Using Standard Verifier to Check Secure Information Flow in Java Bytecode

Cinzia Bernardeschi, Nicoletta De Francesco, Giuseppe Lettieri
Dipartimento di Ingegneria dell’Informazione
Università di Pisa
Via Diotisalvi 2, 56126, Pisa, Italy

Abstract

When an applet is sent over the internet, Java Virtual Machine code is transmitted and remotely executed. Because untrusted code can be executed on the local computer running the web browser, security problems may arise. Here we present a method to check illicit flows in Java bytecode, that exploits the type-level abstract interpretation of bytecode verification. We present an algorithm transforming a bytecode into another one that, when abstractly executed by the standard bytecode Verifier, reveals illicit information flows. We show an example of application of the method.

Keywords: Security, Information Flow, Java bytecode, Bytecode Verifier

Contact author:
Cinzia Bernardeschi
Dipartimento di Ingegneria dell’Informazione, Università di Pisa
Via Diotisalvi 2, 56126, Pisa, Italy
Phone: +39 50 568541, Fax: +39 50 568522, email: cinzia@iet.unipi.it

1 Introduction

When an applet is sent over the internet, Java Virtual Machine (JVM) code (also called bytecode) [16] is transmitted and remotely executed. Because untrusted code can be executed on the local computer running the web browser, security problems may arise: a malicious downloaded applet could try to observe, alter, or use information it is not authorized to. 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, thus causing unauthorised information flows.

Java provides facilities for enforcing security properties. Java security model mainly relies on the Java bytecode Verifier and on the Security Manager [16]. Bytecode verification checks the bytecode for consistency prior to execution. The Verifier checks type correctness, stack overflow or underflow, code containment, registers and objects initialisation. The Verifier executes a data-flow analysis applied to a type-level abstract interpretation of the virtual machine. The Java Security Manager 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 cannot take any action outside those boundaries. The Security Manager will disallow most operations when they are requested by untrusted code, and will allow trusted code to do whatever it wants.
The standard Java security model may be in some cases too restrictive. In fact, many useful programs need to access to private files in order to perform their tasks. These programs can be allowed to read secret information, provided that they do not make it available outside the private environment. To cope with these problems, a security policy should model the allowed information flow between the applications. A candidate for this policy is a multilevel security policy: information is assigned a security level, and the presence of illicit information flows can be checked by requiring that information at a given security level does not flow to lower levels. This property is called secure information flow property \cite{E1, E2, E3}. Let us suppose that an object $o_1$ has security level higher than that of another object $o_2$. Assume that both $o_1$ and $o_2$ have an attribute with type $T$, which is read and written by methods $\textit{read()}$ and $\textit{write}(T x)$, respectively. Examples of violation of secure information flow are: $o_2.\textit{write}(o_1.\textit{read}())$ and $\textit{if} (o_1.\textit{read}() == 0) \textit{then} o_2.\textit{write}(1) \textit{else} o_2.\textit{write}(0)$. In the first case, there is an explicit information flow from $o_1$ to $o_2$, while, in the second case there is an implicit information flow: in both cases, observing the final value of $o_2$ reveals information on the value of the higher security object $o_1$.

Here we present a technique to check illicit information flows in Java bytecode. The technique uses the standard bytecode Verifier: we transform the class files corresponding to a set of classes into another set of class files that, when checked by the Verifier, reveal illicit information flows. In the transformed class files there is a class for each security level and the values are instances of these classes. Moreover, each instruction of the original bytecode of the methods is transformed into a sequence of instructions. When the transformed bytecode is abstractly executed by the Verifier, a value indicates the least upper bound of the security levels of the information flows, both explicit and implicit, the value depends on. Thus we exploit the type-level abstract interpretation of bytecode verification to detect illegal information flows. The main advantage of our approach is that we use the standard JVM bytecode Verifier, instead of defining ad hoc methods.

2 Background

We consider the subset of JVM language shown in Figure 1. The instructions are typed: in the figure $\alpha$ stays for a set of types: for example $\textit{i}load$ represents loading an integer value while $\textit{aload}$ represents loading an object reference. We do not consider subroutines. For simplicity, we suppose that all methods have at most one parameter.

The bytecode of a method is a sequence $B$ of uniquely labeled instructions: each element in $B$ has the form $k : \textit{instr}$, where $k$ is the label and $\textit{instr}$ is an instruction. We denote as $L(B)$ the labels of the instructions in bytecode $B$. The JVM is a stack machine manipulating an operand stack, a set of local variables, or registers, and a heap containing object instances. When a method is invoked it executes with a new empty stack and with an initial memory where all variables are undefined except for the first one, say it $x_0$, that contains the object instance on which the method was called and the second one, say it $x_1$, that contains the actual parameter. When the method returns, control is transferred to the calling method: the caller’s execution environment (operand stack and local variables) is restored and the returned value, if any, is pushed onto the operand stack.

We use the notion of control flow graph of a program and the notion of postdomination and immediate postdomination in directed graphs \cite{E4}.

Given a bytecode $B$, the control flow graph of the bytecode is the directed graph $(V, E)$, where $V = L(B)$ 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
pop  Pop top operand stack
dup  Duplicate top operand stack element
aop  Pop two operands with type α off the stack, perform the
      operation op ∈ {add, mult, compare ..}, and push the result
      onto the stack
aconst d  Push constant d with type α onto the operand stack
aconst_null  Push the null object reference onto the operand stack
aload x  Push the value with type α of the variable x onto the stack
astore x  Pop a value off the stack and store the α value into local variable x
ifge L  Pop a value off the stack, and compare it against zero.
        Branch to L if the value is greater than or equal to 0
ifnull L  Pop a reference off the operand stack
        and branch to L if reference is null
goto L  Jump to L
getfield C/: f  Pop a reference to an object of class C off the operand stack;
      fetch field f of such object and put the field on top of the stack
putfield C/: f  Pop a value v and a reference to an object of class C from the
      operand stack; set field f of the object to v
invoke C/: mt  Pop a reference to an object of class C
      and a value v from the stack.
      Invoke method C/: mt of the referenced object with parameter v
return  Return void from the method
new C  Create an instance of class C and put a reference to this
      instance on top of the stack

Figure 1: Instruction set.

at address i. We assume that there is one and only one final node.

Let i, j ∈ V: j postdominates i, denoted by j pd i, if j ≢ i and j is on every path
from i to the final node. j immediately postdominates i, denoted by j = ipd(i), if
j pd i and there is no node r such that j pd r pd i.

2.1 Java bytecode verification

The result of the compilation of a Java program is a set of class files. A class file
is generated by the Java Compiler from each class definition of the program. It is
composed of the declaration of the class and of a bytecode for each method of the
class.

The bytecode is subject to a static analysis called bytecode verification, whose
purpose is to make sure that the code is well typed. The bytecode verification is a
crucial component in the Java security model and considerable research efforts have
been expended to formalise bytecode verification algorithms and their correctness

The bytecode Verifier executes a data-flow analysis applied to a type-level ab-
stract interpretation of the virtual machine [16]. The abstract interpreter for the
JVM instruction set executes the instructions, but operates over types instead of
values. It manipulates stacks of types and registers of types and simulates the exec-
ution of instructions at the level of types. The abstract interpreter is defined by a
transition function \( i : (S, R) \rightarrow (S', R') \) for each instruction, where \( i \) is the address
of the instruction, \( (S, R) \) is the stack type and register type before the instruction,
and \( S', R' \) is the stack type and register type after the instruction. The types ma-
Manipulated by the abstract interpreter include primitive types and object reference types represented by the names of the corresponding classes and array types.

The verification is method per method assuming that all other methods are well typed when verifying a method. A coinductive argument shows that if all methods are well-typed, the program (a collection of methods) is well-typed. The initial stack and register types reflect the state of the JVM on method entrance: the stack type is empty; the types of the first n registers corresponding to the n method parameters are set to the types of the method parameters in the signature. A standard fixpoint iteration is applied. The transition function of the abstract interpreter is applied over the instructions, taking the stack type and the register type after the preceding instruction as the stack type and the register type before the next instruction. If an instruction at address i has several predecessor instructions, the stack and register types before i is the least upper bound of the states after all predecessors of i. The least upper bound of two types is the smallest common supertype of them. If the fixpoint is reached, verification succeeds. Verification fails if the interpreter cannot make a transition, or when the least upper bound is undefined.

3 The security model

We assume no dynamic object creation and no dynamic class loading. Thus we consider a set \( C \) of class definitions and a set \( Objects = \{ o_1, \ldots, o_n \} \) of instances of the classes in \( C \). We denote by \( O = Objects \rightarrow ObjectValues \) the domain of object valuations, i.e. the functions assigning a value to each object in \( Objects \). An execution state of a method \( C.mt \) is a tuple \(( B, i, M, S ) \), where \( B \) is the bytecode corresponding to \( C.mt \), i is the address held by the program counter, \( M : Var(B) \rightarrow Values \) is the local memory, representing the current state of the local variables of \( B \) (\( Var(B) \)), and \( S \in Values^* \) is the current state of the operand stack. Given a variable \( x \), we denote by \( M(x) \) the contents of \( x \) in the memory \( M \). The initial state of the execution of a method \( C.mt \) is \(( B, 0, M_0, \lambda ) \), where 0 is the address of the first instruction, \( M_0(x) \) contains the reference to the object acted on by the method invocation, \( M_0(x1) \) contains the actual parameter of \( C.mt \) and the other variables are undefined in \( M_0 \). We denote by \( \Omega \) the domain of the states of all methods. A state of the system execution is a pair \(( E, O ) \) where \( E \in \Omega ^* \) is the activation stack and \( O \in O \) is the current state of the objects. We denote as \( Q \) the domain of the states of the system.

We give the semantics of the language as a set of inference rules. Figure 2 reports the rules for some instructions. The semantics performs no safety check: we assume that the bytecode has been accepted by the Verifier. \( M[k/x] \) is used to indicate the memory \( M' \) which agrees with \( M \) for all variables, except for \( x \), for which it is \( M'(x) = k \). Similarly, \( O[k/o,f] \) indicates the object valuation \( O' \), which differs from \( O \) only on field \( f \) of object \( o \), which is assigned \( k \). The rules of the semantics define a transition relation \( \rightarrow \subseteq Q \times Q \). We denote as \( \rightarrow^* \) the reflexive and transitive closure of \( \rightarrow \).

We assume a set \( \Sigma \) of security levels, among which an order relation \( \sqsubseteq \) is established. Here, for simplicity, we suppose only two levels, i.e. \( \Sigma = \{ L, H \} \), with \( L \sqsubseteq H \) (\( L \) is dominated by \( H \)). A security specification defines a security policy for a set of class definitions. Given a set of class definitions \( C \), a security specification \( S : C \rightarrow \Sigma \) associates to each class \( C \in C \) a security level. We assume that all instances and all attributes of a class have the level of the class.

Given a valuation \( O \in O \) we denote by \( O \downarrow L \) the restriction of \( O \) to the objects with security level \( L \). Given a set of classes \( C \), we denote by \( Mtids(C) \) the methods belonging to the classes in \( C \). We now define the secure information flow for a method in \( Mtids(C) \). The definition is parametric with respect to a security specification \( S \),
an assignment \( \mathcal{P} : \textit{Mtds}(C) \rightarrow \Sigma \) of a security level to the parameter, if any, of each method in \( \textit{Mtds}(C) \), and an assignment \( \mathcal{R} : \textit{Mtds}(C) \rightarrow \Sigma \) of a security level to the return value, if any, of each method. A method has secure information flow if two executions of the method starting from object valuations that agree on the low objects and local memories that agree on the value of the low parameters, produce object valuations that agree on the low objects and return the same value if the return value is specified as low.

**Definition 1 (secure information flow for a method)** Let \( C \) be a set of class definitions and Objects a set of object instances of classes in \( C \). Given \( \Sigma : C \rightarrow \Sigma \) and \( \mathcal{P}, \mathcal{R} : \textit{Mtds}(C) \rightarrow \Sigma \), a method \( C.mt \in \textit{Mtds}(C) \) has secure information flow with respect to \( \Sigma, \mathcal{R}, \mathcal{P} \) (it is \( \mathcal{S}, \mathcal{R}, \mathcal{P} \)-secure) if the following property holds:

Let \( B \) be the bytecode corresponding to \( C.mt \). Let \( M_1 \) and \( M_2 \) such that

\[
M_1(x) = M_2(x) = \text{undefined for each } x \in \text{Var}(B), x \neq x0, x1
\]

\[
M_1(x0) = M_2(x0)
\]

if \( \mathcal{P}(C.mt) = L \), then \( M_1(x1) = M_2(x1) \)

Let \( O_1, O_2 \in \mathcal{O} \) with \( O_1 \downarrow_L = O_2 \downarrow_L \).

\[
\langle (B, 0, M_1, \lambda, \mathcal{O}_1) \rangle \xrightarrow{\sigma} \langle (B, i, \tilde{M}_1, \tilde{S}_1, \lambda, \tilde{O}_1) \rangle \text{ with } B[i] \in \{ \text{神州}, \text{return} \}
\]

and

\[
\langle (B, 0, M_2, \lambda, \mathcal{O}_2) \rangle \xrightarrow{\sigma} \langle (B, j, \tilde{M}_2, \tilde{S}_2, \lambda, \tilde{O}_2) \rangle \text{ with } B[j] \in \{ \text{神州}, \text{return} \}
\]

implies

\[
\tilde{O}_1 \downarrow_L = \tilde{O}_2 \downarrow_L
\]

if \( \tilde{S}_1 = k_1 \cdot \lambda, \tilde{S}_2 = k_2 \cdot \lambda \) and \( \mathcal{R}(C.mt) = L \), then \( k_1 = k_2 \)

Given a security specification \( S(C) \), \( C \) has secure information flow if, for each set \( \text{Objects} \) of instances of classes in \( C \), the execution of any method belonging to a class in \( C \) does not write the low objects in \( \text{Objects} \) with a value depending on information at level \( H \).

**Definition 2 (secure information flow for a set of classes)** Let \( C \) be a set of class definitions and \( S(C) \) a security specification for \( C \). \( C \) has secure information flow with respect to \( S \) if, for any set \( \text{Objects} \) of instances of classes in \( C \):
Note that the secure information flow property considers the value of the low objects. Thus it is defined with respect to a security specification and not to a particular assignment of a security level to the method parameters and result values. Any pair \( \mathcal{P}, \mathcal{R} \) making secure \( \mathcal{C} \) w.r.t \( \mathcal{S} \) can be taken as correct.

4 The transformation algorithm

To check secure information flow we use a transformational approach. Let \( \mathcal{S} \) be a security specification for a set \( \mathcal{C} \) of classes, and \( \mathcal{P}, \mathcal{R}: \text{Mtds}(\mathcal{C}) \rightarrow \Sigma \) be assignments of a security level to the parameter and the return value of the methods in \( \mathcal{C} \). We transform the set of class files produced by the Java compiler for \( \mathcal{C} \) into another set of class files that, when abstractly executed by the Verifier, reveals violations of the secure information flow.

We use the class mechanism of Java to implement the security levels: \( \{[|\mathcal{C}|]\}_{\mathcal{S}, \mathcal{P}, \mathcal{R}} \) has a class file for each security level in \( \Sigma \), and the class hierarchy implements the order between security levels: if level \( \sigma_1 \) is less than level \( \sigma_2 \), then \( \sigma_1 \) belongs to the hierarchy rooted at \( \sigma_2 \). In our simplified case with two classes \( \mathcal{H} \) and \( \mathcal{L} \) we have:

```java
class H {
    public H h;
    Mtds(H)
};
class L extends H {
    public L l;
    Mtds(L)
};
```

They form a hierarchy rooted at \( \mathcal{H} \). Class \( \mathcal{H} \) has only one attribute ( \( \mathcal{h} \) ) with type \( \mathcal{H} \) and class \( \mathcal{L} \) has an attribute ( \( \mathcal{L} \) ) with type \( \mathcal{L} \). \( \text{Mtds}(\mathcal{H}) \) and \( \text{Mtds}(\mathcal{L}) \) contain the methods of high classes and low classes, respectively. We suppose that all methods in \( \text{Mtds}(\mathcal{H}) \cup \text{Mtds}(\mathcal{L}) \) have different names. This is achieved by including the original class name in the method name. For each method declaration \( mt \) of a class \( \mathcal{C} \) such that \( \mathcal{S}(\mathcal{C}) = \sigma \), there is in \( \text{Mtds}(\sigma) \) the declaration of a method \( \mathcal{R}(\mathcal{C}, mt) \ C_{\text{mt}}(\mathcal{P}(\mathcal{C}, mt)) \).

The bytecode of each method \( C_{\text{mt}} \) in \( \mathcal{C} \) is obtained from the original bytecode produced by the compiler for \( mt \) by a transformation algorithm. Values of the original bytecode are substituted by references to a class representing a security level, i.e. \( \mathcal{H} \) or \( \mathcal{L} \). The transformation guarantees that when the bytecode is abstractly executed by the Verifier, a value indicates the least upper bound of the security levels of the information flows, both explicit and implicit, the value depends on. The security level of a value assigned to a variable or assigned to a field of an object, or pushed onto the stack, by an instruction at address \( k \), depends on the security levels of the open implicit flows at \( k \). A new implicit flow is opened whenever a branching instruction is executed and it is closed when the ipd of such instruction is executed.

Note that also the calls to other methods performed under an implicit flow depend on it. Since the Verifier handles each method independently from the other methods, we cannot know locally the implicit flow under which the method under verification may be called. To be sure that no illegal information flow may occur, we impose the constraint that no method call is executed under a high implicit flow.

Let \( \{x_1, \cdots, x_n\} \) be a set of local variables holding object types representing security levels. Figure 3 shows a piece of code that, when abstractly executed by the Verifier, leaves on top of the stack the least upper bound (\( \sqcup \) ) of the reference types held by the variables \( \{x_1, \cdots, x_n\} \). The code is composed of a sequence of "dummy" conditional instructions. Starting from the value of \( x_1 \) on top of the stack,
\[ \cup \{x\} = \text{aload} \ x \]
\[ \cup \{x_1, \ldots, x_n\}, n \geq 2 = \text{aload} \ x_1 \]
\[ \quad \quad \text{acast, null} \]
\[ \quad \quad \text{ifnull} \ A_1 \]
\[ \quad \quad \text{pop} \]
\[ \quad \quad \text{aload} \ x_2 \]
\[ A_1 : \quad \text{acast, null} \]
\[ \quad \quad \text{ifnull} \ A_2 \]
\[ \quad \quad \text{pop} \]
\[ \quad \quad \text{aload} \ x_3 \]
\[ A_2 : \quad \cdots \]
\[ \quad \cdots \]
\[ A_{n-1} : \quad \text{acast, null} \]
\[ \quad \quad \text{ifnull} \ A_n \]
\[ \quad \quad \text{pop} \]
\[ \quad \quad \text{aload} \ x_n \]
\[ A_n : \]

Figure 3: Performing the \( \cup \) of security levels

one branch of the first \text{ifnull} instruction substitutes the top of the stack with the value of \( x_2 \), while the other branch does nothing. When the two branches join, the Verifier calculates the new state having as top element of the stack the least upper bound between \( x_1 \) and \( x_2 \). If for example, \( x_1 \) holds a reference type to class \( L \) and \( x_2 \) holds a reference type to class \( H \), the least upper bound is a reference to \( H \), which is the supertype of both classes. The procedure is iterated for the other variables.

We now describe the bytecode transformation algorithm.

**Step 1:** prepare variables for low and high security values

We add two variables \( wL \) and \( wH \) and we initially assign them an object of class \( L \) and \( H \), respectively. These variables are used by the transformation algorithm whenever a high or low value is required. Note that the register holding the method parameter is automatically set by the Verifier to the security level of the parameter, as specified by \( P \).

\[
\begin{align*}
\text{new} & \ L \\
\text{dup} & \\
\text{invoke} & \ L. < \text{init} > \\
\text{astore} & \ wL \\
\text{new} & \ H \\
\text{dup} & \\
\text{invoke} & \ H. < \text{init} > \\
\text{astore} & \ wH
\end{align*}
\]

**Step 2:** prepare variables for implicit flows

We add a variable \( w_k \) for each \( k: \text{ifge} L \) or \( k: \text{ifnull} L \) instruction occurring in \( B \): \( w_k \) will record the security level of the implicit flow started at address \( k \). Initially we assign all of these variables an object of class \( L \).
aload $w_L$
astore $w_k$, 
...  
aload $w_L$
astore $w_{k_n}$

**Step 3**: transform each instruction in the bytecode

We now present the transformation of the most representative instructions. The transformation of an instruction at label $k$ takes into account the security level of all open implicit flows at $k$. Given $k \in \mathcal{L}(B)$, we denote by $DEP(k)$ the subset of $\{w_j\}$ corresponding to the implicit flows that are open when the instruction at label $k$ is executed.

$$DEP(k) = \{ w_j \mid j \text{ is the label of a branching instruction and } k \text{ belongs to a path from } j \text{ to } ipd(j), k \neq ipd(j), k \neq j \}$$

The least upper bound of $DEP(k)$ represents the least upper bound among the open implicit flows to which the instruction labeled by $k$ belongs. To compute least upper bounds, we use the code defined in Figure 3. Moreover, we assume $\sqcup \emptyset = \text{aload } w_L$, that pushes the low security level onto the stack.

A const instruction is transformed into a piece of code pushing onto the stack the least upper bound of the levels of the open implicit flows.

$$\{ k : \text{aconst } d \} = k : \sqcup DEP(k)$$

An instruction corresponding to an operation with two operands is transformed into a piece of code pushing onto the stack the least upper bound among the operands and the levels of the open implicit flows.

$$\{ \mid k : \text{aop } \} = k : \begin{cases} \text{astore } w_1 \\ \text{astore } w_2 \\ \sqcup (DEP(k) \cup \{w_1, w_2\}) \end{cases}$$

The code corresponding to a load instruction pushes onto the stack the $\sqcup$ among the the value of the variable and the levels of the open implicit flows.

$$\{ \mid k : \text{aload } x \} = k : \sqcup (DEP(k) \cup \{x\})$$

A store instruction is transformed into a piece of code storing into the referred variable the $\sqcup$ among the top of the stack and the levels of the open implicit flows. If $DEP(k) = \emptyset$ (i.e. the instruction does not belong to the scope of any conditional instruction) we can optimize the transformation by deleting the piece of code between the lines. The same occurs for the successive transformations.

$$\{ \mid k : \text{astore } x \} = k : \begin{cases} \text{astore } x \\ \sqcup (DEP(k) \cup \{x\}) \\ \text{astore } j \end{cases}$$

The code corresponding to ifge $L$ opens the scope of an implicit flow having as security level the $\sqcup$ among the top of the stack and the levels of the already open implicit flows. The level of this implicit flow is stored into variable $w_k$, where $k$ is the label of the instruction.

$$\{ \mid k : \text{ifge } L \} = k : \begin{cases} \text{astore } w_k \\ \sqcup (DEP(k) \cup \{w_k\}) \\ \text{dup} \\ \text{astore } w_k \\ \text{ifnull } L \end{cases}$$
If the specification $S$ assigns to class $C$ of the source code the security level $S(C) = \sigma$, the piece of code corresponding to `getfield C.f` pushes onto the stack the \( \sqcup \) among: a) $\sigma$ (the level of field $f$ of class $C$); b) the top of the stack (the level of the reference to an object of class $C$); c) the levels of the open implicit flows.

\[
\{ k : \text{getfield } C.f \} = k : \text{astore } w \\
\sqcup(DEP(k) \cup \{ w \} \cup \\
\{(S(C) == H) ? wH : wL\}
\]

The piece of code corresponding to `putfield C.f` performs a putfield operation on field $h$ of class $H$, or on field $l$ of class $L$, according to $S(C)$. Note that the Verifier checks that the level of the reference (the element below the top of the stack) and the levels of the open implicit flows are dominated by $S(C)$. The value put is the \( \sqcup \) among: a) the value on top of the stack (the value to be stored); b) the levels of the open implicit flows. In fact the value to be stored in the field depends also on the implicit flows. In this way the Verifier checks that the level of the assigned value is dominated by the level of the field ($S(C)$).

\[
\{ \text{putfield } C.f \} = k : \text{astore } wp \\
\text{astore } wo \\
\sqcup(DEP(k) \cup \{ wo \} \cup \\
\{(S(C) == H) ? H : H : L \}
\]

The transformation of a `return` instruction pushes on top of the stack the \( \sqcup \) among the open implicit flows and the top of the stack; afterwards, the bytecode returns.

\[
\{ k : \text{return} \} = k : \text{astore } wo \\
\sqcup(DEP(k) \cup \{ wo \}) \\
\text{return}
\]

The piece of code corresponding to `invoke C.mt` first write field $l$ of class $L$ with the \( \sqcup \) of the levels of the open implicit flows. This forces the Verifier to reject the bytecode if the call is performed under an high implicit flow. After, a method is invoked of class $L$ or $H$ according to $S(C)$. The invoked method is that corresponding to $mt$.

\[
\{ k : \text{invoke C.mt} \} = k : \sqcup\text{DEP}(k) \\
\text{putfield } L.l \\
\text{invoke}((S(C) == H) ? \\
\text{H.C.mt} : L.C.mt)
\]

The following proposition states the correctness of the method. 

**Proposition 1** A set $C$ of class files has secure information flow under a security specification $S : C \rightarrow \Sigma$ if $P : Mtds(C) \rightarrow \Sigma$ and $R : Mtds(C) \rightarrow \Sigma$ exist such that $\{ [C] \}_{S,P,R}$ is accepted by the Verifier.

The proof exploits the work of some of the authors and others [5, 6, 7] showing an abstract interpretation approach to secure information flow in Java bytecode: there an abstract semantics is given for the language, correctly capturing the secure
information flow property. The first part of the proof shows that the abstract semantics of the source code is equivalent to the abstract semantics of the transformed code. After, we show that the Verifier, when applied to \( \{C\}_{S,P,R} \), agrees with the abstract semantics of \( \{C\}_{S',P,R} \). This guarantees that the Verifier accepts only programs judged correct by the abstract semantics.

5 Example

---

**Figure 4**: The tax program

Let us consider, as an example, a tax program whose execution produces reports on the amount of the tax of the users. The program needs to access sensitive information (income). Moreover, we assume it also needs to access over the internet to find information related to the correct calculation of the amount of the tax. There is a danger that the method may send over the internet information regarding the amount of the income. Figure 4 reports the class schema of the example. In the figure, an arrow between an attribute of a class \( A \) and a class \( B \) indicates that each instance of class \( A \) has a reference to an instance of class \( B \). Figure 5 illustrates the source code of the schema.

We suppose that each class has only one instance. An instance of class \( \text{CalcTax} \) depends on an instance of class \( \text{Income} \), representing the income of the user, and an instance of class \( \text{Net} \), needed to query information over the internet. The method \( \text{calc}(\cdot) \) of \( \text{CalcTax} \) has an instance of class \( \text{Tax} \) as output parameter. This method applies a lower rate or a higher rate to the income according to a threshold. The actual values of the threshold and the rates are retrieved from the internet. The method first invokes method \( \text{get}(\cdot) \) of class \( \text{Income} \) whose execution queries the amount of the income of the user; then it invokes method \( \text{get}(\cdot) \) of class \( \text{Net} \) to find the threshold. Successively, it invokes method \( \text{get}(\cdot) \) of class \( \text{Net} \) a second time to retrieve the correct rate to be applied. Finally, it calculates the new tax amount of the user and updates the tax module by calling the method \( \text{add}(\cdot) \) of class \( \text{Tax} \).

Figure 6 reports the class file of class \( \text{CalcTax} \). For clarity, we use the syntax of Jasmin [12] instead of that used by Java compiler. Jasmin is a Java Assembler Interface. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files suitable for loading into a Java interpreter.

The syntax used in Section 4 differs from that of Jasmin since in Section 4 the type of the parameters and the result from methods definitions and invocations are omitted and \texttt{invoke} is used to denote either \texttt{invokeonvirtual} or \texttt{invokevirtual}.
Consider the system composed of the classes Tax, Income and Net and the security specification \( S(\text{Income}) = S(\text{Tax}) = H \) and \( S(\text{Net}) = L \). We assume that these classes have already been verified to satisfy the secure information flow property and that the only secure assignment of security levels to methods parameters and results is the following: \( P(\text{Tax}.\text{add}) = P(\text{Income}.\text{Income}) = H, P(\text{Net}.\text{get}) = L \), \( R(\text{Tax}.\text{add}) = R(\text{Income}.\text{get}) = R(\text{Tax}.\text{get}) = H, R(\text{Net}.\text{get}) = L \).

Let us assume we want to add class CalcTax to our system. Moreover, we want the class to have a low security level and secure information flow property to be satisfied. We have to find an assignment of security levels to methods parameters and results of the new class, that agree with \( R \) and \( P \) for the old classes as defined above. If such assignment does exist, the class can be securely added to our system, i.e., it does not violate security. Otherwise, at least one of the methods of the new class may cause a disclosure of sensitive information. CalcTax has a single method calc with a unique parameter and no return value. We apply the transformation for the two possible assignments of the security level to the method parameter. Since we want CalcTax to have a low level, it is abstracted into class L and the calc method becomes a method of class L.

In Figure 8 there is the class file of class L when the level assigned to the parameter of method calc is low. The class file obtained when \( P(\text{calc}) = H \) differs only in the declaration of the method (CalcTax\_calc( LL; )V is substituted by CalcTax\_calc( LH; )V ).

Once obtained the class file for class L and class H, we execute the program shown in Figure 7 to force the verification of the transformed class files. We use Sun's JDK 1.3.1 [1]. The program creates one instance of class H and one instance of class L. When the Verifier examines these instructions, it verifies all the methods of both classes. If verification succeeds, the program is executed. In this case no method is called and the program terminates. If the program terminates without
Figure 6: The Jasmin source code of class CalcTax.

error indication, the verification has been successful and the secure information flow property holds. In our example, the Verifier signals an error at label 120 in both cases ($P(calc) = L$ and $P(calc) = H$). In fact, the method calc invokes method get(L) with actual parameter 1 or 2 according to the condition at label 110 ($amount < net.get(0)$). In turn, amount has been obtained by invokevirtual at label 112 from a method whose return value is high. Therefore, the actual parameter 1 or 2 is an high information. In this case information is leaked since, looking at the code saved into a field of object net and sent over the internet, one may understand if the amount was below or over the threshold.

Figure 7: Jasmin source code of class Test.
Figure 8: Jasmin source code of class L

6 Related work and conclusions

The problem of secure information flow has been extensively studied for programs written in structured high level languages [17, 13, 4, 5]. The works [6, 7, 8] address secure information flow for stack based languages, and in particular Java bytecode. All of these works are based on a combination of model checking and abstract interpretation: an abstract model of the execution is built, that is successively model checked for a set of security properties.

The method presented in the paper uses the standard bytecode Verifier to check secure information flow. Verification is performed on the code output by a transformation algorithm applied to the original code. The algorithm is very simple and
its complexity is linear in the number of instructions of the bytecode.

The advantage of the proposed method is that it does not rely on any extra verification tool, but can be performed inside a Java environment. Thus it is applicable also to Java smart cards, either for off-card or for on-card verification [14, 15]. The method is not applicable if the security levels are not ordered hierarchically: in fact security levels are modeled with classes and the Java class model does not allow multiple inheritance.

As a future work, we intend to extend the method to handle dynamic object creation and subroutines.

References


