Checking Security of Java Bytecode by Abstract Interpretation

Roberto Barbuti
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56100 Pisa, Italy
barbuti@di.unipi.it

Cinzia Bernardeschi
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Via Diotisalvi 2
56100 Pisa, Italy
c.bernardeschi@iet.unipi.it

Nicoletta De Francesco
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Via Diotisalvi 2
56100 Pisa, Italy
n.defrancesco@iet.unipi.it

ABSTRACT

We present a method to certify a subset of the Java bytecode, with respect to security. The method is based on abstract interpretation of the operational semantics of the language. We define a concrete small-step enhanced semantics of the language, able to keep information on the flow of data and control during execution. A main point of this semantics is the handling of the influence of the information flow on the operand stack. We then define an abstract semantics, keeping only the security information and forgetting the actual values. This semantics can be used as a static analysis tool to check security of programs. The use of abstract interpretation allows, on one side, being semantics based, to accept as secure a wide class of programs, and, on the other side, being rule based, to be fully automated.

Keywords

Security, Information Flow, Java bytecode, Abstract Interpretation

1. INTRODUCTION

The problem of security leakages for Java bytecode is of great importance since programs compiled into the Java Virtual Machine bytecode language, JVML [21], can be downloaded by the Internet and executed by a Java-compatible web browser. Assume that a downloaded program needs to access to the user private data to compute some information. If the program also needs to access over the Internet, the private data could be leaked. We want to be sure that, given a program that accesses private information, it will be unable to leak such data. Of course, users have the option to forbid downloaded code from accessing any local file, but useful programs need to access to private files in order to perform their tasks.

SAC 2002, Madrid, Spain

The classical formulation of the *secure information flow* property [10, 11, 2] requires that information at a given security level does not flow to lower levels. The problem has been extensively studied for programs written in structured high level languages. Given a program in which every variable is assigned a security level, it has secure information flow if, when the program terminates, the value of each variable does not depend on the initial value of the variables with higher security level. Let us suppose that variable `y` has security level higher than that of variable `x`. Examples of violation of secure information flow in high level languages are: `x:=y` and `if y=0 then x:=1 else x:=0`. In the first case, there is an *explicit* information flow from `y` to `x`, while, in the second case there is an *implicit* information flow: in both cases, checking the final value of `x` reveals information on the value of the higher security variable `y`. Other security leakages occur when high level information is revealed not only by the value of the variables, but by the behavior of the program [22]. Consider the program `while (y > 0) do skip`. This is an example of security leakage due to nontermination, since it loops indefinitely when the high security variable is greater than zero.

JVML is a stack based assembly language. When considering information flow in assembly code, some specific aspects must be taken into account:

- *explicit flow.* In a high level language *explicit flow* of information occurs with the assignment statement. Thus, the security level of the flowing information can be deduced by the expression on the right-hand side of the assignment. In machine code, instead, values are pushed onto and popped off the stack, and a pop instruction does not syntactically reveal the source of the value.

- *implicit flow.* In high level languages, the scope of the *implicit flow* caused by the condition of conditional or repetitive commands can be easily derived, since it coincides with the scope of the command itself. Since assembly languages are unstructured, jumps may go to any program point, making more complicated to find the scope of implicit flows. Moreover, also the operand stack is influenced by the implicit flow, because the stack may be manipulated in different ways by the branches of a branching instruction: they can perform a different number of pop and push operations, and

with a different order.

- *machine state.* Since an assembly language defines an abstract machine on which programs are executed, also modifications of the state of the machine caused by high level information must be taken into account, like as the contents of the operand stack and the contents of the program counter when the program terminates.

We present a method to certify security of programs in a subset of JVML. The method is based on abstract interpretation of operational semantics [8, 9, 19]. We define a concrete operational semantics of the language, able to keep information flow during execution. The basic ideas on which the semantics is based are: i) values carry a security level which changes dynamically, depending on how the values are manipulated, and ii) implicit flow is modeled by an environment, which records the security level of the implicit flow. The environment is updated when an implicit flow begins (when a branching instruction is executed), and it is restored on termination of the implicit flow. A main point is handling this environment, and in particular its influence on the operand stack. We then introduce an abstract operational semantics that disregards the numerical part of the values, and operates only on their security levels. We show that the abstract semantics can be used to check security of programs. The use of abstract interpretation allows, on one side, being semantics based, to accept as secure a wide class of programs, and, on the other side, being rule based, to be fully automated.

The remainder of the paper is organised as follows: Section 2 presents the language. Section 3 introduces the security model. Section 4 and 5 define the concrete and abstract semantics, respectively. Section 6 compares our approach with other ones. The proofs of theorems are only sketched. The complete proofs can be found in [7].

2. THE LANGUAGE

This section presents the language and its semantics. With N we denote the natural numbers. Given a set A, $A^\star$ denotes the set of finite sequences of elements of A; λ indicates the empty sequence; if w is a finite sequence, $\sharp w$ denotes the length of w, i.e. the number of elements of w; $\cdot$ denotes both the concatenation of a value to a sequence and the standard concatenation operation between sequences, i.e. if $w, u \in A^\star$ and $k \in A$, $k \cdot w$ is the sequence obtained prepending k to w, and $w \cdot u$ is obtained by appending u to w; finally, if $i \in \{1, \ldots, \sharp w\}$, with $w[i]$ we denote the i-*th* element of w, i.e. if $w = w_1 \cdots w_n$, $w[i] = w_i$. We represent stacks by sequences, with the convention that, if w is a nonempty stack, $w[1]$ is the top element.

Our language is the subset of JVML [21] called JVML0 in [20]. It has an operand *stack*, a *memory* containing the local variables, simple arithmetic instructions and conditional/unconditional jumps. The instructions are reported in Figure 1, where x ranges over a set *var* of *local variables* and op over a set of binary arithmetic operations (`add`, `sub`, ..). Note that the language supports subroutine calls via the `jsr` j and `ret` x instructions.

A program is a sequence c of instructions, numbered starting from address 1; $\forall i \in \{1, \cdots, \sharp c\}$, $c[i]$ is the instruction at address i. In the following, given a sequence of instructions c, we denote by $Var(c)$ the variable names occurring in c. We assume that programs respect the following static constraints, generally checkable using the code verifier: no stack overflow and underflow occur, and executions will not jump to undefined addresses.

`op`	pop two operands off the stack, perform the operation, and push the result onto the stack
`pop`	discard the top value from the stack
`push` k	push the constant k onto the stack
`load` x	push the value of the variable x onto the stack
`store` x	pop off the stack and store the value into x
`if` j	pop off the stack and jump to j if non-zero
`goto` j	jump to j
`jsr` j	at address p, jump to address j and push return address $p+1$ onto the operand stack
`ret` x	jump to the address stored in x
`halt`	stop

Figure 1: Instruction set.

We give the semantics of the program in terms of a transition system, whose paths represent execution traces and whose nodes display the program's changing states.

Definition 1. A transition system T is a triple $(S, \rightarrow, s_0)$, where S is a set of states, $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation. We say that there is a transition from s to s' if and only if $(s, s') \in \rightarrow$, usually we write $s \rightarrow s'$. We denote by $\stackrel{*}{\longrightarrow}$ the reflexive and transitive closure of $\longrightarrow$. Moreover, we say that $s \in S$ is a final state of the transition system (denoted by $s \not\rightarrow$) if and only if no s' exists such that $s \rightarrow s'$.

The *standard semantics* of the language, defined as a set of inference rules, is reported in Figure 2. The semantics uses a domain $\mathcal{V}^e$ of constant values, ranged over by $k, k_1, k_2, \cdots$, including both data and addresses (ranged over by $i, j, \cdots$); a domain of functions $\mathcal{M}^e : var \rightarrow \mathcal{V}^e$ of memories from variable identifiers to values, ranged over by $m, m', m_1, \cdots$ and a domain $\mathcal{S}^e = (\mathcal{V}^e)^\star$ (finite sequences over $\mathcal{V}^e$) of stacks, ranged over by $s, s', s_1, \cdots$. In the following, given a memory m, we denote by $D(m) \subseteq var$ the domain of m, i.e. the variables stored in m.

We model a state of the program execution as a tuple, $\langle i, m, s \rangle$, where i is the address held by the program counter, m is the memory representing the current state of the local variables and s is the current state of the operand stack. We assume that a program is always executed starting from the instruction $c[1]$ and with an empty operand stack. We denote as $\mathcal{C}^e$ the domain of states.

The rules of the standard semantics define a relation $\longrightarrow^e \subseteq \mathcal{C}^e \times \mathcal{C}^e$. The notation $m[k \ / \ x]$ is used to indicate the memory m' which agrees with m for all variables, except for x, for which it is $m'(x) = k$.

Given a program c and a memory $m \in \mathcal{M}^e$, the standard semantics of the program is the transition system defined as $(\mathcal{C}^e, \longrightarrow^e, \langle 1, m, \lambda \rangle)$, where the initial state consists of the address of the first instruction, the given memory and the empty operand stack. Since the program is deterministic, the corresponding transition system has only one, possibly infinite, path. The final state, if it exists, is unique and it has the form $\langle i, m', s \rangle$ with $c[i] =$ `halt`, for some s and m'. We do not require that the operand stack is empty on program termination.

$$\textbf{op}\quad \frac{c[i] = \texttt{op}}{\langle i, m, k_1 \cdot k_2 \cdot s\rangle \longrightarrow^{e} \langle i+1, m, (k_1\ op\ k_2)\cdot s\rangle}$$

$$\textbf{pop}\quad \frac{c[i] = \texttt{pop}}{\langle i, m, k \cdot s\rangle \longrightarrow^{e} \langle i+1, m, s\rangle}$$

$$\textbf{push}\quad \frac{c[i] = \texttt{push k}}{\langle i, m, s\rangle \longrightarrow^{e} \langle i+1, m, k \cdot s\rangle}$$

$$\textbf{load}\quad \frac{c[i] = \texttt{load x} \qquad m(\mathtt{x}) = k}{\langle i, m, s\rangle \longrightarrow^{e} \langle i+1, m, k \cdot s\rangle}$$

$$\textbf{store}\quad \frac{c[i] = \texttt{store x}}{\langle i, m, k \cdot s\rangle \longrightarrow^{e} \langle i+1, m[k\ /\ \mathtt{x}], s\rangle}$$

$$\textbf{if}_{false}\quad \frac{c[i] = \texttt{if j}}{\langle i, m, 0 \cdot s\rangle \longrightarrow^{e} \langle i+1, m, s\rangle}$$

$$\textbf{if}_{true}\quad \frac{c[i] = \texttt{if j}}{\langle i, m, k \neq 0 \cdot s\rangle \longrightarrow^{e} \langle j, m, s\rangle}$$

$$\textbf{goto}\quad \frac{c[i] = \texttt{goto j}}{\langle i, m, s\rangle \longrightarrow^{e} \langle j, m, s\rangle}$$

$$\textbf{jsr}\quad \frac{c[i] = \texttt{jsr j}}{\langle i, m, s\rangle \longrightarrow^{e} \langle j, m, (i+1)\cdot s\rangle}$$

$$\textbf{ret}\quad \frac{c[i] = \texttt{ret x}}{\langle i, m, s\rangle \longrightarrow^{e} \langle m(x), m, s\rangle}$$

Figure 2: Standard semantics rules.

We now recall the notion of control flow graph of a program, containing the control dependencies among the instructions of the program, and the notion of postdomination and immediate postdomination in directed graphs [4].

Definition 2. Given a program c composed of n instructions ($\sharp c = n$), the *control flow graph* of the program is the directed graph (V, E), where $V = \{1, .., n+1\}$ is the set of nodes and $E \subseteq V \times V$ contains the edge (i, j) if and only if the instruction at address j can be immediately executed after that at address i; moreover it contains the edge $(i, n+1)$ from each address i such that $c[i] = \texttt{halt}$.

Definition 3. Let i and j be nodes of a control flow graph of a program with n instructions. We say that node j *postdominates* i, denoted by $j\ pd\ i$, if $j \neq i$ and j is on every path starting from i. We say that node j *immediate postdominates* i, denoted by $j\ ipd\ i$, if $j\ pd\ i$ and there is no node r such that $j\ pd\ r\ pd\ i$. We also use the notation $j = ipd(i)$. For each node $i \neq n+1$ such that $ipd(i)$ does not exists, we put $ipd(i) = n+1$.

The control flow graph is built by statically examining the program. It has one and only one initial node, the node 1, and one and only one final node, $n+1$. Moreover, $ipd(i)$ exists for each node $i \in \{1, \cdots, n\}$. We use the control flow graph and the notion of immediate postdomination to handle implicit information flows. Each branching instruction `if` j or `ret` x causes the beginning of an *implicit flow*: if the instruction is at address i, then the implicit flow affects all instructions belonging to a path from i to $ipd(i)$; $ipd(i)$ is the first instruction not affected by the implicit flow, since it represents the point in which the different branches join. Note that $ipd(i) = n+1$ for each node i belonging to a cycle. Thus the implicit flow holding on entering a cycle affects all successive instructions.

3. THE SECURITY MODEL

We assume a finite lattice $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set of security levels, ranged over by $\sigma, \tau, ..$, partially ordered by $\sqsubseteq$. Given $\sigma, \tau \in \mathcal{L}$, $\sigma \sqcup \tau$ denotes the least upper bound of σ and τ; $\sigma \sqsubset \tau$ (or $\tau \sqsupset \sigma$) means $\sigma \sqsubseteq \tau$ and $\sigma \neq \tau$.

To model security, we consider annotated programs, where each variable is associated with a security level. A program P is a pair (c, Λ) where c is a sequence of instructions, and Λ is a partition of the variables in $Var(c)$: $\Lambda = \{\Lambda_\sigma | \sigma \in \mathcal{L}\}$ where $\forall \sigma \in \mathcal{L} :\ \Lambda_\sigma \subseteq Var(c)$ is the set of variables with security level σ.

Given a program $P = (c, \Lambda)$ and $\sigma \in \mathcal{L}$, we denote by $\Lambda_{\sqsubseteq\sigma} = \cup_{\tau \sqsubseteq \sigma} \Lambda_\tau$ the variables of P with level $\sqsubseteq \sigma$ and by $\Lambda_{\not\sqsubseteq\sigma}$ the other variables.

The notion of security we are going to introduce, denoted as σ-security, is parametric with respect to a security level σ, and describes the fact that information with security level $\not\sqsubseteq \sigma$ is kept secret. σ-security guarantees the absence of different possible leakages: it assures that the initial values of the variables in $\Lambda_{\not\sqsubseteq\sigma}$ do not affect the final value of the variables in $\Lambda_{\sqsubseteq\sigma}$ and that information on such values cannot be retrieved in the machine state looking at the operand stack elements or at the address of the last instruction executed. Moreover, it guarantees that high level information is not revealed by observation on the termination of the program.

Definition 4. Let $P = (c, \Lambda)$ and $\sigma \in \mathcal{L}$. We say that P is σ-secure if, for each assignment of values to the variables in $\Lambda_{\sqsubseteq\sigma}$, it holds that: for each pair of memories m_1 and m_2, such that $D(m_1) = D(m_2) = Var(c)$ and $\forall x \in \Lambda_{\sqsubseteq\sigma}$: $m_1(x) = m_2(x)$,
$\langle 1, m_1, \lambda\rangle \overset{*}{\longrightarrow}^{e} \langle i, m_1', s\rangle \not\rightarrow$
implies
$\exists m_2'$ such that $\langle 1, m_2, \lambda\rangle \overset{*}{\longrightarrow}^{e} \langle i, m_2', s\rangle \not\rightarrow$ with $\forall x \in \Lambda_{\sqsubseteq\sigma}$: $m_1'(x) = m_2'(x)$

The notion of σ-security does not exclude that the final value of a variable $x \in \Lambda_{\sqsubseteq\sigma}$ is influenced by the value of some other variable $y \in \Lambda_{\sqsubseteq\sigma}$ with security level higher than that of x. In fact σ-security partitions the security levels into two groups: those lower than or equal to σ, and those higher than or not related to σ, and ensures only that there is no flow of information from the second group to the first one, while variables in $\Lambda_{\sqsubseteq\sigma}$ can depend from each other in any way. As a consequence, σ-security does not guarantee σ'−security for $\sigma' \sqsubset \sigma$.

Let $(\mathcal{L}, \sqsubseteq)$, with $\mathcal{L} = \{\sigma, \tau\}$ and $\sigma \sqsubset \tau$. Assume $y \in \Lambda_\tau$ and $x \in \Lambda_\sigma$. Examples of non-σ-secure programs are shown in figure 3. Program 3 (a) corresponds to `if y=0 then x:=1 else x:=0`. Program 3(b) shows an assembly code which jumps to different points depending on the value of y (instruction `ret y` at address 7), and then x is assigned a different value. The program in Figure 3(c) terminates or does not terminate depending on the value zero or non-zero of `y`. Figures 3(d) (3(e)) reports a program which terminates with different stacks (respectively at different program points) according to the value zero or non-zero of `y`.

4. THE CONCRETE SEMANTICS

This section presents an enhanced concrete operational semantics of the language in Figure 1, able to check violations of σ-security in a program. The semantics

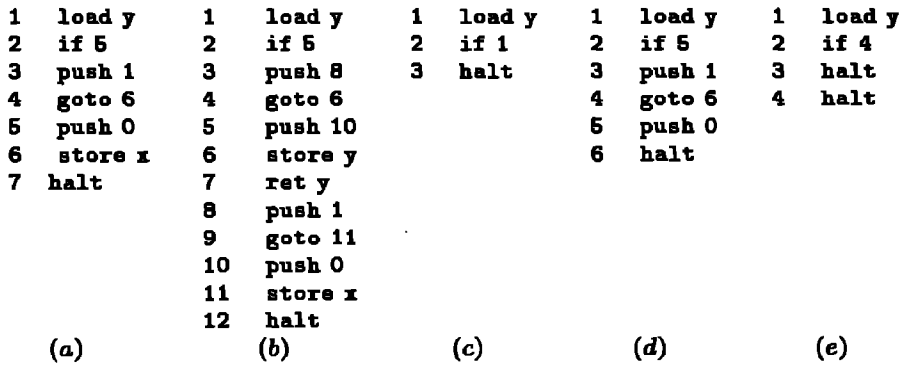

Figure 3: Not σ-secure programs.

- Handles values enriched with a security level. During the execution of a program, the security level of a value indicates the least upper bound of the security levels of the information flows, both explicit and implicit, on which the value depends.

- Executes instructions under a *security environment*, which is a security level. At each moment during the execution, the security environment represents the least upper bound of the security levels of the open implicit flows. The security environment can be upgraded by a branching instruction (`if` and `ret`) and can be downgraded when an implicit flow terminates.

The semantics uses the control flow graph of a program to handle implicit flows (see section 2). The implicit flow of an `if` or `ret` instruction at address i terminates at the instruction with address $ipd(i)$. On executing the `if` or `ret` instruction within a security environment σ, the semantics records the pair $(ipd(i), \sigma)$, and then the security environment is (possibly) upgraded to consider: i) for the `if` instruction, the security level of the condition, i.e. the top value on the operand stack and ii) for the `ret` x instruction, the security level of the address stored into `x`. When the implicit information flow terminates, i.e. when the instruction at address $ipd(i)$ is reached, the environment is reset to σ, i.e. the environment holding when the instruction which originated the implicit flow was executed. To model nested branching instructions, the semantics records the pairs $(ipd(i), \sigma)$'s in a stack, since, in presence of nested branching instructions, the innermost implicit flow terminates first. The stack is called *ipd* stack.

We now introduce the domains of the concrete semantics. Enriched values are pairs $v = (k, \sigma)$, where $k \in \mathcal{V}^e$ (denoted as the *numerical part* of v), is a constant, and $\sigma \in \mathcal{L}$ is denoted as the *security level* of v. Given $\sigma \in \mathcal{L}$, (k, σ) is called σ-value and $\mathcal{V}_\sigma$ is the set of σ-values. $\mathcal{V} = (\mathcal{V}^e \times \mathcal{L})$ is the domain of concrete values, ranged over by $v, v', v_1, \cdots$. $\mathcal{M} : var \rightarrow \mathcal{V}$ is the domain of concrete memories, ranged over by $M, M', M_1, \cdots$. $\mathcal{S} = \mathcal{V}^*$ are the concrete operand stacks, ranged over by $S, S', S_1, \cdots$, and $\mathcal{R} = (N \times \mathcal{L})^*$, ranged over by $\rho, \rho', \cdots$ are the ipd stacks. Given $M \in \mathcal{M}$, $D(M)$ denotes the domain of M.

The rules of the operational semantics are shown in Figure 4. The rules define a relation $\longrightarrow \subseteq \mathcal{C} \times \mathcal{C}$, where $\mathcal{C}$ is a set of concrete states. Each state has the structure $\sigma \models (i, M, S, \rho)$, where $\sigma \in \mathcal{L}$ is the environment, $i \in \{1, .., \sharp c\}$ is the contents of the program counter, $M \in \mathcal{M}$ is a concrete memory, $S \in \mathcal{S}$ represents the concrete operand stack and $\rho \in \mathcal{R}$ is the *ipd* stack. There is a one to one correspondence between the standard and the concrete semantics rules, except for the **ipd** rule, which is new. When this rule is applicable, no other rule is so. The rules ensure that the elements present in the operand stack have security level higher than or equal to the security level of the environment (see Lemma 1).

To keep the security level of a value equal to the security level of the information on which it depends, the semantics modifies the security level of each value pushed onto the operand stack according to the present environment. Rule **push** assigns to the constant the security level of the environment (a not yet used constant can be thought as having the minimum security level). Rule **load**, if the instruction is `load x`, assigns to the value pushed onto the stack the least upper bound between the security level of $M(x)$ and the environment. The meaning of the rules **op**, **pop**, **store** and **goto** is straightforward. Rule **jsr** associates the return address pushed onto the stack with the security level of the environment.

An implicit flow is entered when an `if` or a `ret` instruction is executed. Consider Rules $\mathbf{if}_{true}$ and $\mathbf{if}_{false}$ applied to an `if` instruction at address i: if the value on top of the operand stack is (k, τ), the selected branch of the instruction is executed under the τ security environment. Moreover $(ipd(i), \sigma)$ is pushed onto the *ipd* stack ($(ipd(i), \sigma) \odot \rho$), from which it will be taken on termination of the implicit flow. We assume that $\odot$ avoids pushing an address i onto the ipd stack if i is already on the top of the stack. In fact, if two or more nested branching instructions terminate at the same ipd, on termination of the implicit flows the environment must be the one holding on entering the outermost one.

The rules for `if` upgrade the security level of each value held by a variable assigned by a `store` instruction in at least one of the two branches: let $W = \{x | c[j] =$ `store` x and j belongs to a path of the control flow graph starting at i and ending at $ipd(i)$, excluding $ipd(i)\}$. For each $x \in W$, if $M(x) = (k, \sigma)$, then $upgrade_M(M, i, \tau)(x) = (k, \sigma \sqcup \tau)$. The security level of the variables not in W is not changed by $upgrade_M(M, i, \tau)$. Upgrading the memory in this way takes into account the fact that a variable may be modified in one branch and not in the other one.

$$\textbf{ipd}\quad \frac{\rho = (i,\tau)\cdot\rho'}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \tau \models \langle i, M, S, \rho'\rangle}$$

$$\textbf{op}\quad \frac{c[i] = \texttt{op} \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, (k_1,\tau_1)\cdot(k_2,\tau_2)\cdot S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, (k_1\ \mathit{op}\ k_2, \tau_1 \sqcup \tau_2)\cdot S, \rho\rangle}$$

$$\textbf{pop}\quad \frac{c[i] = \texttt{pop} \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, (k,\tau)\cdot S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, S, \rho\rangle}$$

$$\textbf{push}\quad \frac{c[i] = \texttt{push}\ k \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, (k,\sigma)\cdot S, \rho\rangle}$$

$$\textbf{load}\quad \frac{c[i] = \texttt{load}\ x \qquad M(x) = (k,\tau) \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M, (k, \tau\sqcup\sigma)\cdot S, \rho\rangle}$$

$$\textbf{store}\quad \frac{c[i] = \texttt{store}\ x \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, (k,\tau)\cdot S, \rho\rangle \longrightarrow \sigma \models \langle i+1, M[(k,\tau)\ /\ x], S, \rho\rangle}$$

$$\textbf{goto}\quad \frac{c[i] = \texttt{goto}\ j \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle j, M, S, \rho\rangle}$$

$$\textbf{if}_{false}\quad \frac{c[i] = \texttt{if}\ j \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, (0,\tau)\cdot S, \rho\rangle \longrightarrow \tau \models \langle i+1, \mathit{upgrade}_M(M,i,\tau), \mathit{upgrade}_S(S,\tau), (\mathit{ipd}(i),\sigma)\odot\rho\rangle}$$

$$\textbf{if}_{true}\quad \frac{c[i] = \texttt{if}\ j \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, (k\neq 0,\tau)\cdot S, \rho\rangle \longrightarrow \tau \models \langle j, \mathit{upgrade}_M(M,i,\tau), \mathit{upgrade}_S(S,\tau), (\mathit{ipd}(i),\sigma)\odot\rho\rangle}$$

$$\textbf{jsr}\quad \frac{c[i] = \texttt{jsr}\ j \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma \models \langle j, M, ((i+1),\sigma)\cdot S, \rho\rangle}$$

$$\textbf{ret}\quad \frac{c[i] = \texttt{ret}\ x \qquad M(\mathtt{x}) = (j,\tau) \qquad i\ \mathit{not\_in}\ \rho}{\sigma \models \langle i, M, S, \rho\rangle \longrightarrow \sigma\sqcup\tau \models \langle j, \mathit{upgrade}_M(M,i,\sigma\sqcup\tau), \mathit{upgrade}_S(S,\sigma\sqcup\tau), (\mathit{ipd}(i),\sigma)\odot\rho\rangle}$$

Figure 4: Concrete semantics rules.

Also the operand stack may be influenced by the implicit flows. The rules in fact modify also the security level of each value present in the operand stack by applying the function $\mathit{upgrade}_S(S,\tau)$: it upgrades the security level of each value v in S to the least upper bound of τ and the security level of v. The `ret` x instruction, handled similarly, sets the new environment to the least upper bound between the present one and the security level of x, and upgrades the memory and the stack accordingly.

Upgrading the operand stack on entering an implicit flow is an abstraction to take into account the fact that the stack may be manipulated in different ways by different branches. The branches can perform a different number of pop and push operations, and with a different order. Thus also elements used in no branch may change their position within the stack, due to the implicit flow. These elements can be used after the termination of the implicit flow, i.e. when the previous environment is restored. Thus they must record the fact that they were affected by the implicit flow. Our choice is to upgrade all elements of the stack on entering an implicit flow. As an example, consider the program in Figure 5. When instruction $c[6]$ = `store x` is executed, the top of the stack is 0 or 1, depending on the branch chosen at the `if` instruction at address 4, testing the value of the high variable `y`. On the other hand, instruction $c[6]$ does not belong to the scope of the high implicit flow generated by instruction $c[4]$, and thus it is executed under the low environment holding before entering the implicit flow. The non-secure flow is detected by upgrading the operand stack on entering the implicit flow, which causes a high value to be assigned to `x` by instruction $c[6]$.

The **ipd** rule updates the security environment when an implicit flow terminates, i.e. when the instruction which is the *ipd* of a previously executed branching instruction

```
1  push 1
2  push 0
3  load y
4  if 6
5  pop
6  store x
7  halt
```

Figure 5: Need of upgrading the stack.

is reached. The rule is applied only if the contents of the program counter is present on top of ρ, i.e. if the contents of the program counter is i and $\rho[1] = (i,\tau)$ for some τ. The rule does not change the program counter, but only modifies the environment, which is set to τ.

Given a program $P = (c,\Lambda)$ and a memory $M \in \mathcal{M}$, the concrete semantics of P is the transition system $C(P,M) = (\mathcal{C}, \longrightarrow, \sigma_0 \models \langle 1, M, \lambda, \lambda\rangle)$, where σ_0 is the lowest security level and the initial state consists of the address of the first instruction, the given memory and the empty operand stack and ipd stack.

Note that, if we ignore information on security, the concrete semantics coincides with the standard semantics of the language. Thus it can be used safely to execute the program.

We give now some definitions and results. With $\mathcal{V}_{\sqsubseteq\sigma} = \cup_{\tau\sqsubseteq\sigma}\mathcal{V}_\tau$ we denote the values with security level lower than or equal to σ. We use also $\mathcal{V}_{\sqsupseteq\sigma} = \cup_{\tau\sqsupseteq\sigma}\mathcal{V}_\tau$, $\mathcal{V}_{\not\sqsubseteq\sigma} = \cup_{\tau\not\sqsubseteq\sigma}\mathcal{V}_\tau$. With $\mathcal{R}_{\sqsubseteq\sigma} = \{(i,\tau) : \tau \sqsubseteq \sigma\}$ we denote the couples (i,τ) with security level lower than or equal to σ.

The following lemma states some properties of the concrete semantics concerning the operand and the ipd stacks.

Lemma 1. Given $P = (c, \Lambda)$ and $M_0 \in \mathcal{M}$, in each state $\sigma \models (i, M, S, \rho)$ belonging to $C(P, M_0)$,

1. $S \in (\mathcal{V}_{\sqsupseteq\sigma})^*$
2. $\rho \in (\mathcal{R}_{\sqsubseteq\sigma})^*$
3. $\forall i \in \{1, .., \sharp\rho - 1\} : \rho[i] = (j, \tau)$ *implies* $\rho[i+1] \in \mathcal{R}_{\sqsubseteq\tau}$

The lemma ensures that in each state with security environment σ the elements in the operand stack have security level higher than or equal to σ, while the elements in the ipd stack have security level lower than or equal to σ and are ordered in decreasing security from the top to the bottom.

We now define the notion of $\sigma-$safeness. Informally, a memory is $\sigma-$safe for P if each variable associated in P with a security level lower than or equal to σ, holds in M a security level lower than or equal to σ. Moreover, an operand stack is $\sigma-$safe for P if it contains only values with security level lower than or equal to σ.

Definition 5. Let $P = (c, \Lambda)$.

A concrete memory M with $D(M) = Var(c)$ is $\sigma-$safe for P if and only if $\forall x \in \Lambda_{\sqsubseteq\sigma} : M(x) \in \mathcal{V}_{\sqsubseteq\sigma}$.

A concrete stack S is $\sigma-$safe for P if and only if $S \in (\mathcal{V}_{\sqsubseteq\sigma})^*$.

The following theorem states the adequacy of the concrete semantics to characterize secure information flows. Given $P = (c, \Lambda)$, consider a concrete memory M with $D(M) = Var(c)$ in which, for each $\tau \in \mathcal{L}$, the variables with level τ hold a $\tau-$value. We call this memory Λ-consistent.

THEOREM 1. *Let $P = (c, \Lambda)$ be a program and $\sigma \in \mathcal{L}$. If every terminating execution of P starting form a Λ-consistent memory ends with a state $\tau \models \langle i, M, S, \rho\rangle \not\rightarrow$ such that M and S are $\sigma-$safe for P and $\tau \sqsubseteq \sigma$, then P is $\sigma-$secure.*

Proof Sketch. *Consider a terminating execution $C(P, M_0)$, where M_0 is a Λ-consistent memory. In each state, the value (k, τ) associated to a variable in the memory or present in the operand stack captures the least upper bound of the explicit and implicit flows on which this value depends (τ), and this occurs also for the final state. Thus, the $\sigma-$safeness of the memory and the operand stack in the final state ensures that the contents of the variables in $\Lambda_{\sqsubseteq\sigma}$ is not affected by the information with level $\not\sqsubseteq \sigma$. Moreover, if the final environment is less than or equal to σ, this means that all branches belonging to an implicit flow $\not\sqsubseteq \sigma$ have been completely executed until the respective ipds. Thus, under the condition of the theorem, the address of the final instruction is not affected by the information with level $\not\sqsubseteq \sigma$. Moreover, if the final environment is $\sqsubseteq \sigma$, no cycle has been executed under an environment $\not\sqsubseteq \sigma$: in fact no cycle reaches its ipd, that is the node $\sharp c + 1$ of the control graph, which does not correspond to any instruction of the program. Since the final environment is $\sqsubseteq \sigma$ for all terminating executions, the termination of the program does not depend on the information with level $\not\sqsubseteq \sigma$.*

5. THE ABSTRACT SEMANTICS

The concrete semantics cannot be used as static analysis tool for σ-security, because it shows the information flow of a particular execution, while σ-security concerns all executions. Moreover, the concrete transition system could be infinite. The purpose of abstract interpretation (or abstract semantics) [8, 9] is to correctly approximate the concrete semantics of all executions in a finite way, in order to be used to build a tool useful in practice. In this section we present an abstract operational semantics able to check σ-security. The semantics is an abstraction of the concrete semantics presented in the previous section: concrete values are abstracted by keeping their security level and disregarding their numerical part. All other structures are abstracted consequently.

The abstract domains are the following: $\mathcal{V}^\natural = \mathcal{L}$, $\mathcal{M}^\natural = var \rightarrow \mathcal{V}^\natural$, $\mathcal{S}^\natural = (\mathcal{V}^\natural)^*$. The domain of ipd stacks is the same of that of the concrete semantics. The abstract states, $\mathcal{C}^\natural$, are analogous to the concrete ones but with the memory M substituted by an abstract memory $M^\natural \in \mathcal{M}^\natural$ and the operand stack S substituted by an abstract stack $S^\natural \in \mathcal{S}^\natural$.

The abstraction function on values, $\alpha_V : \mathcal{V} \rightarrow \mathcal{L}$, is such that $\alpha_V((k, \sigma)) = \sigma$. The abstraction function on memories, $\alpha_M : \mathcal{M} \rightarrow \mathcal{M}^\natural$, is such that $(\alpha_M(M))(x) = \alpha_V(M(x))$ for each $x \in D(M)$. The abstraction function on stacks, $\alpha_S : \mathcal{S} \rightarrow \mathcal{S}^\natural$, is such that $(\alpha_S(S))[i] = \alpha_V(S[i])$, for each $i \in \{1, \cdots, \sharp S\}$. The abstraction function on states, $\alpha_C : \mathcal{C} \rightarrow \mathcal{C}^\natural$ is as follows:
$\alpha_C(\sigma \models (i, M, S, \rho)) = \sigma \models (i, \alpha_M(M), \alpha_S(S), \rho)$.

The rules of the abstract semantics are the same as those of the concrete one, but defined on the new domains, except for the rules of `if` and `ret`. These rules are defined in Figure 6, where E is the set of edges of the control flow graph of the program. Consider the `if` rule. In the abstract semantics, both branches are explored for every condition. Similarly, the `ret` rules explore every branch. As a consequence, the abstract transition system may contain multiple paths. The functions upgrading the memory and the stack are the same as those of the concrete semantics defined on the abstract domains. The transition relation of the abstract semantics is denoted by $\longrightarrow^\natural$. Given a program P and an abstract memory $M^\natural$, the abstract transition system defined by the abstract rules is denoted by $A(P, M^\natural) = (\mathcal{C}^\natural, \longrightarrow^\natural, \sigma_0 \models (1, M^\natural, \lambda, \lambda))$.

THEOREM 2. *Let $P = (c, \Lambda)$ be a program and $M_0^\natural \in \mathcal{M}$ be the $\Lambda-$consistent abstract memory. Consider the final states of $A(P, M_0^\natural)$, i.e. the states $\tau \models (i, M^\natural, S^\natural, \rho) \not\rightarrow$. If, for each of these states it holds that $M^\natural$ and $S^\natural$ are σ-safe for P and $\tau \sqsubseteq \sigma$, then P is σ-secure.*

Proof Sketch. *By definition of the concrete and abstract rules, for each path $C_0 \longrightarrow C_1 \longrightarrow \ldots$ in $C(P, M)$ there exists a path $\alpha_C(C_0) \longrightarrow^\natural \alpha_C(C_1) \longrightarrow^\natural \ldots$ in $A(P, \alpha_M(M))$. Moreover, a concrete memory $M \in \mathcal{M}$ and a concrete stack $S \in \mathcal{S}$ are $\sigma-$safe for P if and only if $\alpha_M(M)$ and $\alpha_S(S)$ are $\sigma-$safe for P. Thus the proof follows from Theorem 1.*

The above theorem is the basis of our security checking methodology: proving σ-safety of a program $P = (c, \Lambda)$ can be done by building the abstract transition system starting from an initial state with the lowest environment in $\mathcal{L}$ and in which each variable of the program is assigned its security level as specified by Λ. After, final states are examined for σ-safety of memories and operand stacks and for the value of the environments. Note that the abstract transition system is finite. In fact, since security levels, environments and abstract values are finite, then abstract memories are

$$\textbf{if}^\natural \quad \frac{c[i] = \texttt{if } j \qquad (i,j') \in E \qquad i \; not\_in \; \rho}{\sigma \models (i, M^\natural, \tau \cdot S^\natural, \rho) \longrightarrow^\natural \tau \models (j', upgrade^\natural_S(M^\natural, i, \tau), upgrade^\natural_S(S^\natural, \tau), (ipd(i), \sigma) \odot \rho)}$$

$$\textbf{ret}^\natural \quad \frac{c[i] = \texttt{ret } x \quad M^\natural(x) = \tau \qquad (i,j') \in E \qquad i \; not\_in \; \rho}{\sigma \models (i, M^\natural, S^\natural, \rho) \longrightarrow^\natural \sigma \sqcup \tau \models (j', upgrade^\natural_M(M^\natural, \sigma \sqcup \tau), upgrade^\natural_S(S^\natural, \sigma \sqcup \tau), (ipd(i), \sigma) \odot \rho)}$$

Figure 6: Abstract semantics rules

finite too. Moreover, ipd stacks are finite since they contain at most all addresses. Abstract operand stacks are finite because we assume stack boundedness.

6. RELATED WORK

The secure information flow property of programs was first formulated in [10]. In [11, 2], and successively [5], program certification was addressed, which statically checks secure information flow by inspecting the dependencies among the variables of the program. Successive works give the problem a more formal and precise basis. These works can be divided into two categories: type-based approaches and semantics-based approaches, to which our method belongs.

In type-based approaches, the security information of any variable belongs to its type and secure information flow is checked by means of a type system. The first works belonging to this approaches are [23, 22], concerning imperative high level languages. They use the notion of non-interference , which states that the final value of each variable does not depend on the initial value of variables at higher or not related security levels. In its stronger version non-interference includes also the absence of non-secure flow due to non-termination. In our setting, this property holds if the program is σ-secure for every σ. Non-interference can be inspected by checking that in the final states of the abstract transition system every variable in Λ_σ has a level $\sqsubseteq \sigma$. Recent works following the type-based approach are [12, 1, 17], where functional languages are considered and complex object and higher order programs are given a type system for security, and [16], where an extension to the Java language is presented that adds to programs information flow annotations. In semantics-based approaches, the work [14] defines an abstract interpretation approach for checking secure information flow in a high level imperative language. The method is based on the denotational semantics of the language. It defines a concrete enhanced semantics, and abstracts it to obtain a static tool for the analysis. In [13] a semantic approach is described, based on an axiomatic semantics, by means of which many secure programs considered insecure by other methods are accepted. Another semantic approach is defined in [18], based on partial equivalence relations. The disadvantage of these approaches is that they are not easily made fully automated.

Our approach is semantics-based and consider secure information flow for stack based assembly languages. Its application to high level languages has been described in [6]. Similarly to [14], we build an abstract semantics to statically check secure information flow. A difference is that we use an operational semantics instead of a denotational semantics. The advantage of using an operational semantics is that it defines an abstract machine and thus naturally allows keeping also information which is not related to the input-output behaviour of the program, but concerns the state of the machine during the execution. The advantage of the semantic approaches over the type-based ones is that, in general, they are able to certify a wider class of secure programs. On the other hand, since any semantic method executes the program, even if abstractly, a drawback may be the complexity of the analysis, in terms of space and time, which may be high for large programs. Instead, a typing system can in general handle larger programs, since it keeps less information.

We remark that, while all above works concern structured high level languages, we concentrate on assembly code. It could be interesting to define a secure typing system for stack based assembly language, starting, for example, from the work [15], which defines a typed stack based assembly language, or starting from [20], where a type system for java bytecode is defined. Semantics-based methods are defined in the literature to check safety of machine code. In [3] the proof carrying code approach is defined, where safety is proved using a high order logic. In [24] the machine program is annotated with some initial information on the machine on which the code must be executed and correctness properties are derived by means of an abstract execution of the program. An extension of these methods to cope with security properties could be developed. Since we are concerned with a non-structured languages, we use the notion of postdomination to handle implicit flows. An alternative approach is [25], which follows a continuation-passing style.

Security leaks not dealt with in the paper are those arising from partial operations that can raise exceptions or from timing leaks [22]. As future work, we plan to extend the proposed method to deal with these covert flows. Moreover, we intend to extend the approach to the whole Java bytecode.

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, 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] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carring code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation Proceedings*, pages 243–253, 2000.

[4] 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.

[5] J. Banatre, C. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.

[6] R. Barbuti, C. Bernardeschi, and N. D. Francesco.

Abstract interpretation of operational semantics for secure information flow. *Information Processing Letters*, To appear.

[7] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking secure information flow in assembly code by abstract interpretation. *IET Int. Report 20-01*, 2001.

[8] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Comp.*, 2:511–547, 1992.

[9] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *ACM POPL'92 Proceedings*, pages 83–94, 1992.

[10] D. E. Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–243, 1976.

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.

[12] N. Heintze and J. Riecke. The slam calculus: programming with secrecy and integrity. In *ACM POPL'98 Proceedings*, pages 365–377, 1998.

[13] R. Joshi and K. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, May 2000.

[14] 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.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

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

[17] F. Pottier and S. Conchon. Information flow inference for free. In *ACM ICFP'00 Proceedings*, pages 46–57, 2000.

[18] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *LNCS*, 1576:40–58, 1996.

[19] D. A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *ACM POPL'98 Proceedings*, 1998.

[20] R. Stata and M. Abadi. A type system for java bytecode subroutine. *ACM Trans. Program. Lang. Syst.*, 21(1):90–137, 1999.

[21] L. T. and F. Yellin. *The java virtual machine specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[22] D. Volpano and G. Smith. Eliminating covert flows with minimum typing. In *Proceedings 10th IEEE Computer Security Security Foundation Workshop*, pages 156–168, 1997.

[23] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[24] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *Proceedings ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 70–82, 2000.

[25] S. Zdancewic and A. Myers. Secure information flow and cps. *LNCS*, 2028:46–61, 2001.