Memory consumption analysis of Java smart cards

Pablo Giambiagi
Swedish Institute of Computer Science, Box 1263, SE-164 28, Kista, Sweden.
E-mail: {pablo at sics.se}

and

Gerardo Schneider*
Dept. of Informatics, University of Oslo - PO Box 1080 Blindern, N-0316 Oslo, Norway
E-mail: {gerardo at ifi.uio.no}

Abstract

Memory is a scarce resource in Java smart cards. Developers and card suppliers alike would want to make sure, at compile- or load-time, that a Java Card applet will not overflow memory when performing dynamic class instantiations. Although there are good solutions to the general problem, the challenge is still out to produce a static analyser that is certified and could execute on-card. We provide a constraint-based algorithm which determines potential loops and (mutually) recursive methods. The algorithm operates on the bytecode of an applet and is written as a set of rules associating one or more constraints to each bytecode instruction. The rules are designed so that a certified analyser could be extracted from their proof of correctness. By keeping a clear separation between the rules dealing with the inter- and intra-procedural aspects of the analysis we are able to reduce the space-complexity of a previous algorithm.

1 Introduction

Embedded systems, special-purpose computer systems built into larger devices, can be found almost everywhere: from a simple coffee machine, to a mobile phone and a car, all may contain at least one embedded system, if not many. Programmable smart cards are small personal devices provided with a microprocessor capable of manipulating confidential data, allowing the owner of the card to have secure access to chosen applications. Applications, called applets, can be downloaded and executed in these small communicating devices, raising major security issues. Indeed, without appropriate security measures, a malicious or simply buggy applet could be installed in smart cards and compromise sensitive data, perform unauthorised transactions or even render the card useless by consuming all of its resources.

The multi-application model, i.e. the ability to load applets from possibly competing providers, puts strong demands on the reliability of the software used to manipulate the data entrusted to the card. Hence, it is essential that the platform, as well as the applets running on it, satisfy a minimum of safety and security guarantees in order to preserve confidentiality, integrity and availability of information. To guarantee availability of the services offered by small devices the management and control of resources (e.g. memory) is crucial.

These days a top-notch smart card has about 64KB EEPROM, 200KB ROM, 4KB RAM and a 32-bits data bus. The corresponding numbers for banking and other low-end cards are 16KB, 16KB, 1KB and 8 bits. In the smart card world, memory is a very precious resource which must be manipulated carefully. Consequently the smart card industry has developed specific programming guidelines and procedures to keep memory usage under control. We quote Chen: "Because memory in a smart card is very scarce, neither persistent nor transient objects should be created willy-nilly" [7, Section 4.4]. The advice is extended to method invocations: "You should also limit nested method invocations, which could easily lead to stack overflow. In

*Partially supported by the *RNTL* French project, CASTLES (*Conception d'Analyses Statiques et de Tests pour le Logiciel Embarqué Sécurisé*). Part of this work was done while the author was spending one year as a researcher at IRISA-INRIA.

particular, applets should not use recursive calls" [7, Section 13.3]; and object allocation: "An applet should always check that an object is created only once" [7, Section 13.4]. Even though these recommendations are generally respected by the industry, mistakes regarding memory usage –like the instantiation of a class inside a loop– either accidental or intentional, may have dire consequences. In fact, nothing prevents a(n) (intentionally) badly written applet to allocate all persistent memory on a card. Hence, the ability to detect recursive methods and loops is imperative, both during the development process and at applet load-time.

As far as we know there is no *on-card* tool available for detecting the dynamic instantiation of classes inside cycles and/or recursive functions for Java smart cards. Leroy [10, 11] describes a bytecode verifier that could be made to run on-card, but it does not address properties related to memory usage. Previous work presents a certified analyser to determine loops and mutually recursive methods but its memory footprint prevents it from being deployed on-card [6].

This paper takes up the challenge of constructing a formally certified static analyser whose memory requirements are low enough that it could be added, eventually, to an on-card bytecode verifier. This implies making the right trade-off between the precision of the analysis and the practical viability of the formal certification process. Regarding the latter requirement we have adopted the approach introduced in [6]. According to this approach, the analyser should be first described as a constraint-based algorithm, and then formally extracted from the proof of correctness of the algorithm in the proof assistant Coq [4]. Whereas [6] demonstrates the feasibility of the extraction process, the difficulty remains on how to describe more efficient algorithms (in terms of memory consumption) without compromising the ability to perform code extraction.

The algorithms presented here improve those presented in [6], in terms of the auxiliary memory used and because they also cover subroutines and exceptions, which were not addressed originally. Although the memory consumption is not fully optimised, we have taken care to partition the tasks to reduce the amount of data that needs to be kept simultaneously in memory.

One further feature of our algorithm is that it works directly on the bytecode and there is no need to precede its execution with the construction of extra data structures (e.g., a control-flow graph). Our approach includes both intra- and inter-procedural analyses. Both work on arbitrary bytecode, i.e. it is not assumed that the bytecode is produced by a "well-behaved" compiler. For the intra-procedural analysis, the construction of the graph of basic blocks –a prerequisite of many textbook algorithms– is done implicitly without resorting to any auxiliary, pre-existent analysis.

The language being considered is the *bytecode* language JCVML (Java Card Virtual Machine Language), although the techniques discussed in this paper are independent of this choice and can be applied to most bytecode languages. JCVML manipulates (dynamic) objects and arrays and besides the usual stack and register operations it accommodates interesting features like virtual method calls, (mutually) recursive methods, (un)conditional jumps, exceptions and subroutines. We assume there is no garbage collector, which is the case for Java Card up to version 2.1[1].

The paper is organised as follows. Section 2 briefly presents the language being considered while in Section 3 we present the specification of the algorithm. We prove, in Section 4, termination of our algorithm as well as its soundness and completeness with respect to an abstraction of the operational semantics of the language. In Section 5 we show how we could treat subroutine calls and exceptions. In the last section we discuss the complexity of our algorithm, related and future work.

2 The language

We consider in this paper the whole JCVML language. The instruction set, as well as an operational semantics of a language that models JCVML is given in [16]. It comprises, among others, the following instruction sets:

- Stack manipulation: `push`, `pop`, `dup`, `dup2`, `swap`, `numop`;
- Local variables manipulation: `load`, `store`;
- Jump instructions: `if`, `goto`;
- Heap manipulation: `new`, `putfield`, `getfield`;

[1]Starting with Java Card 2.2 the machine includes a garbage collector which may be activated invoking an API function at the end of the execution of the applet.

- Array instructions: `arraystore`, `arrayload`;
- Method calls and return: `invokevirtual`, `invokestatic`, `invokespecial`, `invokeinterface`, `return`.

In Section 5 we sketch how to extend the analysis to cover subroutine calls and exception handling.

A JCVML program P (which will be called in what follows a *bytecode program* or simply, a *program*) is a set of methods $m \in \mathit{Method}_P$ consisting of a numbered sequence of instructions. We write $\mathit{InstrAt}(m, pc) = \mathtt{i}$ to denote that the instruction at program line pc (usually called a *pc-number*) in method m is `i`. Let $\mathit{ProgCounter}_P$ be the set of all the pc-numbers. We will usually omit the subscript P, being understood that the analysis depends on a given program P.

3 Specification of the analysis

We present in this section a constraint-based specification of an analyser for detecting the occurrence of a `new` instruction inside potential cycles, due to intra-method loops and/or (mutually) recursive method calls. It consists in the computation of three functions *Loop*, *Loop*' and *Rec* respectively providing information about intra-method cycles, methods called from intra-method cycles and (mutually) recursive methods (as well as the methods reachable from these). Using the above functions, the main algorithm checks whether a `new` instruction occurs inside a potential cycle. The specification of the main algorithm and all the above-mentioned functions are presented as a set of rules, each rule associating one or more constraints to each instruction in the program. The solution of the set of constraints (which are of the form $\{F_1(X) \sqsubseteq X, \ldots F_n(X) \sqsubseteq X\}$) is the least fix-point of a function $\mathcal{F}$ obtained from $F_1, \ldots F_n$. By a corollary to Tarski's Fix-point Theorem, this element may be obtained as the limit of the stabilising sequence $(\mathcal{F}^n(\bot))_{n\in\mathbb{N}}$.

We show next how to detect intra-method loops and (mutually) recursive methods.

3.1 Detection of loops

We will define two functions, *Loop* and *Loop*', for detecting cycles in a given program P: *Loop* will be defined intra-procedurally while *Loop*' will propagate inter-procedurally the results given by *Loop*.

Intra-procedural analysis. The analysis works by identifying the basic blocks in each method, and by associating to each program point the set of basic blocks that may be traversed on an execution path ending in the point in question. With this information at hand, the analysis is able to mark the instructions involved in a potential cycle. The concept of *basic block* is well-established in program analysis [14]. It refers to a contiguous, single-entry code segment whose (conditional) branching instructions may only appear at the end of the block. This implies that, with the exception of the very first block (which starts at $pc = 1$), all other basic blocks start at the destination of some jump. Our analysis identifies each basic block by the pc-number of its first instruction, which is found observing the targets of branching instructions.

We proceed to formalise this intuition. Given a program P, let *Method* and *ProgCounter* be respectively the sets of method names and pc-numbers of P. Given a method m, its pc-numbers range from 1 to the constant END_m, which is equal to the number of lines of method m plus one. The analysis defines the function

$$\mathit{Loop}\colon \mathit{Method} \times \mathit{ProgCounter} \to \wp(\mathit{ProgCounter} \uplus \{\bullet\})\,,$$

where $\mathit{Loop}(m, pc) \cap \mathit{ProgCounter}$ is the set of basic blocks (identified by their first location) that may be traversed on an execution path to (m, pc). Moreover, if $\bullet \in \mathit{Loop}(m, pc)$, then we know that the location (m, pc) lies in a cycle of the control flow graph of P.

We do not assume any structure on the bytecode program being considered, except that each `goto` is intra-method – a property guaranteed by the bytecode verifier [10]. Table 1 shows the rules (one for each instruction) used for computing *Loop*. That is, *Loop* is defined as the least element of the lattice

$$\mathcal{L} = (\mathit{Method} \times \mathit{ProgCounter} \to \wp(\mathit{ProgCounter} \uplus \{\bullet\}),\ \sqsubseteq),$$

that satisfies all the constraints derived from the code using the rules in Table 1. The order relation of the lattice is defined as $f \sqsubseteq f'$ iff $f(m, pc) \sqsubseteq f'(m, pc)$, for all (m, pc). Notice that the co-domain of *Loop* is a powerset. This is a lattice under subset inclusion; its least element, $\bot$, is the empty set.

$$\frac{}{\{1\} \sqsubseteq Loop(m,1)} \quad (1)$$

$$\frac{(m,pc) : \texttt{goto}\ pc'}{F(Loop(m,pc), pc') \sqsubseteq Loop(m,pc')} \quad (2)$$

$$\frac{(m,pc) : \texttt{if}\ t\ op\ \texttt{goto}\ pc'}{\begin{array}{l} F(Loop(m,pc), pc') \sqsubseteq Loop(m,pc') \\ F(Loop(m,pc), pc+1) \sqsubseteq Loop(m,pc+1) \end{array}} \quad (3)$$

$$\frac{(m,pc) : \texttt{invokevirtual}\ m'}{Loop(m,pc) \sqsubseteq Loop(m,pc+1)} \quad (4)$$

$$\frac{(m,pc) : \texttt{return}}{\bot \sqsubseteq Loop(m, \mathrm{END}_m)} \quad (5)$$

$$\frac{(m,pc) : \texttt{instr}}{Loop(m,pc) \sqsubseteq Loop(m,pc+1)} \quad (6)$$

Table 1: Rules for *Loop*

Rule (1) labels the first line of each method as a basic block. Although this rule is not needed for detecting loops, it helps to identify which loops are actually reachable (see Example 1). Rule (2) serves two simultaneous purposes: It records location (m, pc') as the start of a basic block, and takes care of detecting cycles. If the list of blocks possibly visited up to (m, pc) contains the destination of the `goto` instruction, i.e. $pc' \in Loop(m, pc)$, the rule marks (m, pc') as belonging to a cycle. These two tasks are achieved with the help of the following auxiliary function:

$$F(L_{m,pc}, pc') = \begin{cases} L_{m,pc} \cup \{\bullet\} & \text{if } pc' \in L_{m,pc} \\ L_{m,pc} \setminus \{\bullet\} \cup \{pc'\} & \text{otherwise} \end{cases} \quad (7)$$

where "\" is the usual set subtraction operator.

A conditional branch instruction determines the starting point of two basic blocks: (1) the destination of the jump and (2) the location of the instruction immediately after the conditional jump. This is taken care of by rule (3) again using function F.

Rule (4) concerns virtual method invocations. As we are considering here only intra-procedural cycles, the content of *Loop* is not propagated to method m'. Similar rules for `invokestatic`, `invokespecial` and `invokeinterface` may be considered. The `return` instruction –rule (5)– does not propagate anything, as expected for an intra-procedural analysis. In rule (6), `instr` stands for any other instruction not defined by rules (1)–(5); in this case the information is simply propagated to the next instruction.

For the sake of simplicity define a predicate $Loop_{m,pc} \equiv (\{1, \bullet\} \subseteq Loop(m, pc))$, so that $InstrAt(m, pc)$ is in a reachable loop iff $Loop_{m,pc}$.

Example 1 *In Fig. 1 we show part of three bytecode methods; the value of Loop appears to the right of each program. We assume that the lines marked with ellipsis have no branching instructions. Program (a) contains no cycles, which is reflected by the absence of* $\bullet$ *in the right column. Program (b) has a cycle involving lines 20 through 70. Notice how the analysis discovers the basic blocks (with start in lines 1, 20, 31, 41, 50, 51 and 90). Observe as well that* $\bullet$ *is not propagated to line 90 by the branching instructions at lines 40 and 50. The annotation is filtered by F (in rule 3) to reflect the fact that line 90 lies outside the cycle. Finally, Program (c) illustrates the case of an unreachable cycle. Lines 2-3 are marked as belonging to a cycle involving the basic block starting in line 2, but there is no path to this block from line 1.* □

Inter-procedural analysis. Given a program P, we define the following lattice:

$$\mathcal{L} = (Method \times ProgCounter \to \{\bot, \top\},\ \sqsubseteq),$$

where *Method* and *ProgCounter* are as before and $\{\bot, \top\}$ is the usual lattice with $\bot \sqsubseteq \top$. The order for the function lattice is defined as follows: $f \sqsubseteq f'$ if and only if $f(m, pc) \sqsubseteq f'(m, pc)$, for all (m, pc). For convenience and to be consistent with the notation used in the computation of *Loop*, we will write $\bullet$ for $\top$.

$Loop'$ captures all the instructions reachable from intra-procedural cycles through method calls. It is defined by the rules shown in Table 2. To keep the presentation simple the first constraint for rule (8) has been written $\bullet \sqsubseteq Loop'(m', 1)$, but it must be understood as: $\forall m_{ID} \in implements(m')$, $\bullet \sqsubseteq Loop'(m_{ID}, 1)$, and similarly for rule (9). The same remark holds for the `invokevirtual` rules of the function *Rec* defined later. Function *implements* is a static over-approximation of the dynamic `methodLookup` function [16], which

```
 1 ...          {1}
30 if goto 50   {1}
31 goto 49      {1,31}
   ...
40 goto 60      {1,31,40,49,50}
   ...
49 if goto 60   {1,31,49}
50 goto 40      {1,31,49,50}
   ...
60 ...          {1,31,40,49,50,60}
```

(a)

```
 1 ...          {1}
20 ...          {1,20,31,41,50,51,•}
30 if goto 50   {1,20,31,41,50,51,•}
31 ...          {1,20,31,41,50,51,•}
   ...
40 if goto 90   {1,20,31,41,50,51,•}
41 ...          {1,20,31,41,50,51,•}
   ...
50 if goto 90   {1,20,31,41,50,51,•}
51 ...          {1,20,31,41,50,51,•}
   ...
70 goto 20      {1,20,31,41,50,51,•}
   ...          {}
90 ...          {1,20,31,41,50,51,90}
```

(b)

```
1 goto 4      {1}
2 ...         {2,•}
3 goto 2      {2,•}
4 return      {1,4}
```

(c)

Figure 1: Example

$$\frac{(m,pc) : \texttt{invokevirtual}\, m' \quad Loop_{m,pc}}{\begin{array}{c}\bullet \sqsubseteq Loop'(m',1)\\ Loop'(m,pc) \sqsubseteq Loop'(m,pc+1)\end{array}} \quad (8)$$

$$\frac{(m,pc) : \texttt{invokevirtual}\, m' \quad \neg Loop_{m,pc}}{\begin{array}{c}Loop'(m,pc) \sqsubseteq Loop'(m',1)\\ Loop'(m,pc) \sqsubseteq Loop'(m,pc+1)\end{array}} \quad (9)$$

$$\frac{(m,pc) : \texttt{instr}}{Loop'(m,pc) \sqsubseteq Loop'(m,pc+1)} \quad (10)$$

$$\frac{(m,pc) : \texttt{return}}{\bot \sqsubseteq Loop'(m,\mathrm{END}_m)} \quad (11)$$

Table 2: Rules for $Loop'$

returns all possible implementations of a given method with name m' relative to a program P. Notice that on Java cards the information needed to compute such function is available at load-time. We do not specify it in further detail.

3.2 Detection of (mutually) recursive methods

Given a program P, we define a lattice $\mathcal{L}$ as follows:

$$\mathcal{L} = (Method \times ProgCounter \times \to \wp(Method \uplus \{\bullet\}),\ \sqsubseteq),$$

where the order relation is inherited from the powerset lattice in the usual way, i.e. $f \sqsubseteq f'$ iff $f(m,pc) \sqsubseteq f'(m,pc)$, for all (m,pc). *Rec* takes values over the above lattice. The definition of *Rec* is given by the rules described in Table 3. The first rule corresponds to the case of a recursive method m; it annotates the first instruction of the method with $\bullet$ and the method name. The application of the other rules will then propagate these annotations to all the instructions in the method. Rule (13) shows the case of a non self-invocation. The content of *Rec* is propagated unchanged to the next instruction inside the method, and to $InstrAt(m',1)$ as determined by function $G : \wp(Method \uplus \{\bullet\}) \times Method \longrightarrow \wp(Method \uplus \{\bullet\})$:

$$G(R_{m,pc}, m') = \begin{cases} R_{m,pc} \cup \{m, \bullet\} & \text{if } m' \in R_{m,pc} \\ R_{m,pc} \cup \{m\} & \text{if } m' \notin R_{m,pc} \end{cases}$$

At each method call $(m,pc) : \texttt{invokevirtual}\, m'$, G adds to $Rec(m,pc)$ the calling method name; if the called method name is already in $Rec(m,pc)$, then also $\bullet$ is added. Intuitively, G detects whether a given method has been visited more than once following the same "path". The other rules only propagate the result defined by the rules corresponding to `invokevirtual` (as before, `instr` stands for any instruction not already covered by rules (12)–(14)).

For a given method m and program counter pc, we define the predicate $Rec_{m,pc} \equiv (\bullet \in Rec(m,pc))$.

Remark. Notice that we are interested in knowing which methods may be executed an unknown number of times due to recursion. Thus, *Rec* detects not only all the (mutually) recursive methods but also all the methods which are called from those: for any method m such that $Rec_{m,1}$, if $m \in Rec(m,1)$, m is (mutually) recursive, otherwise m is reachable from a (mutually) recursive method. From now on we will say that a method is recursive if it calls itself or if it belongs to a set of mutually recursive methods.

$$\frac{(m,pc) : \texttt{invokevirtual}\, m' \quad m = m'}{\begin{array}{c} Rec(m,pc) \cup \{m, \bullet\} \sqsubseteq Rec(m',1) \\ Rec(m,pc) \sqsubseteq Rec(m,pc+1) \end{array}} \quad (12) \qquad \frac{(m,pc) : \texttt{return}}{Rec(m,pc) \sqsubseteq Rec(m, \mathrm{END}_m)} \quad (14)$$

$$\frac{(m,pc) : \texttt{invokevirtual}\, m' \quad m \neq m'}{\begin{array}{c} G(Rec(m,pc), m') \sqsubseteq Rec(m',1) \\ Rec(m,pc) \sqsubseteq Rec(m,pc+1) \end{array}} \quad (13) \qquad \frac{(m,pc) : \texttt{instr}}{Rec(m,pc) \sqsubseteq Rec(m,pc+1)} \quad (15)$$

Table 3: Rules for *Rec*

3.3 Main algorithm

In this section we present the specification of the main algorithm, which uses the three analyses described so far.

The only instructions sensitive to our analysis are the ones that enlarge the heap, namely instructions that create array objects and class instances (see the operational semantics in [16]). For simplicity we consider here only one instruction `new`, but we mean both $\texttt{new}(cl)$ and $\texttt{new}(\mathsf{array}\, a)$, where a is an array type. We write $Cycle_{m,pc}$ as a shortcut for $Loop_{m,pc} \vee Loop'_{m,pc} \vee Rec_{m,pc}$. The specification of the algorithm is given by the following three-valued function $\Gamma : Method \times ProgCounter \rightarrow \{0, 1, \infty\}$:

$$\Gamma(m,pc) = \begin{cases} \infty & \text{if } (m,pc) : \texttt{new}(cl) \ \wedge \ Cycle_{m,pc} \\ 1 & \text{if } (m,pc) : \texttt{new}(cl) \ \wedge \ \neg Cycle_{m,pc} \\ 0 & \text{otherwise} \end{cases}$$

So $\Gamma(m,pc)$ represents a bound on the number of object instances that may be created by the program instruction at (m,pc). If there is no `new` instruction there, then $\Gamma(m,pc)$ is clearly 0. When $(m,pc) : \texttt{new}(cl)$ and (m,pc) lies in no (potential) cycle, then we know that the instruction may be executed at most once and therefore $\Gamma(m,pc) = 1$. Finally, $\Gamma(m,pc) = \infty$ if the current instruction is a `new` and lies inside a potential cycle. Notice that the main algorithm is obtained without computing a fix-point, since all the information is already in *Cycle*.

4 Properties of the analysis

4.1 Termination

One of the crucial properties for proving termination of our analyser is the *ascending chain condition*, i.e. that the underlying lattices have no infinite, strictly increasing chains. The property is trivially satisfied by all our lattices as their height are finite. The algorithm reduces to the problem of solving a set of constraints over a lattice $\mathcal{L}$, which can be transformed into the computation of a fix-point of a monotone function over $\mathcal{L}$. Termination follows from the proof of the termination of the fix-point computation for obtaining *Loop*, *Loop*', *Rec* and Γ.

We need the following auxiliary lemma about the functions F and G used in the definition of *Loop* and *Rec*.

Lemma 1 *The functions F and G are monotone.* $\square$

It is well known (see for instance [14]) that the solution of a set of constraints of the form $\{F_1(X) \sqsubseteq X, \ldots F_n(X) \sqsubseteq X\}$, where each F_i is monotone, is the least solution of a function $\mathcal{F}$ obtained from $F_1, \ldots F_n$. Moreover, by a corollary of Tarski's fix-point theorem, this element may be obtained as the limit of the stabilising sequence $(\mathcal{F}^n(\bot))_{n\in\mathbb{N}}$. As a corollary of the previous lemma we have the following result.

Corollary 1 *The computations of the least fix-points corresponding to the set of constraints defining Loop, Loop' and Rec always terminate.* $\square$

Termination of the algorithm follows directly from termination of *Loop*, *Loop*' and *Rec*, given that the algorithm simply scan each method sequentially without iterating.

Theorem 1 *The computation of the function Γ always terminates.* $\square$

4.2 Soundness and completeness

We consider here a *maximal semantics*. That is, the semantics of a program P, noted $[\![P]\!]$, is the set of all its possible executions (traces). Such traces may be obtained symbolically by applying the rules of the operational semantics [16]. We prove soundness and completeness of the functions introduced in Section 3 w.r.t. an appropriate abstraction of the operational semantics. For *Rec* we consider the usual method-call graph, which is an abstraction of the control-flow graph $\mathcal{G}(P)$. For the intra-procedural case (*Loop*) we consider $\mathcal{G}_m(P)$, which is a modified restriction of $\mathcal{G}(P)$ to the given method m. The trace obtained from a traversal of the graph $\mathcal{G}(P)$, which does not take into consideration the tests in branching instructions, is an *abstract trace* of P; let $[\![\widehat{P}]\!]$ denote the set of all the abstract traces of program P. In what follows we will use the fact that $[\![P]\!] \subseteq [\![\widehat{P}]\!]$ (see [13, 15] and references therein).

4.2.1 Loop

Given a program P, its *control-flow graph* $\mathcal{G}(P)$, is a 4-tuple $(\mathcal{S}, \rightarrow, s_0, F)$, where

- $\mathcal{S}$ is a set of *nodes* (or *program states*), ranging over elements of $\mathit{Method} \times \mathit{ProgCounter}$;
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *transitions*, which models the flow of control;
- s_0 is the initial state: $s_0 = (m_0, 1)$, where m_0 is the main method;
- $F = (m_0, \mathrm{END}_{m_0})$ is the final state.

In what follows we will write $\mathcal{G}$ instead of $\mathcal{G}(P)$. More formally, $\rightarrow$ is defined as follows:

$$\frac{(m, pc) : \texttt{goto}\ pc'}{(m, pc) \rightarrow (m, pc')} \qquad \frac{(m, pc) : \texttt{if}\ c\ \texttt{goto}\ pc'}{(m, pc) \rightarrow (m, pc') \qquad (m, pc) \rightarrow (m, pc+1)}$$

$$\frac{(m, pc) : \texttt{return}}{(m, pc) \rightarrow (m, \mathrm{END}_m)} \qquad \frac{(m, pc) : \texttt{invokevirtual}\ m'}{(m, pc) \rightarrow (m', 1) \qquad (m', \mathrm{END}_{m'}) \rightarrow (m, pc+1)}$$

$$\frac{(m, pc) : \texttt{instr}}{(m, pc) \rightarrow (m, pc+1)}$$

where `instr` is any instruction different from `invokevirtual`, `goto`, `if`, and `return`. As usual, $\rightarrow^+$ denotes the transitive closure of $\rightarrow$; we say that a state s' is *reachable* from s iff $s \rightarrow^+ s'$ and write $s' \in \mathit{Reach}(s)$. We also introduce a *satisfaction* relation: $\mathcal{G} \models \phi$ iff $\mathcal{G}$ satisfies the property ϕ.

For our purposes it is convenient to define the control-flow graph of each method independently, which may be obtained from the graph $\mathcal{G}$ by defining a transition relationship $\rightarrow_m$ which agrees with $\rightarrow$ except for the `invokevirtual` instruction:

$$\frac{(m, pc) : \texttt{invokevirtual}\ m'}{(m, pc) \rightarrow_m (m, pc+1)}.$$

For each method m, we denote the graph obtained using the new transition $\rightarrow_m$ by $\mathcal{G}_m$, and call it the *m-control-flow graph*. We say that *pc*' is reachable from *pc* in $\mathcal{G}_m$, written $pc' \in \mathit{Reach}_m(pc)$, if and only if $(m, pc) \rightarrow_m^+ (m, pc')$. We will omit the subindex m and we will simply write *Reach* and $\rightarrow$ instead of Reach_m and $\rightarrow_m$ respectively whenever understood from the context. We define the following predicate:

$$\mathcal{G}_m \models \mathit{SynC}(m, pc) \quad \text{iff} \quad \mathcal{G}_m \models pc \in \mathit{Reach}(pc).$$

Thus, the predicate *SynC* determines whether a given instruction is in a *syntactic cycle*. Notice that this predicate only characterises *intra-method* cycles.

The following theorem establishes that the function *Loop* characterises exactly all the syntactic cycles of a method.

Theorem 2 (Soundness and Completness of *Loop*) *$\mathit{Loop}_{m,pc}$ if and only if $\mathcal{G}_m \models \mathit{SynC}(m, pc)$.* □

4.2.2 *Loop'*

The following predicate, *SynCReach*, determines whether a given instruction is reachable from a method invocation inside a syntactic cycle:

$$\mathcal{G} \models \mathit{SynCReach}(m, pc)$$
$$\text{iff}$$
$$(\exists m')\ \mathcal{G}_{m'} \models \mathit{SynC}(m', pc') \text{ and } \mathcal{G} \models (m, pc) \in \mathit{Reach}(m', pc').$$

Loop' characterises exactly all instructions reachable from a cycle:

Theorem 3 (Soundness and Completeness of *Loop'*) *$\mathit{Loop}'_{m,pc}$ if and only if $\mathcal{G} \models \mathit{SynCReach}(m, pc)$.* □

4.2.3 *Rec*

Given a program P, its *method-call graph* $\mathcal{M}(P)$, is a 3-tuple $(\mathcal{N}, \rightarrow, m_0)$, where

- $\mathcal{N}$ is a set of *nodes* ranging over *Method* names;
- $\rightarrow \subseteq \mathcal{N} \times \mathcal{N}$ is a set of *transitions*;
- m_0 is the initial state.

Notice that the transitions of this graph are not labelled and that there are no final states. Indeed, $\mathcal{M}(P)$ models the *method call* relationship: $m \rightarrow m'$ if and only if $\exists pc \cdot \mathit{InstrAt}(m, pc) = \texttt{invokevirtual}\ m'$. Whenever understood from the context we will write $\mathcal{M}$ instead of $\mathcal{M}(P)$. Given the method-call graph, we say that a method m' is *reachable* from m if and only if there is a path in the graph from m to m'. More formally, $\mathit{Reach}(m) = \{m' \mid m \rightarrow^{+} m'\}$. Given the graph of method calls, we define the following predicate which determines whether a given method m is recursive:

$$\mathit{MutRec}(m) \equiv m \in \mathit{Reach}(m).$$

The following predicate characterises not only the mutually reachable methods but also the methods reachable from those:

$$\mathit{MutRecReach}(m) \equiv \exists m' \cdot \mathit{MutRec}(m') \wedge m \in \mathit{Reach}(m').$$

We introduce a *satisfaction* relation: $\mathcal{M} \models \phi$ iff $\mathcal{M}$ satisfies the property ϕ. We state now soundness and completeness of *Rec*.

Theorem 4 (Soundness and Completeness of *Rec*) *$\mathit{Rec}_{m,pc}$ if and only if $\mathcal{M} \models \mathit{MutRecReach}(m)$.* □

4.2.4 *Main algorithm*

We state now the soundness and completeness of our algorithm with respect to our abstraction, which follow directly from the definition of Γ and the soundness and completeness of *Loop*, *Loop'* and *Rec*.

Theorem 5 (Soundness of the algorithm) *If $\Gamma(m, pc) = \infty$, for some (m, pc), then $\mathit{InstrAt}(m, pc) = \texttt{new}(cl)$ and such instruction occurs in a syntactic cycle and/or in a recursive method. Furthermore, if $\Gamma(m, pc) = 1$, for some (m, pc), then $\mathit{InstrAt}(m, pc) = \texttt{new}(cl)$ and $\mathit{InstrAt}(m, pc)$ is not in a syntactic cycle nor in a recursive method.*

Theorem 6 (Completeness of the algorithm) *If $\mathit{InstrAt}(m, pc)$ occurs in a syntactic cycle and/or in a recursive method and $\mathit{InstrAt}(m, pc) = \texttt{new}(cl)$, then $\Gamma(m, pc) = \infty$. On the other hand, if $\mathit{InstrAt}(m, pc) = \texttt{new}(cl)$ does not occur in a syntactic cycle nor in a recursive method then $\Gamma(m, pc) = 1$.*

Notice that the above soundness and completeness result are with respect to an abstraction, which is the identification of `new` instructions inside *syntactic* cycles. However, the real interesting conclusion is:

Corollary 2 *If $\Gamma(m, pc) = 1$ then for any real execution of the applet, $\mathit{InstrAt}(m, pc)$ will be executed a finite number of times.* □

Our algorithm may be easily refined to give an upper bound for the memory used by an applet if no `new` instruction occurs inside a loop. This may be done by simply counting their occurrences and considering the memory allocated by each `new`, as done in [6].

$$\frac{(m, pc) : \texttt{throw}\ e \qquad (m, pc') \in \mathit{findHandler}(m, pc, e)}{F(\mathit{Loop}(m, pc)) \sqsubseteq \mathit{Loop}(m, pc')} \tag{16}$$

$$\frac{(m, pc) : \texttt{throw}\ e \qquad (m', pc') \in \mathit{findHandler}(m, pc, e) \qquad m' \neq m}{G(\mathit{Rec}(m, pc), m') \sqsubseteq \mathit{Rec}(m', pc')} \tag{17}$$

Table 4: Added rules for `throw`

$$\frac{(m, pc) : \texttt{jsr}\ pc'}{\begin{array}{c} F(\mathit{Loop}(m, pc)) \sqsubseteq \mathit{Loop}(m, pc') \\ F(\mathit{Loop}(m, pc)) \sqsubseteq \mathit{Loop}(m, pc+1) \end{array}} \tag{18}$$

$$\frac{(m, pc) : \texttt{ret}\ i}{\bot \sqsubseteq \mathit{Loop}(m, \mathrm{END}_{ret})} \tag{19}$$

Table 5: Added rules for `jsr`

5 Handling exceptions and subroutines

One advantage of the rule-based approach presented in the previous section is its ease of extension. We sketch how to extend the rules for *Loop* and *Rec* in order to cover particular cases of exception handling and subroutine calls.

5.1 Exceptions

According to the operational semantics of the JCVM, an exception is raised either when a program instruction violates its semantic constraints, or when the current instruction is a `throw`. To illustrate the flexibility and modularity of the approach, we extend the algorithm of the previous section to handle exceptions raised by the `throw` instruction. Exceptions are slightly difficult to treat statically because finding the right exception handler may require searching through the frame stack. Obviously, this is not possible at compile time, hence we use a function $\mathit{findHandler}(m, pc, e)$ [16] that over-approximates the set of possible handlers that could possibly catch exception e when raised from location (m, pc).

The rules in Table 4 extend the constraints on *Loop* and *Rec* to handle explicit exceptions. Rule (16) takes care of the case when there is a handler for the exception in the current method. When there is a handler outside the current method, a `throw` is treated like a method invocation –rule (17).

5.2 Subroutines

The `finally` block of a `try`...`finally` Java construct is usually compiled into a subroutine, a fragment of code called with the `jsr` bytecode instruction. We treat subroutines as in Leroy's bytecode verifier [11], i.e. "as a regular branch that also pushes a value of type 'return address' on the stack; and `ret` is treated as a branch that can go to any instruction that follows a `jsr` in the current method". We assume that the applet being analysed has passed bytecode verification, guaranteeing the above property. Notice that the treatment of `ret` represents a considerable loss of precision since the analysis should take into account all the *pc*-numbers after the `jsr` instruction as possible return addresses of the subroutine. To simplify the presentation, in Table 5 we consider that the return address is always the instruction immediately after the `jsr` instruction (like in method calls); END_{ret} may be defined similarly as END_m. The more general case described by Leroy could easily be represented with the help of an auxiliary set function, analogous to *findHandler* (cf. Section 5.1).

6 Final discussion

This work is a first step in the construction of a certified, on-card analyser for estimating memory usage on Java cards. We have given a constraint-based algorithm which detects all the potential loops and recursive methods in order to determine the presence of dynamic class instantiations inside cycles. We have proved its soundness and completeness w.r.t. an abstraction of the operational semantics. Our abstraction is conservative and correct w.r.t. a run-time execution of the program, in the sense that if a run-time cycle exists, then such cycle is detected by our algorithm, and if our analysis gives as an answer that no `new` instruction is inside a cycle, then it is indeed the case. Besides the advantages mentioned in the introduction, the modularity of our algorithm allows the analyser, as well as the functions *Loop*, *Loop'* and *Rec*, to be reused by other constraint-based analysers as programs which have been proved correct.

We have defined two functions, *Loop* and *Loop'* for detecting cycles. Notice that we could have defined only *Loop* just changing the rule of `invokevirtual` to:

$$\frac{(m, pc) : \texttt{invokevirtual}\, m'}{\begin{array}{l} \mathit{Loop}(m, pc) \sqsubseteq \mathit{Loop}(m', 1) \\ \mathit{Loop}(m, pc) \sqsubseteq \mathit{Loop}(m, pc + 1) \end{array}}$$

In this case we would not need the definition of *Loop'*. Our choice is, however, not arbitrary and it is motivated by modularity and memory usage concerns. Computing the intra-procedural cycles first yields a local reasoning which may be done once and for all, also minimising the auxiliary memory used.

Complexity of the analysis. Let N be the number of bytecode instructions, N_U the number of unconditional instructions (including `throw`) and N_C the number of conditional jump instructions in a given method m. If B is the number of bits used for representing a *pc*-number, then the memory needed to compute *Loop* using the algorithm in Section 3.1 is bounded by $N \times ((N_U + 2N_C + 1) \times B + 1)$. The second factor is a bound on the size of $\mathit{Loop}(m, pc)$ for a fixed *pc*, where $(N_U + 2N_C + 1)$ estimates the number of basic blocks in method m: besides the first basic block, each unconditional jump instruction may determine one basic block, and each conditional jump may determine two basic blocks. An instruction found to be in a potential cycle needs one further bit (corresponding to $\bullet$). This information is propagated along the method, and in the worst case to all of its instructions. For *Rec* the auxiliary memory used is definitely less, since we only propagate method's names, which might be associated to each method without needing to propagate them to every method line, although the specification presented here does it. The time efficiency of our algorithm strongly depends on the efficiency of the constraint solver, but we believe that in principle each fix-point computation converges in at most three iterations. Moreover, the order in which the different predicates are applied may drastically improve the performance and space-complexity of the analyser. By computing *Rec* first, for instance, we would not need to compute *Loop* for the methods already marked as recursive.

A moderately complex Java Card method may have around 50 basic blocks (see [11, Section 2.3]). Considering a method with up to 200 instructions and 50 basic blocks the computation of *Loop* would require approximately 10KB of memory in the worst case. This still exceeds the 4 KB of RAM available in top-quality smart cards, so it is not yet feasible to deploy the analyser on-card. There is possibly enough transient memory (EEPROM) to store the data structures used by the analyser, but its latency (1-10 ms per write operation) would make the analysis extremely slow. On the other hand, the analysis could certainly fit in, for instance, Java-enabled mobile phones.

Related Work. Ad-hoc type systems are probably the most successful tools for guaranteeing that well-typed programs run within stated space-bounds. Previous work along these lines can be found in the context of typed assembly [2, 17] and functional languages [3, 8, 9]. In [2], the authors present a first-order linearly typed assembly language which allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints. Vanderwaart et al. [17] describe a type theory for certified code, with the aim of guaranteeing bounds on CPU usage, by using an instruction-counting mechanism, or "virtual clock". The approach may be extended to ensure bounds on other resources as well. Another recent work is [1] where the resource bounds problem is studied on a simple stack machine. The authors show how to perform type, size and termination verifications at the level of the bytecode. In [13] it is shown how to import ideas from data flow analysis, using abstract interpretation, into inter-procedural control flow analysis. Even though the motivations and the techniques are not the same, it is worth mentioning the work done by the Mobile Resource Guarantees project [12], which applies ideas from proof-carrying code for solving the problem of resource certification for mobile code.

We took inspiration from [5] where a similar technique is applied with success to extract a data flow analyser for Java card bytecode programs. More recently, a Coq-certified constrained-based memory usage analyser has been presented in [6], also based on the formalism introduced in [5]. The analyser has been automatically obtained from its correctness proof using Coq's extraction mechanism. Although such algorithm has simpler rules for computing *Loop*, the space complexity is higher than the one presented here.

To compute *Loop* we have chosen to keep the *pc*-numbers of the target of any jump whereas [6] keeps the *pc*-numbers of all the instructions already visited. Given an applet of 200 lines with 50 basic blocks (including exception handlers) the auxiliary memory for computing *Loop* (in the worst case) using the algorithm defined in [6] would reach about 40 KB, contrasting with the 10 KB of our algorithm.

Future Work. Due to the inter-procedural dependencies the analysis is not compositional: if two applets are "loop-safe" (i.e., no `new` occurs inside a cycle), their composition is not necessarily loop-safe. We should, in principle, compute *Rec* again for the composite applet in order to guarantee the non-existence of `new` instructions inside newly created recursive methods. However, it is possible to minimise the computation of a global fix-point (for detecting recursive methods) by keeping relevant information regarding the methods of one applet called by methods of the other (and vice-versa). Our analyser may be easily extended in that direction.

In Section 5 we have given an idea of how to extend our rules in order to handle exceptions and subroutines. We intend to extend our approach to cover the remaining cases of exception handling and to improve the precision of the subroutine analysis.

The space complexity of the intra-procedural analysis may be improved by propagating the *pc*-numbers only to the beginning of a basic block instead of to all the internal instructions. In this case the memory required would be bounded by $((N_U + 2N_C + 1)^2 \times B) + (N_U + 2N_C + 1)$; on an applet with 50 basic blocks (and at most 256 instructions), it would need only 2.5 KB, leaving on-card verification at reach.

Acknowledgements. The second author is indebted to Thomas Jensen and David Pichardie for insightful discussions in an early stage of this work. Patrick Bernard, Eduardo Giménez, Arnaud Gotlieb and Jean–Louis Lanet have provided us with useful information on the Java card technology.

References

[1] R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. Research report 17-2004, LIF, Marseille, France, 2004.

[2] D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *J. Autom. Reason.*, 31(3-4):261–302, 2003.

[3] D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP*, volume 2305 of *LNCS*, pages 36–52, 2002.

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[5] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *ESOP*, LNCS, pages 385–400, 2004.

[6] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods*, volume 3582 of *LNCS*, pages 91–106, 2005. To appear.

[7] Zhiqun Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison Wesley, 2000.

[8] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[9] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[10] X. Leroy. Java bytecode verification: an overview. In *CAV'01*, volume 2102 of *LNCS*, pages 265–285. Springer-Verlag, 2001.

[11] X. Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.

[12] MRG. Mobile Resource Guarantees project. See `http://groups.inf.ed .ac.uk/mrg/`.

[13] F. Nielson and H.R. Nielson. Interprocedural control flow analysis. In *European Symposium on Programming*, pages 20–39, 1999.

[14] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[15] D.A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *SAS*, LNCS, pages 351–380, 1998.

[16] I. Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Algebraic Programming*, 58(1-2), 2004.

[17] J.C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.