code from untrusted hosts can be allowed.

## 2. BACKGROUND

We consider the subset of JVMIL shown in Figure 1. The JVM is a stack machine manipulating an operand stack and a set of local registers for each method, and a heap containing object instances [18]. The instructions are typed. In the figure,  $\alpha$  means a set of types: for example, `iload` represents loading an integer value, `aload` represents loading an object reference. `putfield` and `getfield` instructions are used to write and read object fields, respectively. To begin with, we consider methods having one and only one parameter and assume no dynamic object creation. Also, we do not consider subroutines, used in Java to implement exceptions. We will add these features in further work.

The bytecode of a method is a sequence  $B$  of instructions. When a method is invoked (`invoke` instruction), it executes with a new empty stack and with an initial memory where all registers are undefined except for the first one, register  $x0$ , that contains the reference to the object instance on which the method is called, and register  $x1$ , that contains the actual parameter. When the method returns, control is transferred to the calling method: the caller's execution environment (operand stack and local registers) is restored and the returned value, if any, is pushed onto the operand stack.

We give the semantics of the language as a set of inference rules. We consider a set  $\mathcal{C}$  of class definitions and a set  $Objects = \{r_1, \dots, r_n\}$  of references to instances of the classes in  $\mathcal{C}$ . We denote by  $\mathcal{O} = Objects \rightarrow ObjectValues$  the domain of object valuations, i.e. the functions assigning a value to each object in  $Objects$ . Figure 2 reports the rules for some instructions. An execution state of a method  $C.mt$  is a tuple  $(B, i, M, S)$ , where  $B$  is the bytecode corresponding to  $C.mt$ ,  $i$  is the address held by the program counter,  $M : Registers \rightarrow Values$  is the local memory, representing the current state of the local registers of  $B$ , and  $S \in Values^*$  is the current state of the operand stack. Given  $x$ , we denote by  $M(x)$  the contents of  $x$  in the memory  $M$ . The initial state of the execution of a method  $C.mt$  is  $(B, 0, M_0, \lambda)$ , where  $0$  is the address of the first instruction,  $M_0(x0)$  and  $M_0(x1)$  are set to the reference to the object and to the actual parameter, respectively; the other registers are undefined. We denote by  $\Omega$  the domain of the states of all

<code>pop</code>	Pop top operand stack element.
<code>dup</code>	Duplicate top operand stack element.
<code><math>\alpha op</math></code>	Pop two operands with type $\alpha$ off the operand stack, perform the operation $op \in \{ \text{add}, \text{mult}, \text{compare} \dots \}$ , and push the result onto the stack.
<code><math>\alpha const d</math></code>	Push constant $d$ with type $\alpha$ onto the operand stack.
<code><math>\alpha load x</math></code>	Push the value with type $\alpha$ of the register $x$ onto the operand stack.
<code><math>\alpha store x</math></code>	Pop a value with type $\alpha$ off the operand stack and store it into local register $x$ .
<code>ifcond <math>j</math></code>	Pop a value off the operand stack, and evaluate it against the condition $cond = \{ \text{eq}, \text{ge}, \text{null}, \dots \}$ ; branch to $j$ if the value satisfies $cond$ .
<code>goto <math>j</math></code>	Jump to $j$ .
<code>getfield <math>C.f</math></code>	Pop a reference to an object of class $C$ off the operand stack; fetch the object's field $f$ and put it onto the operand stack.
<code>putfield <math>C.f</math></code>	Pop a value $k$ and a reference to an object of class $C$ from the operand stack; set field $f$ of the object to $k$ .
<code>invoke <math>C.mt</math></code>	Pop value $k$ and a reference $r$ to an object of class $C$ from the operand stack; invoke method $C.mt$ of the referenced object with actual parameter $k$ .
<code><math>\alpha return</math></code>	Pop the $\alpha$ value off the operand stack and return it from the method.

Figure 1: Instruction set.

methods. A state of the system execution is a pair  $\langle A, O \rangle$  where  $A \in \Omega^*$  is the activation stack and  $O \in \mathcal{O}$  is the current state of the objects.  $O(r.f)$  denotes the contents of field  $f$  of object  $r$ .

$M[k/x]$  is used to indicate the memory  $M'$  which agrees with  $M$  for all registers, except for  $x$ , for which it is  $M'(x) = k$ . Similarly,  $O[k/r.f]$  indicates the object valuation  $O'$ , which differs from  $O$  only on field  $f$  of object  $r$ , which is assigned  $k$ . The rules of the semantics define a transition relation  $\longrightarrow$  between states. We denote as  $\longrightarrow^*$  the reflexive and transitive closure of  $\longrightarrow$ .

The bytecode is subject to a static analysis called bytecode verification, whose purpose is to make sure that the code is well typed [11]. The verification consists in executing a data-flow analysis applied to a type-level abstract interpretation of the virtual machine. The abstract interpreter manipulates stacks of types and registers of types and simulates the execution of instructions at the level of types. Bytecode verification has been formalized as fixpoint construction in [15].

## 3. THE SECURITY MODEL

We assume a lattice  $\Sigma$  of secrecy levels, among which a partial order relation  $\sqsubseteq$  is established. Let  $\sigma_0$  be the bottom element of  $\Sigma$ . Given a set of class definitions  $\mathcal{C}$ , a security specification  $\mathcal{S} : \mathcal{C} \rightarrow \Sigma$  associates to each class  $C \in \mathcal{C}$  a secrecy level. We assume that all instances and all attributes

<b>load</b>	$\frac{B[i] = \alpha\text{load } x}{\langle(B, i, M, S) \cdot A, O\rangle \longrightarrow \langle(B, i + 1, M, M(x) \cdot S) \cdot A, O\rangle}$
<b>getfield</b>	$\frac{B[i] = \text{getfield } C.f}{\langle(B, i, M, r \cdot S) \cdot A, O\rangle \longrightarrow \langle(B, i + 1, M, O(r.f) \cdot S) \cdot A, O\rangle}$
<b>putfield</b>	$\frac{B[i] = \text{putfield } C.f}{\langle(B, i, M, k \cdot r \cdot S) \cdot A, O\rangle \longrightarrow \langle(B, i + 1, M, S) \cdot A, O[k/r.f]\rangle}$
<b>invoke</b>	$\frac{B[i] = \text{invoke } C.mt \quad B' \text{ is the bytecode of } C.mt}{\langle(B, i, M, k \cdot r \cdot S) \cdot A, O\rangle \longrightarrow \langle(B', 0, M[r/x0][k/x1], \lambda) \cdot (B, i + 1, M, S) \cdot A, O\rangle}$
<b>return</b>	$\frac{B[i] = \alpha\text{return}}{\langle(B, i, M, k \cdot \lambda) \cdot (B', j, M', S) \cdot A, O\rangle \longrightarrow \langle(B', j, M', k \cdot S) \cdot A, O\rangle}$

Figure 2: Standard semantics rules

of a class have the level of the class. We denote by  $Mt ds(\mathcal{C})$  the methods belonging to the classes in  $\mathcal{C}$ . We now define the  $\sigma$ -secure information flow for a method in  $Mt ds(\mathcal{C})$ . The definition is parametric with respect to a security specification  $\mathcal{S}$ , an assignment  $\mathcal{P} : Mt ds(\mathcal{C}) \rightarrow \Sigma$  of a secrecy level to the parameter of each method in  $Mt ds(\mathcal{C})$ , and an assignment  $\mathcal{R} : Mt ds(\mathcal{C}) \rightarrow \Sigma$  of a secrecy level to the return value of each method. A method has secure information flow if two executions of the method starting from object valuations that agree on the objects with secrecy level  $\sqsubseteq \sigma$ , and with the same value for the parameter, if it has a secrecy level  $\sqsubseteq \sigma$ , produce object valuations that agree on the objects with secrecy level  $\sqsubseteq \sigma$ , and return the same value if the method return is  $\sqsubseteq \sigma$ . Given a valuation  $O \in \mathcal{O}$  and a secrecy level  $\sigma$ , we denote by  $O \downarrow_\sigma$  the restriction of  $O$  to the objects with secrecy level  $\sqsubseteq \sigma$ .

**DEFINITION 1 (METHOD'S SECURE INFORMATION FLOW).** Let  $\mathcal{C}$  be a set of class definitions and  $\mathcal{O}$  be a set of object instances of classes in  $\mathcal{C}$ . Let  $\sigma \in \Sigma$ . Given  $\mathcal{S} : \mathcal{C} \rightarrow \Sigma$  and  $\mathcal{P}, \mathcal{R} : Mt ds(\mathcal{C}) \rightarrow \Sigma$ , a method  $C.mt \in Mt ds(\mathcal{C})$  has  $\sigma$ -secure information flow with respect to  $\mathcal{S}, \mathcal{R}, \mathcal{P}$  (it is  $\mathcal{S}, \mathcal{R}, \mathcal{P}$ - $\sigma$ -secure) if the following property holds:

Let  $B$  be the bytecode corresponding to  $C.mt$ . Let  $O_1, O_2 \in \mathcal{O}$  with  $O_1 \downarrow_\sigma = O_2 \downarrow_\sigma$ . Let  $M_1$  and  $M_2$  such that

- $M_1(x0) = M_2(x0)$
- $M_1(x1) = M_2(x1)$  if  $\mathcal{P}(C.mt) \sqsubseteq \sigma$
- $M_1(x) = M_2(x) = \text{undefined}$  for each  $x \neq x0, x1$

$\langle(B, 0, M_1, \lambda) \cdot \lambda, O_1\rangle \xrightarrow{*} \langle(B, i, \bar{M}_1, \bar{S}_1) \cdot \lambda, \bar{O}_1\rangle$   
with  $B[i] = \alpha\text{return}$  and

$\langle(B, 0, M_2, \lambda) \cdot \lambda, O_2\rangle \xrightarrow{*} \langle(B, j, \bar{M}_2, \bar{S}_2) \cdot \lambda, \bar{O}_2\rangle$   
with  $B[j] = \alpha\text{return}$

implies

- $\bar{O}_1 \downarrow_\sigma = \bar{O}_2 \downarrow_\sigma$
- if  $\bar{S}_1 = k_1$ ,  $\bar{S}_2 = k_2$ , and  $\mathcal{R}(C.mt) \sqsubseteq \sigma$ , then  $k_1 = k_2$

$C.mt$  is  $\mathcal{S}, \mathcal{R}, \mathcal{P}$ -secure if it is  $\mathcal{S}, \mathcal{R}, \mathcal{P}$ - $\sigma$ -secure for each  $\sigma \in \Sigma$ .

Information flow in the bytecode of a method can be explicit or implicit. We have explicit flow when an assignment is executed. We have implicit flow when a value is used inside the scope of a conditional instruction, as it depends on the condition tested by the instruction. A violation of secure information flow occurs when an object field is assigned a value depending, explicitly or implicitly, on a flow with a higher secrecy level.

Consider the bytecode shown in Figure 3, corresponding to a method  $mt$  of a class  $A$ . Suppose that register  $x1$  (the parameter of  $A.mt$ ) contains a reference to an object of another class  $B$ . Moreover, assume that the secrecy level of class  $B$  is higher than the secrecy level of class  $A$ :  $\mathcal{S}(A) \sqsubset \mathcal{S}(B)$ . Note that register  $x0$  contains a reference to  $A$ . After the bytecode has been executed, the final value of field  $f1$  of the object of class  $A$  is 0 or 1 depending on the value of field  $f2$  of the object of class  $B$ .

The example code represents an *implicit* information flow. Secure information flow is violated since checking the final value of  $A.f1$  reveals information on the value of the higher secrecy field  $B.f2$ . Explicit and implicit information flows are propagated also by method call and return.

```

0: aload    x0
1: aload    x1
2: getfield B.f2
3: ifge     6
4: iconst   0
5: goto     7
6: iconst   1
7: putfield A.f1
8: iconst   1
9: return

```

Figure 3: An implicit flow

## 4. THE METHOD

The Information Flow Verification algorithm for Java bytecode (briefly JBIFV) behaves in a way similar to that of

the standard Java bytecode verifier, which checks the bytecode for correctness. In JBIFV values, instead of being abstracted onto types, are abstracted onto secrecy levels. The main difference between types and secrecy levels is that, while types are "context-free", the secrecy level of a value depends on the context in which the value is manipulated (implicit flows).

We assume that the bytecode which JBIFV is applied to is correct from the point of view of standard verification. JBIFV takes as input a set  $\mathcal{C}$  of classes, an assignment  $\mathcal{S} : \mathcal{C} \rightarrow \Sigma$  of a secrecy level to the classes and an assignment  $\mathcal{P}, \mathcal{R} : \text{Mtds}(\mathcal{C}) \rightarrow \Sigma$  of a secrecy level to the parameter and the result of each method in  $\text{Mtds}(\mathcal{C})$ . As well as standard verification does, JBIFV verifies a method in  $\text{Mtds}(\mathcal{C})$  at a time, assuming that, when verifying a method, the other methods satisfy secure information flow. During the abstract execution of the bytecode made by JBIFV, each value is represented by the secrecy level corresponding to the least upper bound of the secrecy levels of the information flows, both explicit and implicit, the value depends on. To handle implicit flows, the interpreter executes instructions under a *security environment*, which represents the least upper bound of the secrecy levels of the open implicit flows when the instruction is executed. Moreover, the notion of immediate postdominator in control flow graphs is used.

Given a bytecode  $B$ , the *control flow graph* of the bytecode is the directed graph  $(V, L)$ , where  $V$  is the set of nodes, one for each instruction; and  $L \subseteq V \times V$  contains the edge  $(i, j)$  if and only if the instruction at address  $j$  can be immediately executed after that at address  $i$ . For simplicity, we assume that the control flow graph has one and only one final node. If  $i, j \in V$ ,  $j$  *postdominates*  $i$ , denoted by  $j \text{ pd } i$ , if  $j \neq i$  and  $j$  is on every path from  $i$  to the final node.  $j$  *immediately postdominates*  $i$ , denoted by  $j = \text{ipd}(i)$ , if  $j \text{ pd } i$  and there is no node  $r$  such that  $j \text{ pd } r \text{ pd } i$ .

Data manipulated under a security environment  $E$  are considered to have as secrecy level at least  $E$ . When a conditional branching instruction at address  $i$  is executed, the environment is possibly upgraded to take into account the secrecy level of the tested value. Every instruction belonging to a branch starting from  $i$  is executed under this environment. The instruction where all the branches starting from  $i$  join is the first instruction not executed conditionally. If we consider the control flow graph of the program, this instruction is the immediate postdominator of  $i$  [3].

## 4.1 The interpreter

JBIFV performs an abstract execution of the bytecode of a method. Each instruction  $i$  is assigned a state  $Q_i$ , representing the state in which instruction  $i$  is executed.  $Q_i$  is an abstraction of the JVM's state *before* the execution of  $i$ .  $Q_i$  is a triple  $(M, S, E)$ , where  $M : \text{Registers} \rightarrow \Sigma$  is a mapping from local registers to secrecy levels (the memory),  $S \in \Sigma^*$  is a mapping from the elements in the operand stack to secrecy levels (the stack) and  $E \in \Sigma$  is the secrecy level of the environment.

A partial order relation on the domain of the states corresponding to the instructions is defined. This relation is induced from the ordering relation among secrecy levels.

Given two memories  $M_1$  and  $M_2$ ,  $M_1 \sqsubseteq M_2$  iff for each register  $x$ , it is  $M_1(x) \sqsubseteq M_2(x)$ . The domain of stacks has a bottom element,  $\perp_S$ , which is  $\sqsubseteq$  of all stacks. Given two stacks  $S_1$  and  $S_2$  different from  $\perp_S$ ,  $S_1 \sqsubseteq S_2$  iff  $S_1$  and  $S_2$  have the same length and each item in  $S_1$  is  $\sqsubseteq$  of the item occurring in  $S_2$  in the same position. Stacks with different length are unrelated. Given two states  $Q_i = (M, S, E)$  and  $Q'_i = (M', S', E')$ ,  $Q_i \sqsubseteq Q'_i$  iff  $M \sqsubseteq M', S \sqsubseteq S', E \sqsubseteq E'$ .

The least upper bound operation on states corresponding to instructions is defined point-wise on memories, stacks and environments:

$$(M, S, E) \sqcup (M', S', E') = (M \sqcup M', S \sqcup S', E \sqcup E').$$

The abstract interpreter uses also a restriction of the least upper bound operation, denoted as  $\sqcup_{M,S}$ .  $\sqcup_{M,S}$  performs the least upper bound on the memory and the stack, and returns the environment of the first operand (this operation is not commutative):

$$(M, S, E) \sqcup_{M,S} (M', S', E') = (M \sqcup M', S \sqcup S', E).$$

The abstract interpreter uses a *global state*  $Q$  of the method execution. Given a bytecode with  $n$  instructions,  $Q$  is the sequence of the states  $Q_i$ , one for every instruction  $i$  of the bytecode. Moreover,  $Q$  includes also a special state, named  $Q_f$ , that does not correspond to any instruction, but represents the final state reached after the execution of the last instruction of the method.

The domain of global states contains a special state *error*. Global states are partially ordered according to the relation on the state of every instruction:  $Q \sqsubseteq Q'$  iff  $\forall i, Q_i \sqsubseteq Q'_i$ . Moreover, *error* is the top element of the domain of global states:  $Q \sqsubseteq \text{error}$  for every  $Q$ .

The abstract interpreter is based on a set of rules: there is a rule for each kind of instruction. The interpreter builds a chain of global states  $\{Q^j\}$ , where each state is obtained by the preceding one by means of the application of a rule. Figure 4 shows the rules of JBIFV. In the same figure, the shorthand  $Q_r := Q_r \sqcup (\dots)$  is used. It means that the global state  $Q$  is the same except that for state  $Q_r$  (corresponding to instruction  $r$ ), which is changed as specified by the right hand side of the  $:=$  operator.

The abstract execution is defined by starting from an initial global state that reflects the state of the JVM on method entrance. Given a method  $C.mt$ , the initial global state  $Q^0$  is the sequence of the initial state  $Q_i^0 = (M_i^0, S_i^0, E_i^0)$  of every instruction  $i$ , which is defined as follows. If  $i \neq 0$ , i.e.  $i$  is not the first instruction:

- $M_i^0(x) = \sigma_0$  for each register  $x$   
the contents of the registers is set to the minimum secrecy level;
- $S_i^0 = \perp_S$   
the stack is set to the bottom element of the domain of stacks;
- $E_i^0 = \sigma_0$   
the security environment is set to the minimum one.

$$\begin{array}{l}
\text{pop} \quad \frac{c[i] = \text{pop}, \quad Q_i = (M, k \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, S, E)} \\
\\
\text{dup} \quad \frac{c[i] = \text{dup} \quad Q_i = (M, k \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, (k \sqcup E) \cdot (k \sqcup E) \cdot S, E)} \\
\\
\text{op} \quad \frac{c[i] = \alpha \text{op} \quad Q_i = (M, k_1 \cdot k_2 \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, (k_1 \sqcup k_2 \sqcup E) \cdot S, E)} \\
\\
\text{load} \quad \frac{c[i] = \alpha \text{load } x \quad Q_i = (M, S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, (M(x) \sqcup E) \cdot S, E)} \\
\\
\text{store} \quad \frac{c[i] = \alpha \text{store } x \quad Q_i = (M, k \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M[(k \sqcup E)/x], S, E)} \\
\\
\text{const} \quad \frac{c[i] = \alpha \text{const } d \quad Q_i = (M, S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, E \cdot S, E)} \\
\\
\text{if}_{\in \text{IF}} \quad \frac{c[i] = \text{ifcond } j, i \in IF, \quad Q_i = (M, k \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, S, (k \sqcup E)); \\ Q_j := Q_j \sqcup_j (M, S, (k \sqcup E))} \\
\\
\text{if}_{\notin \text{IF}} \quad \frac{c[i] = \text{ifcond } j, i \notin IF, \quad Q_i = (M, k \cdot S, E), Q_{ipd(i)} = (M', S', E')}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, S, (k \sqcup E)); \\ Q_j := Q_j \sqcup_j (M, S, (k \sqcup E)); \\ Q_{ipd(i)} := (M', S', E)} \\
\\
\text{goto} \quad \frac{c[i] = \text{goto } j}{Q_j := Q_j \sqcup_j Q_i} \\
\\
\text{getfield} \quad \frac{c[i] = \text{getfield } C1.f \quad Q_i = (M, r \cdot S, E)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, (S(C1) \sqcup E \sqcup r) \cdot S, E)} \\
\\
\text{putfield} \quad \frac{c[i] = \text{putfield } C1.f \quad Q_i = (M, k \cdot r \cdot S, E), \quad k \sqcup r \sqcup E \sqsubseteq S(C1)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, S, E)} \\
\\
\text{invoke} \quad \frac{c[i] = \text{invoke } C1.mt1 \quad Q_i = (M, k \cdot r \cdot S, E) \quad k \sqcup E \sqsubseteq \mathcal{P}(C1.mt1), r \sqcup E \sqsubseteq S(C1)}{Q_{i+1} := Q_{i+1} \sqcup_{i+1} (M, (\mathcal{R}(C1.mt1) \sqcup E \sqcup r) \cdot S, E)} \\
\\
\text{return} \quad \frac{c[i] = \alpha \text{return} \quad Q_i = (M, k \cdot S, E) \quad (k \sqcup E) \sqsubseteq \mathcal{R}(C.mt)}{Q_f := Q_f \sqcup_f (M, S, E)} \\
\\
\text{putfield\_err} \quad \frac{c[i] = \text{putfield } C1.f \quad Q_i = (M, k \cdot r \cdot S, E), \quad k \sqcup r \sqcup E \not\sqsubseteq S(C1)}{Q := \text{error}}
\end{array}$$

Figure 4: Rules of the secure information flow verification of method  $C.mt$

The initial state of the first instruction differs from the definition above for the initialization of registers  $x_0$  and  $x_1$ , of the security environment, and of the stack:

- $M_0^0(x_0) = \mathcal{S}(C)$   
 $x_0$  (containing the reference to the object in the concrete semantics) is initialized with the level specified for the class to which the object belongs to;
- $M_0^0(x_1) = \mathcal{P}(C.mt)$   
 $x_1$  (containing the parameter in the concrete semantics) is initialized as specified for the parameter of  $C.mt$ ;
- $S_0^0 = \lambda$   
the stack is empty;
- $E_0^0 = \mathcal{S}(C)$   
the environment is initialized to the level specified for the class to which the object belongs.

When the rule for instruction  $i$  is applied,

1. the state *after*  $i$  is calculated, abstractly executing  $i$  in state  $Q_i$ . For example, if  $i$ : `load x` is executed, and  $Q_i = (M, S, E)$ , the state  $\bar{Q}_i$  after  $i$  has the same memory and the same environment of  $Q_i$ , and the least upper bound between the contents of  $x$  in  $M$  and the environment  $E$  is pushed onto  $S$ ;
2. the state  $\bar{Q}_i$  after  $i$  is *merged* with the state of all successive instructions of  $i$ . Let  $j$  be a successive instruction of  $i$ . The *merge* is performed by a least upper bound calculation between the original value of  $Q_j$  and the state  $\bar{Q}_i$ . For example, the state after  $i$ : `load x` is merged with  $Q_{i+1}$ .

As a consequence, if an instruction  $i$  has several predecessor instructions,  $Q_i$  is the *least upper bound* of the states after all the preceding instructions.

The rules for the instruction **if** need some explanation. When the instruction  $i$ : `if j` is executed

1. the environment of the two successive instructions ( $i+1$  and  $j$ ) is upgraded to take into account the secrecy level of the condition tested by the instruction (the top of the stack);
2. the environment of state  $Q_{ipd(i)}$  is set to the environment of  $Q_i$ , as the implicit flow caused by  $i$  terminates at  $ipd(i)$ . However, this must not be done for every **if** instruction: when we have nested **if** instructions sharing the same **ipd**, the instruction at address **ipd** must be executed under the security environment of the outermost **if**. For example, suppose  $ipd(i) = ipd(i')$ , where  $i$  and  $i'$  are addresses of two nested **if** instructions. If  $i$  is the outermost **if**, the environment of state  $Q_{ipd(i)}$  must be updated when the instruction at address  $i$  is executed. Instead, the execution of the **if** instruction at address  $i'$  does not change state

$Q_{ipd(i')} = Q_{ipd(i)}$ . To this purpose, we define  $IF$  as the set containing the address of nested **if** instructions which share the **ipd** with the corresponding outermost **if**. There are two rules for **if j** in Figure 4. If  $i \in IF$ , then rule *if<sub>IF</sub>* is applied, updating only  $Q_{i+1}$  and  $Q_j$ . If  $i \notin IF$ , the other rule is applied. The rule also modifies the environment of  $Q_{ipd(i)}$ .

Finally, for every instruction  $j$ , such that  $j = ipd(i)$  and  $i$ : **if**, the execution of an instruction preceding  $j$  must not modify the environment of state  $Q_j$ , since this environment is set when the interpreter executes  $i$ . The restricted version of the upper bound operation  $\sqcup_{M,S}$  is used in this case by the interpreter. In the rules, we use the notation  $\sqcup_r$ , which is a shorthand to avoid the duplication of rules:

$$\sqcup_r = \begin{cases} \sqcup_{M,S} & \text{if } \exists j \mid r = ipd(j) \\ \sqcup & \text{otherwise} \end{cases}$$

The rule for **getfield C1.f** pushes onto the stack the least upper bound between the secrecy level of **C1** (which is also the secrecy level of **C1.f**), the secrecy level of the reference to **C1** (which may be different from  $\mathcal{S}(C1)$ ) and the environment.

The abstract interpreter performs a standard fixpoint iteration: the transition functions are applied according to the rules in Figure 4. A violation of secure information flow may be discovered when applying the rules for **putfield**, **invoke** and **return**:

- **putfield C1.f**  
it is checked that the level of the value to be written in **C1.f** is less than or equal to the secrecy level of class **C1**. This level is the least upper bound among the top of the stack, the level of the reference to **C1** and the environment;
- **invoke C1.mt1**  
it is checked that the least upper bound between the level of the parameter (the top of the stack) and the environment is less than or equal to  $\mathcal{P}(C1.mt1)$ , and the least upper bound between the level of the reference and the environment is less than or equal to  $\mathcal{S}(C1)$ . In the state after this instruction, the return value of **C1.mt1** is pushed onto the stack. It is the least upper bound among the level of the reference, the environment and the level as specified by  $\mathcal{R}(C1.mt1)$ ;
- **return**  
it is checked that the least upper bound between the level of the returned value (the top of the stack) and the environment is less than or equal to the level specified by  $\mathcal{R}(C.mt)$ .

We assume that, if an error is discovered, the corresponding global state is set to *error*, which is the topmost element of the domain of global states. This means that a second rule must be added for **putfield**, **invoke** and **return**. In Figure 4 we show only the rule for **putfield**.

	$(M(x_0), M(x_1))$	Stack	Env.
$Q_0$	$(l, h)$	$\lambda$	$l$
$Q_1$	$(l, h)$	$(l)$	$l$
$Q_2$	$(l, h)$	$(hl)$	$l$
$Q_3$	$(l, h)$	$(hl)$	$l$
$Q_4$	$(l, h)$	$(l)$	$h$
$Q_5$	$(l, h)$	$(hl)$	$h$
$Q_6$	$(l, h)$	$(l)$	$h$
$Q_7$	$(l, h)$	$(hl)$	$l$
$Q_8$	$(l, l)$	$\perp_S$	$l$
$Q_9$	$(l, l)$	$\perp_S$	$l$
$Q_f$	$(l, l)$	$\perp_S$	$l$

**Figure 5: An example**

In the formulation of the following two propositions, we assume a set  $\mathcal{C}$  of class definitions, and a security specification  $\mathcal{S} : \mathcal{C} \rightarrow \Sigma$  for the classes and for the methods' parameter and result  $\mathcal{P}, \mathcal{R} : \text{Mtds}(\mathcal{C}) \rightarrow \Sigma$ .

**PROPOSITION 1.** *Consider a method  $C.mt$  in  $\text{Mtds}(\mathcal{C})$ . Consider a corresponding chain  $\{Q^j\}$  where, for each  $0 \leq j$ ,  $Q^{j+1}$  is obtained by  $Q^j$  by means of the application of one of the rules in Figure 4.*

1. For each  $j$ ,  $Q^j \sqsubseteq Q^{j+1}$
2.  $\exists n$  such that  $\sqcup\{Q^j\} = Q^n$
3. If  $\{Q^r\}$  is another chain obtained by a different order of application of the rules,  $\sqcup\{Q^r\} = \sqcup\{Q^j\}$

The above proposition means that the fixpoint of the iteration process is reached with a finite number of iterations and, moreover, it does not depend on the order of application of the rules. The following proposition states the correctness of the method.

**PROPOSITION 2.** *Consider a method  $C.mt \in \text{Mtds}(\mathcal{C})$  and a corresponding chain  $\{Q^j\}$ . If  $\sqcup\{Q^j\} \neq \text{error}$ , then  $C.mt$  is  $\mathcal{S}, \mathcal{R}, \mathcal{P}$ -secure.*

**Proof sketch.** *The proof is based on the work of some of the authors and others [5, 6, 7]: there, an abstract semantics for the bytecode, similar to the one used in the present paper, is defined as an abstraction of a concrete semantics, annotating data with security information and including checks on violation of secure information flow.*

Figure 5 shows the application of the abstract interpreter to the example of Figure 3. We assume two secrecy levels  $\Sigma = \{l, h\}$  with  $l \sqsubset h$  ( $l$  is the secrecy level for public data and  $h$  is the secrecy level for private data). Moreover,  $\mathcal{S}(A) = l$ ,  $\mathcal{S}(B) = h$ ,  $\mathcal{P}(A.mt) = h$  and  $\mathcal{R}(A.mt) = l$ . It is  $IF = \{\}$  and  $ipd(3) = 7$ . The figure shows the global state of the interpreter before the application of the rule for 7: **putfield A.f1**. An error is discovered since  $k \sqcup r \sqcup E \not\sqsubseteq \mathcal{S}(A) : h \sqcup l \sqcup l \not\sqsubseteq l$ .

## 5. THE TOOL

Based on the method described above, we are developing a prototype tool that performs the abstract execution of the subset of JVMIL shown in Figure 1. The tool examines a class file in a method-by-method manner. The bytecode which the tool is applied to must be type-safe, i.e., it already passed the standard verification. The tool is written in Java, and is composed of the following building blocks:

**parser:** reads information from the class file to be examined. For each method, gets the bytecode, the number of memory registers, and the stack size.

**levelManager:** asks the user to assign security levels to the class and, for each method, to the parameters and to the return value, if any.

**ipdManager:** builds the *ipd* table.

**abstractInterpreter:** implements the core interpreter as described below. For each instruction of type **putfield**, **invoke** and **return**, visualizes the actual and the required security levels of information flows.

Prior to analyzing a method's bytecode, the following operations are performed:

- The instructions belonging to the method are loaded in a byte array.
- The state of the instructions in the global state are initialized as described in Section 4.1.
- A table is built that contains, for each conditional branch, the address of the corresponding *ipd* instruction. The set *IF* is calculated.

When analyzing a method's bytecode, the rules are applied over the instructions. In the implementation of the fixpoint iteration, the order of application of rules follows the order of the control flow graph. The execution starts from the first instruction, i.e. the instruction at address 0 and continues until either a **return** instruction is reached or the program counter points to an already executed instruction whose state has not changed after the execution of the preceding instruction. For each instruction, the following actions are taken:

1. build the state *after* the execution of the instruction as specified by the corresponding rule;
2. if the instruction under execution is a **putfield** or an **invoke** or a **return** instruction, check if an error occurs by comparing the actual security levels with the one required by the premise of the rule;
3. using the state built at step 1, change the state of the next instructions, if any. All instructions have one successor, except for conditional branches that have two, and **return** that has none;

4. if the instruction is a conditional branch, at address  $i$ , recursively analyze the bytecode belonging to both branches. Each branch starts at one of the two successors' instructions and terminates at  $ipd(i)$ . When the analysis of both branches is complete, the environment of  $ipd(i)$  is set to that of  $i$ ;
5. set the program counter to the address of the next instruction. In the case of conditional branches, the next instruction is the  $ipd$  instruction, since both branches have been already analyzed. The **return** instruction has no successor and the program counter is set to null.

## 6. RELATED WORK AND CONCLUSIONS

Works addressing secure information flow for stack based languages are [6, 8, 7, 9]. In [6, 8] some of the authors and others define an approach based on abstract interpretation of the operational semantics of the language: values are abstracted onto secrecy levels and a small step abstract semantics is presented, producing a transition system representing all possible executions. In these works objects and method calls are not handled. In [7, 9] a combination of abstract interpretation and model checking is used: the abstract transition system is model checked against a set of logic formulas expressing the secure information flow property (and other security properties). The work [9] concerns secure information flow in Java smart cards.

The main advantage of the method proposed in the paper is that the verification can be performed inside the Java environment, and does not rely on any extra verification tool, as it happens with methods based on model checking.

As a future work, we intend to complete the prototype tool to cover the full JVM. Moreover, we are investigating the feasibility of applying the proposed method to the verification of the secure information flow among applets in Java smart cards [13, 16].

## 7. REFERENCES

- [1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Proceedings*, pages 147–160. Texas, Usa, 1999.
- [2] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, 1980.
- [3] T. Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Program. Lang. Syst.*, 2(1-4):1–16, 1993.
- [4] J. Banatre, C. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.
- [5] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Abstract interpretation of operational semantics for secure information flow. *Information Processing Letters*, 83(2):101–108, 2002.
- [6] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of java bytecode by abstract interpretation. In *The 17th ACM Symposium on Applied Computing: Special Track on Computer Security Proceedings*. Madrid, March 2002.
- [7] C. Bernardeschi and N. D. Francesco. Combining abstract interpretation and model checking for analysing security properties of java bytecode. In *Third International Workshop on Verification, Model Checking and Abstract Interpretation Proceedings*, pages 1–15. LNCS 2294, Venice, January 2002.
- [8] C. Bernardeschi, N. D. Francesco, and G. Lettieri. An abstract semantics tool for secure information flow of stack-based assembly programs. *Microprocessors and Microsystems*, 26(8):391–398, 2002.
- [9] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS 2000 Proceedings*, 2000.
- [10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.
- [11] X. Leroy. Java bytecode verification: an overview. In *13th International Conference on Computer Aided Verification, LNCS 2102, Proceedings*, pages 265–285, July 2001.
- [12] M. Mizuno and D. A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.
- [13] J. Possegga and H. Vogt. Bytecode verification for java smartcards based on model checking. In *ESORICS 98 Proceedings*, 1998.
- [14] F. Pottier and S. Conchon. Information flow inference for free. In *ACM ICFP'00 Proceedings*, pages 46–57, 2000.
- [15] Z. Qian. Standard fixpoint iteration for java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.
- [16] E. Rose and K. Rose. Lightweight bytecode verification. In *WFUJ 98 Proceedings*, 1998.
- [17] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *LNCS*, 1576:40–58, 1996.
- [18] L. T. and F. Yellin. *The Java virtual machine specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.
- [19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
- [20] S. Zdancewic and A. Myers. Secure information flow and cps. *LNCS*, 2028:46–61, 2001.