Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 237–254
www.elsevier.com/locate/entcs

# A Space-Aware Bytecode Verifier for Java Cards

Cinzia Bernardeschi[1] Giuseppe Lettieri[2] Luca Martini[3]
Paolo Masci[4]

*Dipartimento di Ingegneria dell'Informazione*
*Università di Pisa*
*Pisa, Italy*

**Abstract**

The bytecode verification is a key point of the security chain of the Java Platform. This feature is optional in many embedded devices since the memory requirements of the verification process are too high. In this paper we propose a verification algorithm that remarkably reduces the use of the memory by performing the verification during multiple specialized passes. The algorithm reduces the type encoding space by operating on different abstractions of the domain of types. The results of the experiments show that this bytecode verification can be performed directly on small memory systems.

*Keywords:* Embedded systems, data-flow analysis, type correctness

## 1 Introduction

The Java programming language was born in the early 90s in order to meet flexible and highly reliable smart electronic device programming requirements. It is used in a growing number of fields and lately also in the embedded system world. Among the embedded devices, Java Cards represent an interesting research challenge since they have limited resources but require high security

[1] Email:cinzia.bernardeschi@iet.unipi.it
[2] Email:giuseppe.lettieri@iet.unipi.it
[3] Email:luca.martini@iet.unipi.it
[4] Email:paolo.masci@iet.unipi.it


doi:10.1016/j.entcs.2005.02.027

features. Actually, a Java Card is a Smart Card running a Java Virtual Machine (VM), the Java Card Virtual Machine (JCVM), and it is going to become a secure token in various fields, such as banking and public administration.

The JCVM is the core of the Java Card: it is a software CPU with a stack-based architecture that creates an execution environment between the device and the programs (Java Card Applets). The JCVM guarantees hardware-independence and enforces the security constraints of the sandbox model. In particular, the Java bytecode Verifier is one of the key components of the sandbox model: the Java Card Applets are compiled in a standardized compact code called Java Card bytecode and the Verifier checks the correctness of the code before it is executed on the JCVM.

The Java bytecode Verifier performs a data-flow analysis on the bytecode in order to ensure the type-correctness of the code. For example, it ensures that the program does not forge pointers, e.g. by using integers as object references.

Bytecode verification enables *post issuance* download of Java Card Applets, even if they are not taken from the card vendor but from different sources. Bytecode verification can be performed off-card or on-card. However, because of the strong memory constraints of the Java Cards, the bytecode verification is unfeasible directly on-card in its standard form.

To perform verification on-card, many approaches have been proposed in the literature: they modify the standard verification process so that its implementation becomes suitable for memory constrained devices. These proposals are reviewed in Section 3.1.

In this paper, we propose and evaluate an alternative approach that is based on checking if the bytecode is correct by means of a progressive analysis that requires much less memory than the standard one.

## 2 Standard 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 for each class defined in the program. A Java Card applet is obtained by transforming the class files into *cap files* in order to perform name resolution and initial address linking [3]. Hereafter, we will refer to the class files only, since cap files and class files conceptually contain the same information.

A class file is composed by the declaration of the class and by the JVM Language (JVML) bytecode for each class method. The JVML instructions are typed: for example, `iload` loads an integer onto the stack, while `aload` loads a reference.

The Verifier performs a data-flow analysis of the code by executing abstractly the instructions over types instead of over actual values. A Java bytecode verification algorithm is presented in [8]: almost all existing bytecode Verifiers implement that algorithm. The verification process is performed method per method: when verifying a method, the other methods are assumed to be correct. Figure 1 shows the JVML instructions. We assume that subroutines have a unique return address. The following notation is used for the types:

$BasicType = \{int, float, \ldots\}$;
$ReferenceType = ReferenceObject \mid ReferenceArray$;
$AddressType = ReferenceType \mid ReturnAddress$;
$\beta \in BasicType$;
$\bar{\tau} \in \{reference\} \cup BasicType$;
$\tau \in \{Object\} \cup BasicType$;
$[\tau \in ReferenceArray$;
$C \in ClassType$.

The types form a domain, where $\top$ is the top element and $\bot$ is the bottom element. In this domain $\top$ represents either an undefined type or an incorrect type.

Class types $C$ are related as specified by the class hierarchy. Figure 2 shows an example of class hierarchy: `E`, `F` and `G` are user defined classes, with `F` and `G` extending `E`. Not all types are shown.

Each method is indicated by an expression of the form $C.m(\tau_1, \ldots, \tau_n) : \tau_r$, where $C$ is the class which method $m$ belongs to, $\tau_1, \ldots, \tau_n$ are the argument types and $\tau_r$ is the type of the return value. Each method is compiled into a (finite) sequence of bytecode instructions $B$. We use $B[i]$ to indicate the $i$-th instruction in the sequence, with $B[0]$ as the entry point.

Figure 3 shows the rules of the standard verification algorithm. Each rule has a precondition, which contains a set of constraints, and a postcondition, which contains the transition from the before to the after state of an instruction. For example, an `iload` $x$ instruction requires a non-full stack and the *int* type to be associated to register $x$; its effect is to push *int* onto the stack. We have used $\lambda$ to indicate the empty stack and, to simplify, the rules show only the constraints on the types.

The goal of the verification is to assign to each instruction $i$ a mapping $M^i$ from the registers to the types, and a mapping $s^i$ from the elements in the stack to the types. These mappings represent the state $St^i = (M^i, s^i)$ in which the instruction $i$ is executed. A state $St^i$ is computed as the least upper bound

| | |
|---|---|
| $\beta op : \beta'$ | Takes two operands of type $\beta$ from the stack, and pushes the result (of type $\beta'$) onto the stack. |
| $\beta$`const` $d$ | Loads a constant $d$ of type $\beta$ onto the stack. |
| $\tau$`load` $x$ | Loads the value of type $\tau$ from register $x$ to the stack. |
| $\bar{\tau}$`store` $x$ | Takes a value of type $\bar{\tau}$ from the stack and stores it into register $x$. |
| `if`$cond$ $L$ | Takes a value of type *int* from the stack, and jumps to $L$ if the value satisfies $cond$ |
| `goto` $L$ | Jumps to $L$. |
| `new` $C$ | Creates an instance of class $C$ and adds a reference to the created instance on top of the stack. |
| `newarray` $\tau$ | Creates an instance of an array of class $\tau$ and adds a reference to the instance on top of the stack. |
| $\beta$`aload` | Takes an array reference and an integer index from the stack. The array reference is of type $[\beta$. Loads on the stack the reference of type $\beta$ saved on the index position in the referenced array. |
| $\beta$`astore` | Takes an array reference, an integer index and a reference from the stack. The array reference is of type $[\beta$, the reference of type $\beta$. The reference is saved in the referenced array on the index position. |
| `aaload` | Takes an array reference and an integer index from the stack. The array reference is of type $[C$. Loads on the stack the reference of type $C$ saved on the index position in the referenced array. |
| `aastore` | Takes an array reference, an integer index and a reference from the stack. The array reference is of type $[C$, the reference of type $C$. The reference is saved in the referenced array on the index position. |
| `getfield` $C.f : \tau$ | Takes an object reference of class $C$ from the stack; fetches field $f$ (of type $\tau$) of the object and loads the field on top of the stack. |
| `putfield` $C.f : \tau$ | Takes a value of type $\tau$ and an object reference of class $C$ from the stack; saves the value in field $f$ of the object. |
| `invoke` $C.m(\tau_1, \ldots, \tau_n) : \tau_r$ | Takes the values $v_1, \ldots, v_n$ (of types $\tau_1, \ldots, \tau_n$) and an object reference of class $C$ from the stack. Invokes method $C.m$ of the object with actual parameters $v_1, \ldots, v_n$; places the method return value (of type $\tau_r$) on top of the stack. |
| $\tau$`return` | Takes the value of type $\tau$ from the stack and terminates the method. |
| `jsr` $L$ | Places the address of the successor on the stack and jumps to L. |
| `ret` $x$ | Jumps to the address specified in register $x$. |

Fig. 1. Instruction set

of all the interpreter states $Q^i = \langle i, M, s \rangle$ obtained while applying the rules. The rules are applied within a standard fixpoint iteration which uses a worklist algorithm. Until the worklist is not empty, an instruction $B[i]$ is taken from the worklist and the states of the successor program points are computed. If

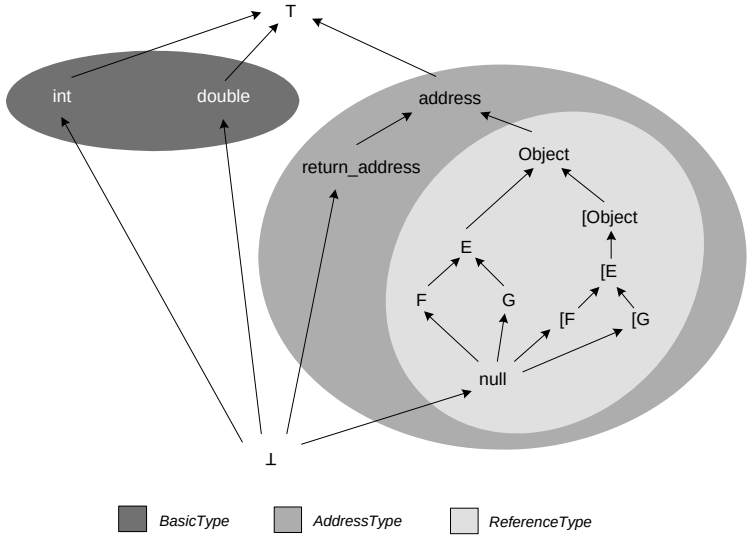


Fig. 2. Some types and the subtyping relation. Not all types are shown.

the computed state for a successor program point $j$ changes (either the state at $j$ has not been computed yet or the state already computed differs), $B[j]$ is added to the worklist. The fixpoint is reached when the worklist is empty. Initially, the worklist contains only the first instruction of the bytecode. The initial types on the stack and registers represent the state at the method entrance: the stack is empty and the type of the registers are set as specified by the method signature (the registers not associated with the method arguments hold the undefined type $\top$).

As a consequence of the algorithm, the state at a program point of the instructions that represent a merge point between control paths (i.e. having more than one predecessor in the control flow graph) is the *least upper bound* of the after state of all the predecessors. If, for example, register $x$ has type *int* on a path and type $\top$ on another path, the type of $x$ at the merge point is $\top$. The least upper bound of stacks and memories is done pointwise. The pointwise least upper bound between stacks requires stacks of the same height (otherwise there is a type error).

## 3 On-card bytecode verification

The verification algorithm is expensive for both computation time and memory space since a data-flow analysis of the code is performed. The before state of each instruction must be recorded during the analysis. As an optimization, Sun's bytecode Verifier maintains the state of each program point that is either

$$\textbf{op}\quad \frac{B[i] = \beta\, op\colon \beta',\quad v_1 \sqsubseteq \beta,\quad v_2 \sqsubseteq \beta}{\langle i,\, M,\, v_1 v_2 s\rangle \longrightarrow \langle i+1,\, M,\, \beta' s\rangle}$$

$$\textbf{const}\quad \frac{B[i] = \tau\texttt{const}\ d}{\langle i,\, M,\, s\rangle \longrightarrow \langle i+1,\, M,\, \tau s\rangle}$$

$$\textbf{load}\quad \frac{B[i] = \tau\texttt{load}\ x,\quad M(x) \sqsubseteq \tau}{\langle i,\, M,\, s\rangle \longrightarrow \langle i+1,\, M,\, M(x) s\rangle}$$

$$\textbf{store}\quad \frac{B[i] = \bar{\tau}\texttt{store}\ x,\quad v \sqsubseteq \bar{\tau}}{\langle i,\, M,\, v s\rangle \longrightarrow \langle i+1,\, M[x/v],\, s\rangle}$$

$$\textbf{aload}\quad \frac{B[i] = \tau\texttt{aload},\ v_1 = \lfloor v',\ v' \sqsubseteq \tau,\ v_2 \sqsubseteq \mathit{int}}{\langle i,\, M,\, v_1 v_2 s\rangle \longrightarrow \langle i+1,\, M,\, v' s\rangle}$$

$$\textbf{astore}\quad \frac{B[i] = \tau\texttt{astore},\ v_1 = \lfloor v',\ v' \sqsubseteq \tau,\ v_2 \sqsubseteq \mathit{int},\ v_3 \sqsubseteq \tau}{\langle i,\, M,\, v_1 v_2 v_3 s\rangle \longrightarrow \langle i+1,\, M,\, s\rangle}$$

$$\textbf{if}^{tt}\quad \frac{B[i] = \texttt{if}\, cond\ L,\quad v \sqsubseteq \mathit{int}}{\langle i,\, M,\, v s\rangle \longrightarrow \langle L,\, M,\, s\rangle}$$

$$\textbf{if}^{ff}\quad \frac{B[i] = \texttt{if}\, cond\ L,\quad v \sqsubseteq \mathit{int}}{\langle i,\, M,\, v s\rangle \longrightarrow \langle i+1,\, M,\, s\rangle}$$

Fig. 3. The rules of the standard verifier.

the target of a branch or the entry point of an exception handler [7]. The set of states saved during the data-flow analysis is called *dictionary*.

The on-card bytecode verification is identical to a standard verification, however special optimizations must be used since cards have limited resources. Commercial 2004 Java Cards typically provide 1-4KB of RAM, 32-64KB of persistent writable memory and 128-256KB of ROM. Only the RAM should be used to store temporary data structures because the persistent memory is too slow and allows a limited number of writing cycles.

**goto**
$$\frac{B[i] = \texttt{goto}\ L}{\langle i,\ M,\ s\rangle \longrightarrow \langle L,\ M,\ s\rangle}$$

**new**
$$\frac{B[i] = \texttt{new}\ C}{\langle i,\ M,\ s\rangle \longrightarrow \langle i+1,\ M,\ Cs\rangle}$$

**newarray**
$$\frac{B[i] = \texttt{newarray}\ \tau,\quad v \sqsubseteq int}{\langle i,\ M,\ vs\rangle \longrightarrow \langle i+1,\ M,\ [\tau s\rangle}$$

**getfield**
$$\frac{B[i] = \texttt{getfield}\ C.f{:}\tau,\quad v \sqsubseteq C}{\langle i,\ M,\ vs\rangle \longrightarrow \langle i+1,\ M,\ \tau s\rangle}$$

**putfield**
$$\frac{B[i] = \texttt{putfield}\ C.f{:}\tau,\quad v_1 \sqsubseteq \tau,\quad v_2 \sqsubseteq C}{\langle i,\ M,\ v_1 v_2 s\rangle \longrightarrow \langle i+1,\ M,\ s\rangle}$$

**invoke**
$$\frac{B[i] = \texttt{invoke}\ C.m(\tau_1,\ldots,\tau_n){:}\tau_r,\quad v_j \sqsubseteq \tau_j\ (1 \le j \le n),\quad v \sqsubseteq C}{\langle i,\ M,\ v_1 \cdots v_n v s\rangle \longrightarrow \langle i+1,\ M,\ \tau_r\rangle}$$

**return**
$$\frac{B[i] = \tau\texttt{return}\ v \sqsubseteq \tau,\ v \sqsubseteq \tau_r}{\langle i,\ M,\ v\rangle \longrightarrow \langle -1,\ M,\ \lambda\rangle}$$

**jsr**
$$\frac{B[i] = \texttt{jsr}\ L}{\langle i,\ M,\ s\rangle \longrightarrow \langle L,\ M,\ r_a s\rangle}$$

**ret**
$$\frac{B[i] = \texttt{ret}\ x,\ M(x) \sqsubseteq \mathit{ReturnAddress}}{\langle i,\ M,\ s\rangle \longrightarrow \langle r_a,\ M,\ s\rangle}$$

Fig. 4. The rules of the standard verifier (continued).

### *3.1 Related work*

Many approaches have been presented to develop an on-card Verifier.

Rose and Rose [11] propose a solution based on a certification system (inspired by the PCC, 'Proof Carrying Code' work by Necula [10]). The verification process is split in two phases: lightweight bytecode certification (LBC) and lightweight bytecode verification (LBV). LBC is performed off-card and produces a certificate that must be distributed with the bytecode. LBV is performed on-card, and it is a linear verification process that uses the certificate to assure that the bytecode is safe. LBV is currently used in the KVM of Sun's Java 2 Micro Edition.

Leroy, in [6], proposes to reduce memory requirements with an off-card code transformation, also known as 'code normalization'. The transformed code complies with the following constraints: every register contains the same type for all method instructions and the stack is empty at the merge points. The verification of a 'normalized' code is not expensive: only one global state is required since the type of the registers never change.

Deville and Grimaud [4] propose to use the persistent memory for storing

the data structures needed for the verification process. Their strategy holds all data structures in RAM as long as possible, and swaps them in persistent memory when RAM space is missing. A special type of encoding is proposed since persistent memory cells have a limited number of writing cycles.

A verification algorithm that reduces the size of the dictionary by allocating dynamically its entries has been presented in [2]. The algorithm assigns a lifetime to the dictionary entries, splits the method code into control regions [5] and analyzes the regions one-by-one. Each region is analyzed by applying the standard verification algorithm: the size of the dictionary is reduced since unnecessary entries are never kept in the memory.

## 4 Our approach

In this paper we present a space-aware bytecode verification algorithm that reduces the size of the dictionary by performing a progressive analysis on different abstractions of the domain of types.

### *4.1 The multipass concept*

The multipass verification algorithm is the standard algorithm performed in many specialized passes. Each pass is dedicated to a type. The dictionary size is reduced since, during each pass, the abstract interpreter needs to know only if the type (saved in a register or in a stack location) is *compatible* with the pass or not. The compatibility of the type is given by the type hierarchy. Actually, only one bit is needed to specify the type of the data saved in registers and stack locations.

The analysis is performed on the whole bytecode for every pass. The number of passes depends on the instructions contained in the method: one pass is needed for each type used by the instructions. Additional passes are also needed to check the initialization of objects (this concept is explained in Section 4.4).

The multipass verification is possible since the bytecode instructions are typed and the number of types is limited: basic types (int, byte, ...), reference types (the ones listed in the constant pool) and return address type.

An example of data-flow analysis (dfa) performed with the standard approach and with the multipass approach is shown in Figure 5. We can notice the different encoding strategies for the two approaches: during the standard dfa the types are fully specified. On the other hand, during each multipass dfa pass, a one bit encoding of the types is used.

| static int *debit* (int *aC*) | STANDARD DFA | | MULTIPASS DFA | | | |
|---|---|---|---|---|---|---|
| | | | array of int | | int | |
| | (registers) | (stack) | (registers) | (stack) | (registers) | (stack) |
| 0: iload_0 | { int T } | ( ) | { 0 0 } | ( ) | { 1 0 } | ( ) |
| 1: getstatic AA.h I | { int T } | ( int ) | { 0 0 } | ( 0 ) | { 1 0 } | ( 1 ) |
| 4: if_icmpne 13 | { int T } | ( int int ) | { 0 0 } | ( 0 0 ) | { 1 0 } | ( 1 1 ) |
| 7: getstatic AA.c [I | { int T } | ( ) | { 0 0 } | ( ) | { 1 0 } | ( ) |
| 10: iconst_0 | { int T } | ( [int ) | { 0 0 } | ( 1 ) | { 1 0 } | ( 0 ) |
| 11: iaload | { int T } | ( int [int ) | { 0 0 } | ( 0 1 ) | { 1 0 } | ( 1 0 ) |
| 12: istore_1 | { int T } | ( int ) | { 0 0 } | ( 0 ) | { 1 0 } | ( 1 ) |
| 13: getstatic AA.c [I | { int T } | ( ) | { 0 0 } | ( ) | { 1 0 } | ( ) |
| 16: iconst_1 | { int T } | ( [int ) | { 0 0 } | ( 1 ) | { 1 0 } | ( 0 ) |
| 17: iaload | { int T } | ( int [int ) | { 0 0 } | ( 0 1 ) | { 1 0 } | ( 1 0 ) |
| 18: istore_1 | { int T } | ( int ) | { 0 0 } | ( 0 ) | { 1 0 } | ( 1 ) |
| 19: iconst_1 | { int int } | ( ) | { 0 0 } | ( ) | { 1 1 } | ( ) |
| 20: ireturn | { int int } | ( int ) | { 0 0 } | ( 0 ) | { 1 1 } | ( 1 ) |

Fig. 5. An example of standard data-flow and multipass data-flow.

### *4.2 The rules*

First we formulate the operations performed by the standard Verifier and then we give a formal description of the multipass algorithm.

**Definition 4.1** [transition system] A transition system $L$ is a triple $(\mathcal{Q}, \rightarrow, Q^0)$, where $\mathcal{Q}$ is a set of states, $Q^0 \in \mathcal{Q}$ is the initial state, and $\longrightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ is the transition relation.

We say that there is a transition from $Q$ to $Q'$ if $(Q, Q') \in \longrightarrow$, and we write $Q \longrightarrow Q'$. We denote with $\stackrel{*}{\longrightarrow}$ the reflexive and transitive closure of $\longrightarrow$. We say that a state $Q \in \mathcal{Q}$ is a final state of the transition system if and only if no $Q'$ exists so that $Q \longrightarrow Q'$ (we write $Q \not\longrightarrow$).

Let $\mathcal{D}$ be the set of types, $\mathcal{V}$ the set of registers, $\mathcal{A}$ the set of bytecode addresses, $\mathcal{I}$ the set of bytecode instructions, $\mathcal{M} : \mathcal{V} \rightarrow \mathcal{D}$ the set of memories ($\mathcal{M}$ associates a type to every register) and $\mathcal{S}$ the set of finite sequences of elements of $\mathcal{D}$ ($\mathcal{S}$ associates a type to every element in the stack locations). An interpreter state is defined as a triple $\langle i, M, s \rangle$, where $i \in \mathcal{A}$, $M \in \mathcal{M}$ and $s \in \mathcal{S}$. Given a method $m$, we define $B_m : \mathcal{A} \rightarrow \mathcal{I}$ as the instruction sequence of the method. We assume that a lattice $\mathcal{L} = (\mathcal{D}, \sqsubseteq)$ of types is defined (it is shown in Figure 2). The $\sqsubseteq$ relation is extended pointwise to the sets $\mathcal{M}$, $\mathcal{S}$.

#### *4.2.1 Standard rules*

The rules shown in Figure 3 define a $\longrightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ relation.

Given a method $C.m(\tau_1, \ldots, \tau_n) : \tau_r$, which contains a bytecode sequence $B$, we define the initial memory $M^0 \in \mathcal{M}$ such that $M^0(0) = C$, $M^0(i) = \tau_i$ if $1 \leq i \leq n$ and $M^0(i) = \top$ otherwise ($M^0$ assigns types to the registers according to the method parameters). We define with $(Q, \longrightarrow, \langle 0,\ M^0,\ \lambda\rangle)$ the transition system defined by the rules in Figure 3, starting from the initial state $\langle 0,\ M^0,\ \lambda\rangle$.

This transition system is an abstract interpretation of the execution of the bytecode. The Verifier implicitly uses this abstract interpretation to associate a state with each instruction of the bytecode. During the data-flow analysis, the state associated to the instruction at offset $i$ is $St^i = (M^i, s^i)$, where $(M^i, s^i)$ is the least upper bound of memories and stacks of all the interpreter states $\langle i,\ M,\ s\rangle$ that appear in the transition system.

The standard Verifier gives an error if, starting from state $\langle 0,\ M^0,\ \lambda\rangle$, it produces a $\langle i,\ M,\ s\rangle$ state where no further transition can be made and $i \neq -1$. Formally, we write: $\langle 0,\ M^0,\ \lambda\rangle \stackrel{*}{\not\longrightarrow}$

### *4.2.2 Multipass rules*

We are now going to give a formal description of the multipass algorithm.

For each pass type $p \in \mathcal{D}$, we define a lattice $\mathcal{L}_p = (\mathcal{D}_p, \sqsubseteq_p)$, where $\mathcal{D}_p = \{\top_p, \bot_p\}$ and $\bot_p \sqsubseteq_p \top_p$. We define also function $\alpha_p : \mathcal{D} \to \mathcal{D}_p$ as:

$$\alpha_p(t) = \begin{cases} \bot_p \text{ if } t \sqsubseteq p \\ \top_p \text{ otherwise} \end{cases}$$

which means that the abstraction of a type $t$ is $\bot_p$ if and only if $t$ is assignment compatible with type $p$ of the pass.

For example, if $t \in C$, the abstraction function for class type `E` (refer to the lattice in Figure 2) is the following:

$$\alpha_{\mathtt{E}}(t) = \begin{cases} \bot_{\mathtt{E}} \text{ if } t \in \{\mathtt{F}, \mathtt{G}, \mathrm{null}\} \\ \top_{\mathtt{E}} \text{ otherwise} \end{cases}$$

The $\alpha_p$ function is extended pointwise to the sets $\mathcal{M}$ and $\mathcal{S}$. We define a $(\mathcal{Q}_p, \longrightarrow_p, \langle 0,\ M_p^0,\ \lambda\rangle)$ transition system for each $p \in \mathcal{D}$ type, where $Q_p = \langle i,\ M_p,\ s_p\rangle \in \mathcal{Q}_p, M_p \in \mathcal{M}_p : \mathcal{V} \to \mathcal{D}_p,\ s_p \in \mathcal{S}_p$.

Each transition relation $\longrightarrow_p$ of the multipass is obtained by the corresponding standard transition relation (defined in Figure 3) simply by applying the $\alpha_p$ function to the types and by changing each $\sqsubseteq$ into $\sqsubseteq_p$. The multipass rules are shown in Figure 6. Some rules are explained hereafter.

Let us take into consideration a $\beta op : \beta'$ instruction: `iadd`. In this case $\beta = int$ and $\beta' = int$. The constraints to be checked are $v_1 \sqsubseteq_p \alpha_p(int)$ and $v_2 \sqsubseteq_p$

$$\mathbf{op}_p \quad \frac{B[i] = \beta op{:}\beta', \quad v_1 \sqsubseteq_p \alpha_p(\beta), \quad v_2 \sqsubseteq_p \alpha_p(\beta)}{\langle i,\, M,\, v_1 v_2 s\rangle \longrightarrow_p \langle i+1,\, M,\, \alpha_p(\beta')s\rangle}$$

$$\mathbf{const}_p \quad \frac{B[i] = \tau \texttt{const}\ d}{\langle i,\, M,\, s\rangle \longrightarrow_p \langle i+1,\, M,\, \alpha_p(\tau)s\rangle}$$

$$\mathbf{load}_p \quad \frac{B[i] = \tau \texttt{load}\ x, \quad M(x) \sqsubseteq_p \alpha_p(\tau)}{\langle i,\, M,\, s\rangle \longrightarrow_p \langle i+1,\, M,\, M(x)s\rangle}$$

$$\mathbf{store}_p \quad \frac{B[i] = \bar{\tau} \texttt{store}\ x, \quad v \sqsubseteq_p \alpha_p(\bar{\tau})}{\langle i,\, M,\, vs\rangle \longrightarrow_p \langle i+1,\, M[x/v],\, s\rangle}$$

$$\mathbf{aload}_p \quad \frac{B[i] = \tau \texttt{aload},\ v_1 \sqsubseteq_p \alpha_p(\lceil\tau), \ v_2 \sqsubseteq_p \alpha_p(int)}{\langle i,\, M,\, v_1 v_2 s\rangle \longrightarrow_p \langle i+1,\, M,\, \alpha_p(\tau)s\rangle}$$

$$\mathbf{astore}_p \quad \frac{B[i] = \tau \texttt{astore},\ v_1 \sqsubseteq_p \alpha_p(\lceil\tau), \ v_2 \sqsubseteq_p \alpha_p(int), \ v_3 \sqsubseteq_p \alpha_p(\tau)}{\langle i,\, M,\, v_1 v_2 v_3 s\rangle \longrightarrow_p \langle i+1,\, M,\, s\rangle}$$

$$\mathbf{if}_p{}^{tt} \quad \frac{B[i] = \texttt{if}\, cond\ L, \quad v \sqsubseteq_p \alpha_p(int)}{\langle i,\, M,\, vs\rangle \longrightarrow_p \langle L,\, M,\, s\rangle}$$

$$\mathbf{if}_p{}^{ff} \quad \frac{B[i] = \texttt{if}\, cond\ L, \quad v \sqsubseteq_p \alpha_p(int)}{\langle i,\, M,\, vs\rangle \longrightarrow_p \langle i+1,\, M,\, s\rangle}$$

Fig. 6. The rules of the multipass verifier.

$\alpha_p(int)$. Notice that the constraints may fail only during the *int* pass, since $\alpha_p(int) = \bot_{int}$ if and only if $p = int$. If the top and next-to-top positions of the stack contain valid types (i.e. $\bot_{int}$) then the rule is successfully applied and the transition is performed: $\langle i,\, M_{int},\, \bot_{int}\bot_{int}s\rangle \longrightarrow_{int} \langle i{+}1,\, M_{int},\, \bot_{int}s\rangle$. For the `iadd` instruction, if the pass is different from *int*, the $\alpha_p$ function returns $\top_{int}$, thus the constraints are always satisfied and therefore the transition is performed. Also notice that when pass $p$ is different from *int*, the `iadd` instruction always places $\top_p$ on the stack, since $\alpha_p(int) = \top_p$ if $p \neq int$. Now let us suppose that there is a type error in the bytecode. The `iadd` instruction, for example, may have found a type that is different from *int* on the stack top: the before state would have been $\langle i,\, M_{int},\, \top_{int}\bot_{int}s\rangle$. The multipass rule finds

$$\textbf{goto}_p \qquad \frac{B[i] = \texttt{goto}\ L}{\langle i,\ M,\ s\rangle \longrightarrow_p \langle L,\ M,\ s\rangle}$$

$$\textbf{new}_p \qquad \frac{B[i] = \texttt{new}\ C}{\langle i,\ M,\ s\rangle \longrightarrow_p \langle i+1,\ M,\ \alpha_p(C)s\rangle}$$

$$\textbf{newarray}_p \qquad \frac{B[i] = \texttt{newarray}\ \tau,\quad v \sqsubseteq_p \alpha_p(int)}{\langle i,\ M,\ vs\rangle \longrightarrow_p \langle i+1,\ M,\ \alpha_p(\lceil\tau)s\rangle}$$

$$\textbf{getfield}_p \qquad \frac{B[i] = \texttt{getfield}\ C.f{:}\tau,\quad v \sqsubseteq_p \alpha_p(C)}{\langle i,\ M,\ vs\rangle \longrightarrow_p \langle i+1,\ M,\ \alpha_p(\tau)s\rangle}$$

$$\textbf{putfield}_p \qquad \frac{B[i] = \texttt{putfield}\ C.f{:}\tau,\quad v_1 \sqsubseteq_p \alpha_p(\tau),\quad v_2 \sqsubseteq_p \alpha_p(C)}{\langle i,\ M,\ v_1v_2s\rangle \longrightarrow_p \langle i+1,\ M,\ s\rangle}$$

$$\textbf{invoke}_p \qquad \frac{B[i] = \texttt{invoke}\ C.m(\tau_1,\ldots,\tau_n){:}\tau_r,\quad v_j \sqsubseteq_p \alpha_p(\tau_j)\,(1 \le j \le n),\quad v \sqsubseteq_p \alpha_p(C)}{\langle i,\ M,\ v_1\cdots v_nvs\rangle \longrightarrow_p \langle i+1,\ M,\ \alpha_p(\tau_r)\rangle}$$

$$\textbf{return}_p \qquad \frac{B[i] = \tau\texttt{return},\ v \sqsubseteq_p \alpha_p(\tau),\ v \sqsubseteq_p \alpha_p(\tau_r)}{\langle i,\ M,\ v\rangle \longrightarrow_p \langle -1,\ M,\ \lambda\rangle}$$

$$\textbf{jsr}_p \qquad \frac{B[i] = \texttt{jsr}\ L}{\langle i,\ M,\ s\rangle \longrightarrow_p \langle L,\ M,\ \alpha_p(M(x))s\rangle}$$

$$\textbf{ret}_p \qquad \frac{B[i] = \texttt{ret}\ x,\ M(x) \sqsubseteq \alpha_p(\mathit{ReturnAddress})}{\langle i,\ M,\ s\rangle \longrightarrow_p \langle r_a,\ M,\ s\rangle}$$

Fig. 7. The rules of the multipass verifier (continued).

the type error in pass $p = int$, since constraint $v_1 \sqsubseteq_p \alpha_p(int)$ is not satisfied.

Let us take in consideration a $\tau$`load` $x$ instruction: `aload`. In this case $\tau = \mathit{ReferenceType}$. As a consequence, the constraint in the rule is $M(x) \sqsubseteq_p \alpha_p(\mathit{ReferenceType})$. The constraint may fail only during the *ReferenceType* pass, since $\alpha_p(\mathit{ReferenceType}) = \bot_p$ if and only if $\mathit{ReferenceType} \sqsubseteq p$. If the constraint fails then the abstraction of the type found in register $x$ was not compatible with a reference type.

Notice that the `aload` and the `astore` instructions have asymmetric behaviors since the `aload` does not work on the *ReturnAddress* type (different prefixes have been used in the rules for these two instructions). The $\beta op$, $\tau$`const`, `if`*cond*, `newarray`, $\tau$`load` and $\bar{\tau}$`store` instructions are always checked during one pass. The other instructions may require operands of different types, thus their constraints are possibly checked in more than one pass.

### *4.2.3 The* `aaload` *instruction*

The `aaload` instruction needs some additional explanation. The instruction loads the elements contained in array of references on the stack: it takes an

index and an array reference from the stack and pushes a reference to the array element on the stack. For instance, if during the verification process the type of the array reference is $[A$, then the type placed on the stack by the `aaload` is $A$.

During the standard verification, the return type is inferred by examining the type of the array reference on the stack; on the other hand, during the multipass verification the type is inferred by the pass type. Since the type returned by the `aaload` is different from the one of the operands, the multipass Verifier checks the type returned in a different pass. In particular, the `aaload` type returned can be checked only during the *ReferenceType* pass since the type is computed by inspecting only the instruction prefix: the `aaload` instruction embeds no detailed class information of the array of references it is going to work with.

The rule of the standard Verifier places a value of type $C$ on the stack, the multipass interpreter places $\alpha_p(\mathit{ReferenceType})$. The standard interpreter can compute class $C$ since the array reference on the stack is of type $[C$. The correct type in the multipass should have been $\alpha_p(C)$, but it cannot be computed with the types stored in the stack of the multipass.

Some type-correct bytecodes may not be accepted because of the loss of precision, nevertheless, we can notice that arrays of references are actually never used in Java Card applets. The loss of precision can be solved, for example, by specializing the analysis for each array of the reference type, and by using a two bits type encoding in order to analyze the array references and the array elements in the same pass.

### *4.3 The correctness*

Now we are going to prove that bytecodes rejected by the standard Verifier are also rejected by the multipass Verifier. The following definitions and lemmas are used.

**Definition 4.2** [safety] Given $p \in \mathcal{D}$, we define a binary relation $safe_p \in \mathcal{Q} \times \mathcal{Q}_p$ as:

$$Q \; safe_p \; Q_p \qquad \text{iff} \qquad \alpha_p(Q) \sqsubseteq_p Q_p$$

which means that $Q$ is safely approximated by $Q_p$ if and only if $\alpha_p(Q) \sqsubseteq_p Q_p$ (i.e. $\alpha_p(M) \sqsubseteq_p M_p$ and $\alpha_p(s) \sqsubseteq_p s_p$). We naturally extend the $safe_p$ relation to transition systems.

The following Lemma states that $\forall p$, $\alpha_p$ is an homomorphism:

**Lemma 4.3 (homomorphism)** *Given a state $Q = \langle i, M, s\rangle$, if $Q \longrightarrow Q'$ then $\forall p \in \mathcal{D}$, $\exists Q'_p$ such that $\alpha_p(Q) \longrightarrow_p Q'_p$ and $\alpha_p(Q') \sqsubseteq_p Q'_p$*

**Proof (Sketch)** By cases on the rules of Figure 3 □

Notice that the abstraction loses precision when $\alpha_p(Q') \sqsubset_p Q'_p$ [12].

**Lemma 4.4 (monotonicity)** *Given* $p \in \mathcal{D}$ *and two states* $Q_p^1 = \langle i, M_p^1, s_p^1 \rangle$ *and* $Q_p^2 = \langle i, M_p^1, s_p^2 \rangle$ *such that* $Q_p^1 \sqsubseteq Q_p^2$, *if* $(Q_p^1 \longrightarrow_p Q_p^{1'}$ *and* $Q_p^2 \longrightarrow_p Q_p^{2'})$ *then* $Q_p^{1'} \sqsubseteq Q_p^{2'}$.

**Proof (Sketch)** By cases on the rules of Figure 6 □

The homomorphism and the monotonicity properties guarantee that the standard transition system is safely approximated by the multipass transition system obtained with the $\alpha_p$ function [12].

**Lemma 4.5** $\forall p \in \mathcal{D}$, $(Q, \longrightarrow, \langle i, M^0, \lambda \rangle)$ *safe*$_p$ $(Q_p, \longrightarrow_p, \langle i, \alpha_p(M^0), \lambda \rangle)$.

**Proof.** By Lemma 4.3 and Lemma 4.4 □

The following Lemma defines a property of the $\alpha_p$ function.

**Lemma 4.6** *Given* $v_1, v_2 \in \mathcal{D}$, *if* $v_1 \not\sqsubseteq v_2 \Rightarrow \exists\, p : \alpha_p(v_1) \not\sqsubseteq_p \alpha_p(v_2)$.

**Proof.** Choose $p = v_2$. It is $\alpha_{v_2}(v_1) \not\sqsubseteq \alpha_{v_2}(v_2)$ by definition of $\alpha_p$ since, if $v_1 \not\sqsubseteq v_2$, then $\alpha_{v_2}(v_1) = \top_{v_2} \not\sqsubseteq_{v_2} \bot_{v_2} = \alpha_{v_2}(v_2)$ □

The following theorem states that if the standard Verifier gives an error, then $\exists p$ such that $\longrightarrow_p$ gets stuck.

**Theorem 4.7** $\langle 0, M^0, \lambda \rangle \overset{*}{\not\longrightarrow} \Rightarrow \exists p : \langle 0, \alpha_p(M^0), \lambda \rangle \overset{*}{\not\longrightarrow}_p$.

**Proof (Sketch)** Given that $\langle 0, M^0, \lambda \rangle \overset{*}{\not\longrightarrow}$, there exists a final state $Q = \langle i, M, s \rangle$, with $i \neq -1$; from Lemma 4.5, we know that, for each $p$, $Q_p = \langle i, M_p, s_p \rangle$ exists so that $Q$ *safe*$_p Q_p$. Then, for each possible final state $Q$, we have to show that there exists $p$ such that $Q_p$ is also final. The proof proceeds by cases: $B[i]$ must be one of the instructions enumerated in the rules of Figure 3. Only one rule (two if the instruction is an `if`) could be applied to make a transition from $Q$. Since $Q$ is final, the corresponding rule cannot be applied: this means that either the preconditions are not met, or the form of the before state in the transition does not match with $Q$. The following reasoning must be repeated for each rule. It is easy to show that, if $Q$ does not match the before state of the transition, the form of $Q_p$ does not match in the corresponding multipass rule, so $Q_p$ is final (for every $p$). If a precondition is not met, we note that all the preconditions, in Figure 3, are of form $v \not\sqsubseteq v'$, with $v$ taken from the before state (in a register or on the stack) and $v'$ fixed by the instruction. The corresponding constraint in Figure 6 becomes $\bar{v} \not\sqsubseteq_p \alpha_p(\bar{v}')$, where $\bar{v}$ is taken from the corresponding position in

the (abstract) before state. From the definition of the $safe_p$ relation, we know that, for each $p$, $\alpha_p(v) \not\sqsubseteq_p \bar{v}$ and, by Lemma 4.6, we know that $\exists \bar{p}$ such that $v \not\sqsubseteq v' \Rightarrow \alpha_{\bar{p}}(v) \not\sqsubseteq_{\bar{p}} \alpha_{\bar{p}}(v')$. Thus, we must have $\bar{v} \not\sqsubseteq_{\bar{p}} \alpha_{\bar{p}}(v')$. This implies that state $Q_{\bar{p}}$ does not meet a precondition in the (only) rule that could be used to make a transition, so it is a final state. □

### *4.4 Checking object initialization*

The creation of a new object is a single statement in the Java programming language: the statement provides object allocation and initialization. However, in the bytecode the object initialization must be checked since the objects are created during two distinct phases. The first phase is allocation of the space in the heap, the second is object initialization. In particular, the `new` instruction allocates the space and the call to the appropriate constructor `<init>` performs the object initialization. The Verifier checks that the objects are not initialized twice and that they are not used before they have been initialized [9].

Notice that references to multiple not-yet-initialized objects may be present in the stack locations and in the registers: when the constructor in called, it must know which reference points to which object in order to initialize them correctly. The standard verification algorithm uses a *special* type to keep trace of the uninitialized objects [9]. The special type contains the bytecode position of the `new` instruction that creates the object instance.

The multipass analysis requires an additional pass for each class type: uninitialized objects of a given class are traced within a pass. A data structure that holds information about the instance of uninitialized objects is also needed. It should be noted that uninitialized objects must not be present in the stack locations and in the registers when a backwards branch is taken [9]. This last constraint simplifies structure of the data needed during the multipass: its size is constant and the object initialization can be resolved with the FIFO strategy.

## 5 Experimental Results

A prototype tool has been developed by using the open-source BCEL/JustIce package [1]. BCEL (ByteCode Engineering Library) is a set of APIs that provides an object-oriented view of the binary class files. JustIce is a bytecode Verifier. The prototype is a modified version of JustIce: the main modifications have been made to specialize the data-flow engine for the multipass process. It is available at http://www.ing.unipi.it/~o1833499.

The prototype has been tested with many methods. Hereafter we are

| package | method | targets | dictionary size (standard \| multipass) | |
|---|---|---|---|---|
| com.aelitis.azureus.core.peermanager.utils.PeerIDByteDecoder | PeerIDByteDecoder | 64 | 4.3 KB | 0.18 KB |
| jasper.Jasper | recurseClasses | 54 | 1.3 KB | 0.05 KB |
| jasper.Code_Collection | < init > | 108 | 2.4 KB | 0.10 KB |
| com.gemplus.purse.Purse | appInitDebit | 15 | 0.97 KB | 0.04 KB |
| com.sun.javacard.impl.AppletMgr | createApplet | 18 | 1.21 KB | 0.05 KB |
| com.sun.javacard.jcasm.ParserTokenManager | jjMoveStringLiteral | 82 | 7.92 KB | 0.33 KB |
| org.jgraph.graph.DefaultEdge$DefaultRouting | route | 14 | 0.99 KB | 0.04 KB |

Fig. 8. Dictionary size of some methods.

presenting the statistics relevant to five applications: 1) Azureus, an open-source peer-to-peer application: it contains a large number of network and identification methods; 2) JGraphT, an open-source mathematical library; 3) Jasper, a class file disassembler; 4) the Java Card Runtime Environment; 5) the Pacap prototype, an Electronic Purse application.

The statistics include the number of targets and size of the dictionary for the standard and multipass verification. As expected, the size of the multipass verification dictionary is more than ten times smaller than the size of the standard verification dictionary. All the space gained is due to the encoding of the types: 1 bit for the multipass, 3 bytes for the standard.

Figure 8 reports the size of the dictionary during the verification process of methods belonging to the examined packages (the space overhead for the dictionary indexing is not taken into account). The dictionary size is computed as $T \times (H + N) \times E$, where $T$ is the number of targets, $H$ and $N$ are the maximum stack height and the maximum register number, $E$ is the number of bytes needed to encode the types. The standard verification of some methods requires more than 2KB of RAM and, for complex methods, 4KB of RAM are not sufficient. On the other hand, the multipass verification comfortably fits in 1KB of RAM for all the examined methods. Moreover, it should be noted that the dictionary usually contains many duplicated states when the number of targets is very high: the one bit encoding of the types reduces the number of possible states thus, in some cases, the dictionary size can be optimized by avoiding state duplication [2].

Some considerations on the time needed to perform a complete multipass verification can be made. We analyzed the number of passes needed for each

method in order to perform a complete multipass verification: it depends on the instructions contained in the method and, in many cases, it is less than what we expected simply by analyzing the constant pool. For the methods we have tested, the total number of types (types in the constant pool and basic types) was 26.7 in average, while the number of types actually used in the methods was only 5.8 (22%), in average. We should also consider that each pass of the multipass Verifier is much simpler than the standard one: in particular, the multipass verification always compares bits, while the standard one usually needs to traverse the class hierarchy in order to compute the results.

## 6 Further work and conclusions

In this paper we presented an approach for bytecode verification that optimizes the use of the system memory and we have proved that it is correct in relation to the standard data-flow analysis. The approach reduces the space for the type encoding by executing multiple passes and by verifying a single type at each pass. It should be noted that, by increasing the number of bits used to encode the types, the multipass analysis can be performed on more than one type during each step. In particular, the multipass analysis can be fine-tuned on the card characteristics. The multipass approach is general and potentially applicable to different optimizations and application areas. For example it can be used to improve the time performances of data-flow analysis on multi-processor systems. In a multipass strategy, each processor could analyze the whole code for a different abstraction and the analysis could be fully parallelized.

## References

[1] Apache Foundation. ByteCode Engineering Library (BCEL) user manual, http://jakarta.apache.org/bcel/index.html, 2002.

[2] C. Bernardeschi, L. Martini, and P. Masci. Java bytecode verification with dynamic structures. In *8th IASTED International Conference on Software Engineering and Applications (SEA)*, Cambridge, MA, USA, 2004.

[3] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.

[5] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

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

[7] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal on automated reasoning*, 30, 2003.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[9] T. Lindholm and F. Yellin. *Java Virtual Machine Specification, 2nd edition*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] G. C. Necula. Proof-carrying code. In *24th Annual Symposium on Principles of Programming Languages Proceedings*, pages 106–119, January 1997.

[11] E. Rose and K. Rose. Lightweight bytecode verification. In *WFUJ 98 Proceedings*, 1998.

[12] D. A. Schmidt. Abstract interpretation in the operational semantics hierarchy. In *Basic Research in computer Science*, 1997.