

# Combining Abstract Interpretation and Model Checking for Analysing Security Properties of Java Bytecode

Cinzia Bernardeschi and Nicoletta De Francesco

Dipartimento di Ingegneria della Informazione  
Università di Pisa, Via Diotsalvi 2, 56126 Pisa, Italy  
{cinzia, nico}@iet.unipi.it

**Abstract.** We present an approach enabling end-users to prove security properties of the Java bytecode by statically analysing the code itself, thus eliminating the run time check for the access permission. The approach is based on the combination of two well-known techniques: abstract interpretation and model checking. By means of an operational abstract semantics of the bytecode, we built a finite transition system embodying security informations and abstracting from actual values. Then we model check it against some formulae expressing security properties. We use the SMV model checker. A main point of the paper is the definition of the properties that the abstract semantics must satisfy to ensure the absence of security leakages.

## 1 Introduction and Motivation

Java Virtual Machine Language (referred to hereafter as JVM) [12] is becoming a widely used medium for distributing platform-independent programs. In multilevel secure systems, the problem of the disclosure of sensitive information of programs written in JVM is particularly important. One of the main motivations is avoiding the damages produced by malicious programs which try to broadcast secret information. Mobile Java bytecode is checked by the Virtual Machine for safety properties: a bytecode Verifier enforces static constraints on Java bytecode to rule out type errors, access control violation, object initialisation failure and other dynamic errors. Moreover, to protect end-users from hostile programs, Java security model assigns access privileges to code and provides a customisable "sandbox" in which Java bytecode runs. At run-time a Java bytecode can do anything within the boundaries of its sandbox, but it can not take any action outside those boundaries.

This paper presents an approach enabling end-users to prove security properties of the Java bytecode by statically analysing the code itself, thus eliminating the run time check for the access permission. The approach is based on the combination of two well-known techniques: abstract interpretation and model checking. *Abstract interpretation* [7] is a method for analyzing programs by collecting approximate information about their run-time behavior. It is based on

a non-standard semantics, that is a semantic definition in which simpler (abstract) domains replace the standard (concrete) ones, and the operations are interpreted on the new domains. Using this approach different analyses can be systematically defined. In particular we refer to abstract interpretation based on operational semantics [7, 15]. *Model checking* [6] is an automatic technique for verifying finite state systems. This is accomplished by checking whether a structure, representing the system, satisfies a temporal logic formula describing the expected behavior. The approach combining abstract interpretation and model checking has been defined in [16, 17].

In [4] we defined an abstract interpretation based method to check *secure information flow* of a subset of JVMIL. The secure information flow property [9, 1, 18, 3] requires that information at a given security level does not flow to lower levels. A program, in which every variable is assigned a security level, 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, observing the final value of  $x$  reveals information on the value of the higher security variable  $y$ . In [4] a concrete operational semantics of the language is defined, 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 flows are modeled by an environment under which the instructions are executed; the environment, at every step of the computation, records the security level of the open implicit flows. Then an abstract operational semantics is defined, which disregards the numerical part of the values, and operates only on their security levels. By examining the final states of the abstract semantics it is possible to check secure information flow.

Other security leakages may occur when high level information is revealed not only by the value of the variables, but by the behavior of the program [10, 19]. These leakages are also known as *covert channels*. Consider the program `while ( $y > 0$ ) do skip`, where  $y$  is an high variable. It loops indefinitely when  $y$  is greater than zero. Thus high level information can be leaked by examining the termination behavior of the program. Another leakage is when high level information affects the number of instructions executed during the computations. For example, information on the initial value of the high security variable  $y$  can be leaked by observing the number of instructions executed by the program `if  $y=0$  then {  $x:=1$ ; skip } else  $x:=1$` .

Covert channels do not concern the input-output behavior of the program, but its dynamic behavior. They can be checked only by examining the intermediate states of the computations. In the present paper we define an approach to check security of programs, and in particular covert channels, which combines abstract interpretation with model checking: once built the abstract semantics

of the program, we inspect it for the security properties. In such a way we fully exploit the information embodied in the abstract semantics, which, being operational, shows (the abstraction of) all possible execution paths of the program. The main point of the paper is the definition of the properties that the abstract semantics must satisfy to ensure the absence of covert channels. The properties are then expressed as temporal logic formulae, and checked by using the SMV model checker [11].

The paper is organised as follows: Section 2 presents the language and the security model. Section 3 defines the concrete and abstract semantics. Section 4 introduces the program security properties and our method. Section 5 concludes the work.

## 2 The language and the security model

Given a set  $A$ ,  $A^*$  denotes the set of finite sequences of elements of  $A$ ;  $\lambda$  indicates the empty sequence; if  $w$  is a finite sequence,  $\#w$  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. Finally, if  $i \in \{1, \dots, \#w\}$ , with  $w[i]$  we denote the  $i$ -th element of  $w$ . 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 JVMML called JVMML0 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 Fig. 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** and **ret** instructions.

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

**Fig. 1.** Instruction set

A program is a sequence  $c$  of instructions, numbered starting from address 1;  $\forall i \in \{1, \dots, \#c\}$ ,  $c[i]$  is the instruction at address  $i$ . In the following, we denote by

$Var(c)$  the variable names occurring in  $c$ . We assume that a program is always executed starting from the instruction  $c[1]$  and with an empty operand stack. Moreover, we assume that programs respect the following static constraints, checked the Java bytecode Verifier: no stack overflow and underflow occur, and executions will not jump to undefined addresses.

We give the standard semantics of the language in terms of a Kripke structure [6]. A Kripke structure  $K = (Q, Q^0, AP, L, \rightarrow)$  is a 5-tuple where:

- $Q$  is a set of states;
- $Q^0 \subseteq Q$  is a set of initial states;
- $AP$  is a finite set of atomic propositions;
- $L : Q \rightarrow 2^{AP}$  is a function that labels each state with the set of atomic propositions true in that state;
- $\rightarrow \subseteq Q \times Q$  is a total transition relation on  $Q$  which gives the possible transitions between states.

The semantics uses the domain  $\mathcal{V}^\epsilon$  of constant values, ranged over by  $v, v', \dots$  and  $\mathcal{A}^\epsilon$  of addresses, ranged over by  $i, j, \dots$ .  $\mathcal{V}^\epsilon \cup \mathcal{A}^\epsilon$  is ranged over by  $k, k', \dots$ . For each  $X \subseteq var$ ,  $\mathcal{M}_X^\epsilon = X \rightarrow (\mathcal{V}^\epsilon \cup \mathcal{A}^\epsilon)$  is the domain of memories defined on  $X$ , ranged over by  $m, m', \dots$ .  $\mathcal{S}^\epsilon = (\mathcal{V}^\epsilon \cup \mathcal{A}^\epsilon)^*$  is the domain of stacks, ranged over by  $s, s', \dots$ . In the following,  $\mathcal{M}^\epsilon = \bigcup_{X \subseteq var} \mathcal{M}_X^\epsilon$  and, given  $m \in \mathcal{M}_X^\epsilon$  and  $Y \subseteq X$ ,  $m \downarrow_Y$  is the restriction of  $m$  to  $Y$ .

The domain of the states of the standard semantics is  $Q^\epsilon = \mathcal{A}^\epsilon \times \mathcal{M}^\epsilon \times \mathcal{S}^\epsilon$ . A state is given by the value of three variables,  $PC$ ,  $MEM$  and  $STACK$ , where  $PC$  is the program counter,  $MEM$  is the memory, and  $STACK$  is the operand stack. Each state is labeled by an atomic proposition for each variable, expressing its value. We denote by  $\langle i, m, s \rangle$  the state labeled by  $PC = i, MEM = m, STACK = s$ .

Given a program  $c$  and a memory  $m_0 \in \mathcal{M}^\epsilon$ , the standard semantics of the program is the structure  $(Q^\epsilon, \langle 1, m_0, \lambda \rangle, AP, L, \rightarrow^\epsilon)$ , where  $\rightarrow^\epsilon$  is defined in Fig. 2. The notation  $m[k / x]$  is used in the figure to indicate the memory  $m'$  which agrees with  $m$  for all variables, except for  $x$ , for which it is  $m'(x) = k$ . Since the program is deterministic, the corresponding structure has only one, possibly infinite, path. We call *final* a state such that  $PC = i$  with  $c[i] = \text{halt}$ . Note that self loops on these states are necessary to respect the totality of the transition relation.

We now recall the notion of control flow graph of a program, containing the control flow information among the instructions of the program, and the notion of postdomination and immediate postdomination in directed graphs [2].

**Definition 1 (control flow graph).** *Given a program  $c$ , the control flow graph of the program is the directed graph  $(V, E)$ , where  $V = \{1, \dots, \#c + 1\}$  is the set of nodes; and  $E \subseteq V \times V$  contains the edge  $(i, j)$  if and only if (a) the instruction at address  $j$  can be immediately executed after that at address  $i$ ; or (b)  $c[i] = \text{halt}$  and  $j = \#c + 1$ . The node  $\#c + 1$  is the final node of the graph and does not correspond to any instruction.*

$$\begin{aligned}
c[i] = \text{op} : & \quad \langle i, m, k_1 \cdot k_2 \cdot s \rangle \longrightarrow^\epsilon \langle i + 1, m, (k_1 \text{ op } k_2) \cdot s \rangle \\
c[i] = \text{pop} : & \quad \langle i, m, k_1 \cdot s \rangle \longrightarrow^\epsilon \langle i + 1, m, s \rangle \\
c[i] = \text{push } k : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle i + 1, m, k \cdot s \rangle \\
c[i] = \text{load } x : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle i + 1, m, m(x) \cdot s \rangle \\
c[i] = \text{store } x : & \quad \langle i, m, k \cdot s \rangle \longrightarrow^\epsilon \langle i + 1, m[k / x], s \rangle \\
c[i] = \text{goto } j : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle j, m, s \rangle \\
c[i] = \text{if } j : & \quad \langle i, m, 0 \cdot s \rangle \longrightarrow^\epsilon \langle i + 1, m, s \rangle \\
c[i] = \text{if } j : & \quad \langle i, m, (k \neq 0) \cdot s \rangle \longrightarrow^\epsilon \langle j, m, s \rangle \\
c[i] = \text{jsr } j : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle j, m, (i + 1) \cdot s \rangle \\
c[i] = \text{ret } x : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle m(x), m, s \rangle \\
c[i] = \text{halt} : & \quad \langle i, m, s \rangle \longrightarrow^\epsilon \langle i, m, s \rangle
\end{aligned}$$

**Fig. 2.** Standard semantics

**Definition 2 (postdomination).** *Let  $i$  and  $j$  be nodes of the control flow graph of a program. We say that node  $j$  postdominates  $i$ , denoted by  $j \text{ pd } i$ , if  $j \neq i$  and  $j$  is on every path from  $i$  to the final node. We say that node  $j$  immediately postdominates  $i$ , denoted by  $j = \text{ipd}(i)$ , if  $j \text{ pd } i$  and there is no node  $r$  such that  $j \text{ pd } r \text{ pd } i$ .*

### 3 Abstract interpretation

This section presents an instrumented concrete operational semantics of the language, embodying annotations on the information flow, and then an abstraction of this semantics, concentrating only on the information flow aspects and ignoring actual values. We assume a set  $\mathcal{L} = \{l, h\}$  of security levels, ordered by  $l \subset h$ , and with  $\sqcup$  we denote the upper bound between levels. We consider annotated programs, where each variable is associated with a security level. A program  $P$  is a triple  $\langle c, H, L \rangle$  where  $c$  is a sequence of instructions, and  $H$  and  $L$  are the high and low variables of  $P$ , respectively, with  $H \cup L = \text{Var}(c)$ .

The semantics 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 explicit and implicit information flows, on which the value depends. Moreover, the semantics 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.

We now introduce the domains of the concrete semantics.  $\mathcal{V} = (\mathcal{V}^\epsilon \times \mathcal{L})$  is the domain of concrete values. Concrete values are pairs  $(v, \sigma)$ , where  $v \in \mathcal{V}^\epsilon$  and  $\sigma \in \mathcal{L}$ . Low (high) values are those with the form  $(v, l)$  (resp.  $(v, h)$ ). The concrete domain of addresses is  $\mathcal{A} = (\mathcal{A}^\epsilon \times \mathcal{L})$ . Note that also addresses need to be annotated, since the decision on the address to jump to, can be made depending on high information. For each  $x \in \text{var}$ ,  $\mathcal{M}_X = X \rightarrow (\mathcal{V} \cup \mathcal{A})$  is the domain of concrete memories, ranged over by  $M, M', \dots$  and  $\mathcal{S} = (\mathcal{V} \cup \mathcal{A})^*$  are the concrete operand stacks, ranged over by  $S, S', \dots$ . The domain of concrete states is  $\mathcal{Q} = \mathcal{L} \times \mathcal{A}^\epsilon \times \mathcal{M} \times \mathcal{S} \times (\mathcal{A}^\epsilon \cup \{0\})$ . Each state is a configuration the state variables  $\langle ENV, PC, MEM, STACK, IPD \rangle$ , where *ENV* is the environment and contains a security level, *PC*, *MEM* and *STACK* are the program counter, the memory and the operand stack, respectively, and *IPD* is a flag used to handle high implicit flow, as explained below. The transition relation  $\longrightarrow$  on the concrete states is shown in Fig. 3.

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. For example, the execution of `load x` assigns to the value pushed onto the stack the least upper bound between the security level of  $M(x)$  and the environment. Note that `jsr` associates the return address pushed onto the stack with the security level of the present environment.

An implicit flow is entered with an `if` or a `ret` instruction. We use the notion of immediate postdomination to control implicit flows. Given an `if (ret)` instruction at address  $i$ ,  $ipd(i)$  is the first instruction not affected by the implicit flow, since it represents the point in which the different branches join. Consider an `if` instruction at address  $i$ . If this instruction is executed under the low security environment and the value on top of the operand stack is high, then the environment is upgraded to  $h$  and  $ipd(i)$  is recorded in *IPD*. Moreover the security level is upgraded of each value held by a variable assigned by a `store` instruction in at least one of the two branches. More precisely, let  $W = \{x | c[j] = \text{store } x \text{ and } j \text{ belongs to a path of the control flow graph starting at } i \text{ and ending at } ipd(i), \text{ excluding } ipd(i)\}$ . For each  $x \in W$ , if  $M(x) = (k, \sigma)$ , then  $up_M(M, i)(x) = (k, h)$ . The contents of the variables not in  $W$  is not changed. 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. Similarly, the security level of each value present in the operand stack is upgraded to  $h$  by applying the function  $ups$ . We upgrade the operand stack on entering an implicit flow to take into account the fact that the stack may be manipulated in different ways by the two branches. When the instruction  $c[ipd(i)]$  is executed, i.e. when  $PC = IPD$ , the environment is downgraded and *IPD* is reset to 0 (corresponding to no instruction). The `ret x` instruction is handled similarly, taking into account the security level of the address stored in  $x$ . Note that having only two security levels simplifies the semantics. In fact, if we consider whatever number of levels, *IPD*

$$\begin{aligned}
& i = i' \\
& \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle l, i, M, S, 0 \rangle \\
\\
& i \neq i' \\
& c[i] = \mathbf{op}, S = (k_1, \tau_1) \cdot (k_2, \tau_2) \cdot S' : \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i+1, M, (k_1 \text{ op } k_2, \tau_1 \sqcup \tau_2) \cdot S', i' \rangle \\
\\
& c[i] = \mathbf{pop}, S = (k, \tau) \cdot S' : \quad \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i+1, M, S', i' \rangle \\
\\
& c[i] = \mathbf{push } k : \quad \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i+1, M, (k, \sigma) \cdot S, i' \rangle \\
\\
& c[i] = \mathbf{load } x, M(x) = (k, \tau) : \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i+1, M, (k, \sigma \sqcup \tau) \cdot S, i' \rangle \\
\\
& c[i] = \mathbf{store } x, S = (k, \tau) \cdot S' : \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i+1, M[(k, \tau)/x], S', i' \rangle \\
\\
& c[i] = \mathbf{goto } j : \quad \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, j, M, S, i' \rangle \\
\\
& c[i] = \mathbf{if } j, S = (k \neq 0, \tau) \cdot S' : \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow ((\sigma = l) \wedge (\tau = h))? \\
& \quad \quad \quad \langle h, j, up_M(M, i), up_S(S), ipd(i) \rangle : \langle \sigma \sqcup \tau, i+1, M, S, i' \rangle \\
\\
& c[i] = \mathbf{if } j, S = (0, \tau) \cdot S' : \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow ((\sigma = l) \wedge (\tau = h))? \\
& \quad \quad \quad \langle h, i+1, up_M(M, i), up_S(S), ipd(i) \rangle : \langle \sigma \sqcup \tau, i+1, M, S, i' \rangle \\
\\
& c[i] = \mathbf{jsr } j : \quad \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, j, M, (i+1, \sigma) \cdot S, i' \rangle \\
\\
& c[i] = \mathbf{ret } x, M(x) = (j, \tau) : \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow ((\sigma = l) \wedge (\tau = h))? \\
& \quad \quad \quad \langle h, j, up_M(M, i), up_S(S), ipd(i) \rangle : \langle \sigma \sqcup \tau, j, M, S, i' \rangle \\
\\
& c[i] = \mathbf{halt} : \quad \quad \quad \langle \sigma, i, M, S, i' \rangle \longrightarrow \langle \sigma, i, M, S, i' \rangle
\end{aligned}$$

**Fig. 3.** Concrete semantics

would be a stack of addresses, instead of a single address. In our case, a high `if` that depends on another high `if` is already in a high region and the region terminates at the `ipd` of the outermost `if`. For the same reason, the upgrading of environment, memory and stack, and the modification of *IPD* is performed only when an `if (ret x)` instruction is executed in the low environment, and with a high value on the top of the stack (resp. a high address stored in *x*).

Given a program  $P = \langle c, H, L \rangle$  and a memory  $M_0 \in \mathcal{M}$ , the concrete semantics of  $P$  is the structure with  $\langle l, 1, M_0, \lambda, 0 \rangle$  as the initial state: it consists of the low environment, the address of the first instruction, the given memory, the empty operand stack and the *IPD* flag equal to 0.

If we ignore information on security, then the concrete semantics is isomorphic to the standard semantics of the language. The concrete semantics has an only extra case (case  $i = i'$  in Fig. 3) concerning the handling of *IPD*. It is applied when  $PC = IPD$  and has the effect of downgrading the environment and resetting *IPD*. Given a memory  $m \in \mathcal{M}_X^\epsilon$  and a concrete memory  $M \in \mathcal{M}_X$ , we say that they are *consistent* ( $M \leftrightarrow m$ ) if  $\forall x \in X : M(x) = (m(x), \tau)$ , for some  $\tau$ . Given a stack  $s \in \mathcal{S}^\epsilon$  and a concrete stack  $S \in \mathcal{S}$ , we say that they are *consistent* ( $S \leftrightarrow s$ ) if  $\#s = \#S$  and  $\forall i \in \{1, \dots, \#S\}, S[i] = (s[i], \tau)$ , for some  $\tau$ .

**Theorem 1 (standard and concrete semantics consistency).** *Given a program  $P = \langle c, H, L \rangle$ , let  $M_0 \in \mathcal{M}_{Var(c)}$  and  $m_0 \in \mathcal{M}^\epsilon$  such that  $M_0 \leftrightarrow m_0$ .*

*$\langle l, 1, M_0, \lambda, 0 \rangle \xrightarrow{*} \langle \tau, i, M, S, j \rangle$  if and only if  $\langle 1, m_0, \lambda \rangle \xrightarrow{*}^\epsilon \langle i, m, s \rangle$  with  $M \leftrightarrow m$  and  $S \leftrightarrow s$ .*

The purpose of abstract interpretation (or abstract semantics) [7, 8] is to correctly approximate the concrete semantics of all executions in a finite way. We now present an abstract operational semantics which is an abstraction of the concrete semantics: concrete values are abstracted by keeping only their security level and disregarding their numerical part. Addresses maintain their identity. All other structures are abstracted consequently.

The domain of values  $\mathcal{V} = \mathcal{V}^\epsilon \times \mathcal{L}$  is abstracted in the following way:  $(\mathcal{V}^\epsilon)^\natural = \{\odot\}$  and  $\mathcal{L}^\natural = \mathcal{L}$ . Thus  $\mathcal{V}^\natural = \{\odot\} \times \mathcal{L}^\natural$  which is isomorphic to  $\mathcal{L}$ . For every concrete value  $(k, \sigma) \in \mathcal{V}$ , its abstraction is given by  $\alpha_{\mathcal{V}}((k, \sigma)) = (\odot, \sigma) = \sigma$ . The domain of addresses  $\mathcal{A} = \mathcal{A}^\epsilon \times \mathcal{L}$  is abstracted in the following way:  $(\mathcal{A}^\epsilon)^\natural = \mathcal{A}^\epsilon$  and thus  $\mathcal{A}^\natural = \mathcal{A} \times \mathcal{L}^\natural$  and  $\alpha_{\mathcal{A}}((j, \sigma)) = (j, \sigma)$ . The abstract memories  $\mathcal{M}_X^\natural : X \rightarrow (\mathcal{V}^\natural \cup \mathcal{A}^\natural)$  are the functions from variable identifiers to abstract values and addresses. The abstraction function on memories  $\alpha_{\mathcal{M}} : \mathcal{M}_X \rightarrow \mathcal{M}_X^\natural$  assigns the abstraction of  $M(x)$  to  $M^\natural(x)$ , for each  $x \in X$ . The domain of stacks  $\mathcal{S}^\natural$  is defined analogously. The abstract states,  $\mathcal{Q}^\natural$ , contains the abstractions of the components of  $\mathcal{Q}$ :  $\alpha_{\mathcal{Q}} : \mathcal{Q} \rightarrow \mathcal{Q}^\natural$  is defined as  $\alpha_{\mathcal{Q}}(\langle \sigma, i, M, S, j \rangle) = \langle \sigma, i, \alpha_{\mathcal{M}}(M), \alpha_{\mathcal{S}}(S), j \rangle$ .

The abstract semantics is defined by the same rules of the concrete semantics, used on the abstract domains. The transition relation of the abstract semantics is denoted by  $\xrightarrow{\cdot}^\natural$ . Note that, for `if` instructions both alternative branches are executed, since every value is abstracted to " $\odot$ ". Moreover, since addresses maintain their identity, also all possible return points are explored. Given  $P = \langle c, H, L \rangle$  we denote by  $A(P)$  the abstract transition system defined by the abstract rules

and starting from the state  $\langle l, 1, M_0^h, \lambda, 0 \rangle$  where  $M_0^h \in \mathcal{M}_{var(c)}$  is such that  $\forall x \in L : M^h(x) = l$  and  $\forall x \in H : M^h(x) = h$ . The following theorem states that the abstract semantics mimics all possible concrete executions: the abstraction of every path of a concrete semantics is a path of the abstract one.

**Theorem 2 (correctness of the abstract semantics).** *Given two concrete states  $Q, Q' \in \mathcal{Q}$ ,  $Q \longrightarrow Q'$  implies  $\alpha_Q(Q) \longrightarrow^h \alpha_Q(Q')$ .*

Note that the abstract semantics is finite. In fact, since security levels, environments and abstract values are finite, then abstract memories are finite too. Abstract operand stacks are finite because we assume stack boundedness.

## 4 Model checking the abstract semantics

In this section we define some security properties guaranteeing the absence of different security leakages and we show how it is possible to prove them for a program  $P$  by model checking the abstract semantics of  $P$  for a set of logic formulae.

In the following, we assume a program  $P = \langle c, H, L \rangle$ . The following property states that the final value of each low variable does not depend on the initial value of the high variables.

**Definition 3 (secure information flow).**  *$P$  satisfies the secure information flow property (SIF) if for each pair of memories  $m_1, m_2 \in \mathcal{M}_{var(c)}^\epsilon$ , with  $m_1 \downarrow_L = m_2 \downarrow_L$ , if  $\langle 1, m_1, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_1, m'_1, s_1 \rangle$  and  $\langle 1, m_2, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_2, m'_2, s_2 \rangle$  with  $c[i_1] = c[i_2] = \text{halt}$ , then  $m'_1 \downarrow_L = m'_2 \downarrow_L$ .*

The second property we consider concerns the timing flows due to termination observation [10, 19]: it is not possible to leak high information by observing the termination of the program.

**Definition 4 (termination agreement).**  *$P$  satisfies the termination agreement property (TERM) if for each pair of memories  $m_1, m_2 \in \mathcal{M}_{var(c)}^\epsilon$ , with  $m_1 \downarrow_L = m_2 \downarrow_L$ , if  $\langle 1, m_1, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_1, m'_1, s_1 \rangle$  with  $c[i_1] = \text{halt}$ , then  $\langle 1, m_2, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_2, m'_2, s_2 \rangle$  with  $c[i_2] = \text{halt}$ .*

The third property concerns timing channels where the number of instructions executed in a computation may reveal information on the value of the high variables.

**Definition 5 (timing agreement).** *We say that  $P$  satisfies the timing agreement property (TIME) if for each pair of memories  $m_1, m_2 \in \mathcal{M}_{var(c)}^\epsilon$ , with  $m_1 \downarrow_L = m_2 \downarrow_L$ , if  $\langle 1, m_1, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_1, m'_1, s_1 \rangle$  with  $c[i_1] = \text{halt}$ , and  $\langle 1, m_2, \lambda \rangle \xrightarrow{*}^\epsilon \langle i_2, m'_2, s_2 \rangle$  with  $c[i_2] = \text{halt}$ , then the two computations have the same length.*

The following theorems relate the abstract semantics with the above properties.

**Theorem 3.** *P satisfies SIF if for each state of  $A(P)$  such that  $c[PC] = \text{halt}$ , then  $\forall x \in L, MEM[x] = l$  or  $MEM[x] = (i, l)$  for some  $i$ .*

**Theorem 4.** *P satisfies TERM if every state of  $A(P)$  such that  $ENV = h$  does not belong to a cycle.*

**Theorem 5.** *P satisfies TIME if:*

*all paths in  $A(P)$  starting from a state satisfying  $STACK[1] = h$  and  $PC = i$  where  $c[i] = \text{if}$  and ending with a state satisfying  $PC = \text{ipd}(i)$  have the same length.*

*all paths in  $A(P)$  starting from a state satisfying  $PC = i$  and  $MEM[x] = (j, h)$  where  $c[i] = \text{ret } x$  and ending with a state satisfying  $PC = \text{ipd}(i)$  have the same length.*

Theorem 3 states that to check *SIF* it suffices to examine the final states of the abstract semantics, and, in particular, to check that in these states the low variables hold low values and the stack contains only low values. Theorem 4 says that *TERM* can be controlled by checking that no instruction is executed more than once under a high environment. Theorem 5 states that, to ensure *TIME*, the branches starting from an *if* instruction at address  $i$  with an high condition (the value on top of the stack is  $h$ ) must have the same length until  $\text{ipd}(i)$  is reached. A similar condition is stated for *ret* instructions with high return address.

The proof of the above theorems is based on a set of properties of the concrete semantics that we now briefly show. We need some definitions. Two concrete values  $(k_1, \sigma_1), (k_2, \sigma_2) \in (\mathcal{V} \cup \mathcal{A})$  are low equivalent ( $(k_1, \sigma_1) \sim^{\mathcal{V}} (k_2, \sigma_2)$ ) if and only if either they are equal or  $\sigma_1 = \sigma_2 = h$ . Two concrete memories  $M, M' \in \mathcal{M}_X$  are low equivalent ( $M \sim^{\mathcal{M}} M'$ ) if and only if for each  $x \in X, M(x) \sim^{\mathcal{V}} M'(x)$ . To define low equivalence of operand stacks, we represent them in a canonical form. Each  $S \in \mathcal{S}$  is uniquely representable in canonical form as  $S = u \cdot w$ , where  $w$  contains only high values and the bottom element of  $u$  is a low value. Two concrete operand stacks  $S = u \cdot w$  and  $S' = u' \cdot w'$  are low equivalent ( $S \sim^{\mathcal{S}} S'$ ) if and only if  $\#u = \#u'$  and  $\forall i \in \{1, \dots, \#u\} : u[i] \sim^{\mathcal{V}} u'[i]$ .

Two operand stacks are low equivalent if and only if the  $u$  parts of their canonical representation have the same length and hold low equivalent values in the same positions.

The following lemma states that, if the environment is low, two concrete transitions starting from the same instruction and low equivalent memories and operand stacks, maintain low equivalence of memories and stacks. Moreover, after the transitions, the environments are equal. Finally, if the environment is still low, then also the contents of the program counter is the same and *IPD*. Instead, if the environment becomes high, then  $\text{ipd}(i)$  is stored into *IPD*.

**Lemma 1.** *Let  $M_1 \sim^{\mathcal{M}} M_2$  and  $S_1 \sim^{\mathcal{S}} S_2$ .*

*$\langle l, i, M_1, S_1, 0 \rangle \rightarrow \langle \tau, i_1, M'_1, S'_1, j \rangle$  implies  $\langle l, i, M_2, S_2, 0 \rangle \rightarrow \langle \tau, i_2, M'_2, S'_2, j \rangle$   
with  $M'_1 \sim^{\mathcal{M}} M'_2$ ,  $S'_1 \sim^{\mathcal{S}} S'_2$ , and,  
if  $\tau = l$ , then  $i_1 = i_2$  and  $j = 0$ ; if  $\tau = h$ , then  $j = \text{ipd}(i)$*

The following lemma states that, in each transition executed under the high security environment, the memory and the operand stack before and after the transition are low equivalent to each other. Moreover, the environment is downgraded only when the instruction at address  $IPD$  is executed, and in this case  $IPD$  is reset to 0.

**Lemma 2.**  *$\langle h, i, M, S, j \rangle \rightarrow \langle \tau, i', M', S', j' \rangle$  implies  $M \sim^{\mathcal{M}} M'$ ,  $S \sim^{\mathcal{S}} S'$  and, if  $\tau = l$ , then  $i' = j$  and  $j' = 0$ .*

The proofs of the Theorems 3, 4 and 5 is based on the following informal reasoning. Consider two standard computations starting from memories that agree on the value of low variables. Consider the corresponding concrete computations, existing by Theorem 1. By Lemma 1, until the environment is low, the two computations perform the same instructions, keep low equivalence of memories and operand stacks, and maintain the same environment and  $IPD = 0$ . By the same lemma, if one of them upgrades the environment, also the other one do. While executing in the high environment, low equivalence of memory and stacks is maintained by Lemma 2. The proof then follows by considering the abstract computations corresponding to the concrete ones, existing by Theorem 2, and the conditions expressed by the theorems.

#### 4.1 Implementation in SMV

We have used the SMV tool [11] to implement our method. SMV is a tool for checking finite state systems against specifications in the temporal logic CTL [6]. The specifications are assertions on the state variables and on the paths of the system. Using the SMV model checker, the three conditions above can be written as follows:

$$\varphi_{SIF} = \bigwedge_{x \in L} AG((PC = i) \wedge c[i] = \text{halt}) \rightarrow ((MEM[x] = l) \vee (MEM[x] = (j, l)));$$

$$\varphi_{TERM} = AG(((PC = i) \wedge (ENV = h)) \rightarrow AG(PC! = i));$$

$$\begin{aligned} \varphi_{TIME} = & AG(((PC = i) \wedge (STACK[1] = h) \wedge ((c[i] = \text{if})) \\ & \rightarrow \bigvee_{r=1..n} X^r (PC = \text{ipd}(i)) \\ & \wedge AG(((PC = i) \wedge (MEM[x] = (j, h) \wedge (c[i] = \text{ret } x)) \\ & \rightarrow \bigvee_{r=1..n} X^r (PC = \text{ipd}(i))) \end{aligned}$$

where  $n = \#c$  and  $X^r = X \dots X$   $r$  times.

We recall that in CTL a state  $Q$  satisfies  $A \phi$  if  $\phi$  is true in all paths starting from  $Q$ ;  $Q$  satisfies  $G \phi$  if  $\phi$  is true in all states reachable from  $Q$ ;  $Q$  satisfies  $X \phi$  if  $\phi$  is true in all states reachable from  $Q$  by only one transition. The three

formulae are the translation in the logic of SMV of the conditions expressed by Theorems 3, 4 and 5. In  $\varphi_{TIME}$ , to check that the lengths of the paths from a state to another one are all equal, we use the sequences of the  $X$  operator with length  $\leq \sharp c$  : the formula is true if  $r \leq \sharp c$  exists such that  $X^r(PC = ipd(i))$  is true; in this case all paths have length  $r$ .

## 4.2 Examples

Consider programs with  $L = \{x\}$  and  $H = \{y\}$ . Fig. 4 shows a non-secure implicit flow. It corresponds to the program: `if y=0 then x:=1 else x:=0`. Fig. 4(c) shows the abstract structure of the program.  $A(P)$  does not satisfy  $\varphi_{SIF}$  nor  $\varphi_{TIME}$ , while  $\varphi_{TERM}$  is satisfied. Fig. 4(b) shows a concrete computation violating  $SIF$ .

```

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

```

(a)

$\langle ENV, PC, [MEM(x) \ MEM(y)], STACK, IPD \rangle$

```

       $\langle l, 1, [(l)(h)], \lambda, 0 \rangle$ 
       $\downarrow_{load}$ 
       $\langle l, 2, [(5, l)(1, h)], (1, h), 0 \rangle$ 
       $\downarrow_{if true}$ 
       $\langle h, 5, [(5, l)(1, h)], \lambda, 6 \rangle$ 
       $\downarrow_{push}$ 
       $\langle h, 6, [(5, l)(1, h)], (0, h), 6 \rangle$ 
       $\downarrow_{ipd}$ 
       $\langle l, 6, [(5, l)(1, h)], (0, h), 0 \rangle$ 
       $\downarrow_{store}$ 
       $\langle l, 7, [(0, h)(1, h)], \lambda, 0 \rangle$ 
       $\downarrow_{halt}$ 

```

(b)

```

       $\langle l, 1, [(l)(h)], \lambda, 0 \rangle$ 
       $\downarrow_{load}$ 
       $\langle l, 2, [(l)(h)], (h), 0 \rangle$ 
       $\swarrow_{if true}$   $\searrow_{if false}$ 
       $\langle h, 5, [(l)(h)], \lambda, 6 \rangle$   $\langle h, 3, [(l)(h)], \lambda, 6 \rangle$ 
       $\downarrow_{push}$   $\downarrow_{push}$ 
       $\langle h, 4, [(l)(h)], h, 6 \rangle$ 
       $\downarrow_{goto}$ 
       $\langle h, 6, [(l)(h)], (h), 6 \rangle$ 
       $\downarrow_{ipd}$ 
       $\langle l, 6, [(l)(h)], (h), 0 \rangle$ 
       $\downarrow_{store}$ 
       $\langle l, 7, [(h)(h)], \lambda, 0 \rangle$ 
       $\downarrow_{halt}$ 

```

(c)

**Fig. 4.** A program not satisfying  $SIF$

The program in Fig. 5 is an example of violation of termination agreement. This program terminates depending on the value non-zero or zero of the high security level variable  $y$ . It corresponds to the high level program: `while (y) do skip`. Fig. 5(b) shows the abstract semantics of the program. Note that it satisfies  $\varphi_{SIF}$ , but not  $\varphi_{TERM}$ : there is a cycle including states with  $ENV = h$ .

Fig. 6 shows an example of not secure program due to a timing channel. The number of steps of the program depends on the value of the high security level variable  $y$ . When the program terminates the low variable  $x$  always holds 1. Fig. 6 (b) shows the abstract semantics of the program. It satisfies  $\varphi_{SIF}$  and  $\varphi_{TERM}$ , but it does not satisfy  $\varphi_{TIME}$ .

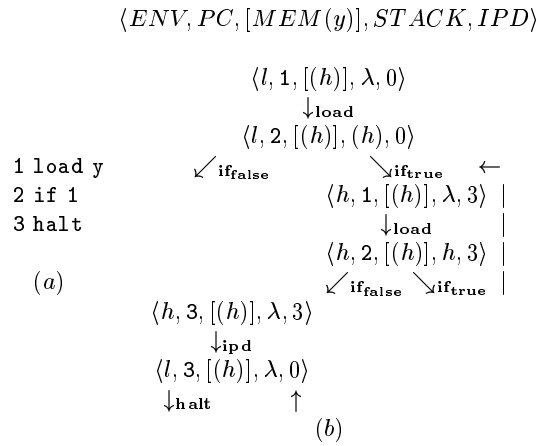
## 5 Conclusions

The work [5] presents an approach, based on abstract interpretation and model checking, enabling a smart card issuer to verify that a new applet securely interacts with already downloaded applets. The work concentrates on applet interfaces, therefore the security levels correspond to the possible interactions among applets. Covert channels are not handled and in general the formulae are not general but specific for the particular applet to be verified.

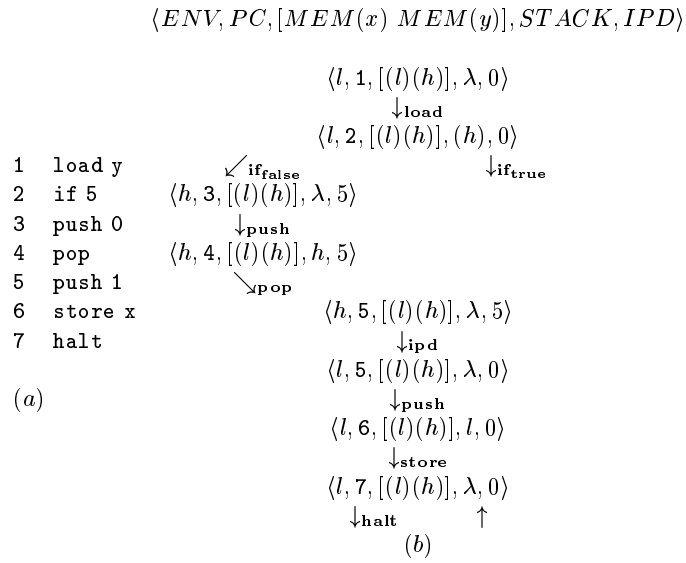
An alternative approach to check secure information flow in assembly code may be developed by defining a typing system for this purpose. Typing systems have been defined for high level languages for example in [19, 14]. Typing systems for assembly code have been defined, for example, in [13, 20, 21], but they check safety and do not handle secure information flow. An advantage of our approach with respect to those based on typing is that it is semantics based and thus keeps information on the dynamic behavior of programs, allowing to check more precisely the desired properties. A further advantage is flexibility: different security properties can be checked on the abstract semantics by expressing them as temporal logic formulae. For example, the condition that a low variable never holds a high value during the computations can be expressed by the formula:  $\bigwedge_{x \in L} AG(MEM[x] = l)$ . This condition, that ensures secure information flow, corresponds to that checked by the typing approaches and it is stronger than that expressed in Theorem 3.

## References

1. G. R. Andrews, R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on programming languages and systems*, 2(1), 1980, pp. 56-76.
2. T. Ball. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming languages and Systems*, Vol. 2, N. 1-4, 1993, pp. 1-16.
3. R. Barbuti, C. Bernardeschi, N. De Francesco. Abstract Interpretation of Operational Semantics for Secure Information Flow. To appear on *Information Processing Letters*.



**Fig. 5.** A program not satisfying *TERM*



**Fig. 6.** A program not satisfying *TIME*

4. R. Barbuti, C. Bernardeschi, N. De Francesco. Checking Security of Java Bytecode by Abstract Interpretation. Proceedings of the Special Track on Security at the ACM Symposium on Applied Computing (SAC2002), March 10-14, Spain 2002, (to appear).
5. P. Bieber, J. Cazin, P. Girard, J-L. Lanet, V.Wiels, G. Zanon. Checking Secure Interactions of Smart Card Applets. Proceedings of ESORICS 2000.
6. E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on programming Languages and Systems*, vol. 8, n. 2, 1986, 244-263.
7. P. Cousot, R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2, 1992, pp. 511-547.
8. P. Cousot, R. Cousot. Inductive Definitions, Semantics and Abstract interpretations. *Proc. 19th ACM Symposium on Principles of programming languages*, POPL'92, 1992, pp. 83-94.
9. D. E. Denning, P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977, pp. 504-513.
10. B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, Vol. 16, n. 10, 1973, pp. 613-615.
11. K.L. McMillan. The SMV language. Cadence Berkeley Labs, Cadence Design Systems, Berkeley, March 1999.
12. Lindholm T., F. Yellin. The java virtual machine specification. Addison-Wesley, 1996.
13. G. Morrisett, D. Walker, K. Crary, N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, Vol. 21, N. 3, 1999, pp. 527-568.
14. A. Sabelfeld, D. Sands. The impact of synchronization on secure information flow in concurrent programs. Proceedings Andrei Ershov 4th International Conference on Perspective of System Informatics, Novosibirsk, LNCS, Springer-Verlag, July 2001.
15. D. A. Schmidt. Abstract interpretation of small-step semantics. Proceedings 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, M. Dam and F. Orava, eds. Springer, 1996.
16. D. A. Schmidt, B. Steffen. Program analysis as model checking of abstract interpretations. *Proc. 5th Static Analysis Symposium*, G. Levi. ed., Pisa, September, 1998. Springer LNCS 1503.
17. D. A. Schmidt. Data-flow analysis is model checking of abstract interpretations. *Proc. 25th ACM Symp. Principles of Programming Languages*, San Diego, 1998.
18. D. Volpano, G. Smith, C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 1996, pp. 167-187.
19. D. Volpano, G. Smith. Eliminating covert flows with minimum typing. Proceedings 10th IEEE Computer Security Security Foundation Workshop, June 1997, pp. 156-168.
20. R. Stata, M. Abadi. A type system for java bytecode subroutine. *ACM Transactions on Programming Languages and Systems*, Vol. 21, n. 1, 1999, pp. 90-137.
21. Z. Xu, B. P. Miller, T. Reps. Safety Checking of Machine Code. Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, Canada, 2000, pp. 70-82.