Certified Memory Usage Analysis*

David Cachera[1], Thomas Jensen[2], David Pichardie[1], and Gerardo Schneider[2,3]

[1] IRISA/ENS Cachan (Bretagne), Campus de Ker Lann, 35170 Bruz, France
[2] IRISA/CNRS, Campus de Beaulieu, 35042 Rennes cedex, France
[3] Dept. of Informatics, Univ. of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway

Abstract. We present a certified algorithm for resource usage analysis, applicable to languages in the style of Java byte code. The algorithm verifies that a program executes in bounded memory. The algorithm is destined to be used in the development process of applets and for enhanced byte code verification on embedded devices. We have therefore aimed at a low-complexity algorithm derived from a loop detection algorithm for control flow graphs. The expression of the algorithm as a constraint-based static analysis of the program over simple lattices provides a link with abstract interpretation that allows to state and prove formally the correctness of the analysis with respect to an operational semantics of the program. The certification is based on an abstract interpretation framework implemented in the Coq proof assistant which has been used to provide a complete formalisation and formal verification of all correctness proofs.

Keywords: Program analysis, certified memory analysis, theorem proving, constraint solving.

1 Introduction

This paper presents a certified algorithm for resource usage analysis, aimed at verifying that a program executes in bounded memory. Controlling the way that software consumes resources is a general concern to the software developer, in particular for software executing on embedded devices such as smart cards where memory is limited and cannot easily be recovered. Indeed, for Java Card up to version 2.1 there is no garbage collector and starting with version 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. This has lead to a rather restrictive programming discipline for smart cards in which the programmer must avoid memory allocation in parts of the code that are within loops. We provide a certified analysis that automatically and efficiently can check that such a programming discipline is respected on a Java Card. This analysis can be deployed in two contexts:

1. As part of a software development environment for smart cards. In that case, it will play a role similar to other program analyses used in type checking and optimisation.

* This work was partially supported by the French RNTL project "Castles".

J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 91–106, 2005.

2. As part of an extended *on-card* byte code verifier that checks applets and software down-loaded on the card after it has been issued.

In both scenarios, there is a need for certification of the analysis. In the first case, the analysis will be part of a software development process satisfying the requirements of the certification criteria. In the second case, the analysis will be part of the card protection mechanisms (the so-called Trusted Computing Base) that have to be certified. The current implementation has a time complexity that is sufficiently low to integrate it in a development tool. However, we have not yet paid attention to the space complexity of the algorithm and current memory consumption excludes any analysis to take place on-device.

The analysis is a constraint-based static analysis that works by generating a set of constraints from the program byte code. These constraints define a number of sets that describe *a)* whether a given method is (mutually) recursive or can be called from (mutually) recursive methods, and *b)* whether a method can be called from intra-procedural cycles. This information is then combined to identify memory allocations (or any other type of resource-sensitive instructions) that could be executed an unbounded number of times. By casting the analysis as a constraint-based static analysis we are able to give a precise semantic definition of each set and use the framework of abstract interpretation to prove that the analysis provide correct information for all programs. The paper offers the following contributions:

- A constraint-based static analysis that formalises a loop-detecting algorithm for detecting methods and instructions that may be executed an unbounded number of times.
- A formalisation based on abstract interpretation of the link between the analysis result and the operational semantics for the underlying byte code language.
- A certification of the analysis in the form of a complete formalisation of the analysis and the correctness proof within the Coq theorem prover.

The paper is organised as follows. Section 2 briefly introduces the byte code language of study. Section 3 gives an informal presentation of the algorithm and its relation to an operational trace semantics. In Section 4 we formalise the correctness relationship. In Section 5 we give a general description of the structure of the Coq proof. Section 6 exposes some complexity considerations and presents some benchmarks. Section 7 describes the background for this work and compares with existing resource analyses. Section 8 concludes.

2 Java Card Byte Code

Our work is based on the Carmel intermediate representation of Java Card byte code [11]. The Carmel language consists of byte codes for a stack-oriented machine whose instructions include stack operations, numeric operations, conditionals, object creation and modification, and method invocation and return. We do not deal with subroutines (the Java `jsr` instruction) or with exceptions. These instructions can be treated in our framework but complicates the control flow and may lead to inferior analysis results.

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{instr}}{\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{instr}} \langle\!\langle h, \langle m, pc+1, l', s'\rangle, sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{if } pc' \quad n = 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{if } pc'} \langle\!\langle h, \langle m, pc', l, s\rangle, sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{if } pc' \quad n \neq 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{if } pc'} \langle\!\langle h, \langle m, pc+1, l, s\rangle, sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{goto } pc'}{\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{goto } pc'} \langle\!\langle h, \langle m, pc', l, s\rangle, sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{new } cl \quad \exists c \in \text{classes}(P) \text{ with nameClass}(c) = cl \quad (h', loc) = \text{newObject}(cl, h)}{\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{new } cl} \langle\!\langle h', \langle m, pc+1, l, loc :: s\rangle, sf\rangle\!\rangle}$$

$$\frac{\begin{array}{c}\text{instructionAt}_P(m, pc) = \texttt{invokevirtual } M \\ h(loc) = o \quad m' = \texttt{methodLookup}(M, o) \quad f = \langle m, pc, l, loc :: V :: s\rangle \\ f' = \langle m', 1, V, \varepsilon\rangle \quad f'' = \langle m, pc, l, s\rangle\end{array}}{\langle\!\langle h, f, sf\rangle\!\rangle \rightarrow_{\texttt{invokevirtual } M} \langle\!\langle h, f', f'' :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{return} \quad f' = \langle m', pc', l', s'\rangle}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle, f' :: sf\rangle\!\rangle \rightarrow_{\texttt{return}} \langle\!\langle h, \langle m', pc'+1, l', v :: s'\rangle, sf\rangle\!\rangle}$$

Fig. 1. Carmel operational semantics

The formal definition of the language is given as a small-step operational semantics with a state of the form $\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle$, where h is the heap of objects, $\langle m, pc, l, s\rangle$ is the current *frame* and sf is the current call stack (a list of frames). A frame $\langle m, pc, l, s\rangle$ contains a method name m and a program point pc within m, a set of local variables l, and a local operand stack s (see [15] for details). Let State_P be the set of all the states of a given program P. We will write simply State if P is understood from the context. The transition relation $\rightarrow_I$ describes how the execution of instruction I changes the state. This is extended to a transition relation $\rightarrow$ on traces such that $tr ::: s_1 \rightarrow tr ::: s_1 ::: s_2$ if there exists an instruction I such that $s_1 \rightarrow_I s_2$[1].

The instructions concerned with control flow and memory allocation: `if`, `goto`, `invokevirtual`, `return` and `new`, need a special treatment in our analysis. The rest of the instructions may have different effects on the operand stack and local variables but behave similarly with respect to memory and control flow (move to the next instruction without doing any memory allocation). For clarity and in order to focus on the essentials, these instructions have been grouped into one generic instruction `instr` with this behaviour. Fig. 1 shows the rules describing the operational semantics of Carmel.

The rule for the generic instruction `instr` is formalised as a (non-deterministic) transition from state $\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle$ to any state of form $\langle\!\langle h, \langle m, pc+1, l', s'\rangle, sf\rangle\!\rangle$. Instructions `if` and `goto` affect the control flow by modi-

[1] Here and everywhere in the paper, ":::" denotes the "cons" operation for traces (appending an element to the right of the trace). We will use "::" as the "cons" operation of the operand stack (the top of the stack being on the left).

fying the pc component of the state. The `if` instruction produces a jump to an indicated program point pc' if the top of the operand stack is 0; otherwise it moves to the instruction $pc + 1$. The `goto` pc' unconditionally jumps to pc'. The `new` instruction modifies the heap (h') creating an object of class cl on location loc; loc is added to the stack and the pc is incremented.

The rule for `invokevirtual` is slightly more complicated. Let M be a method name. The instruction `invokevirtual` M at address (m, pc) of state $\sigma = \langle\!\langle h, f, sf \rangle\!\rangle$ may only occur if the current frame f of σ has an operand stack of the form $loc :: V :: s$, i.e., it starts with a *heap location* denoted by *loc*, followed by a vector of values V. The actual method that will be called is to be found in the object o that resides in the heap h at the address $h(loc)$, and the actual parameters of that method are contained in the vector V. Then, the `methodLookup` function searches the class hierarchy for the method name M in the object o, and returns the actual method to which the control will be transferred. The new method, together with its starting point $pc = 1$, its vector V of actual parameters, and an empty operand stack ε, constitute a new frame f' pushed on top of the call stack of the resulting state $\sigma' = \langle\!\langle h, f', f'' :: sf \rangle\!\rangle$, where $f'' = \langle m, pc, l, s \rangle$ is the frame to be taken into account after the completion of the method invocation. Finally, the `return` instruction pops the control stack and execution continues at the program point indicated in the frame that is now on top of the control stack.

The *partial trace semantics* $[\![P]\!]$ of a Carmel program P is defined as the set of reachable partial traces:

$$[\![P]\!] = \left\{ s_0 :: s_1 :: \cdots :: s_n \in \text{State}^+ \;\middle|\; \begin{array}{l} s_0 \in \mathcal{S}_{init} \wedge \\ \forall k < n,\ \exists i,\ s_k \rightarrow_i s_{k+1} \end{array} \right\} \in \wp(\text{State}^+)$$

where $\mathcal{S}_{init}$ is the set of initial states.

3 Specification of the Analysis

The memory usage analysis detects inter- and intra-procedural loops and checks if the creation of new objects may occur inside such loops, leading to unbounded memory consumption. Intuitively, the algorithm consists of the following steps:

1. Compute the set of potential ancestors of a method m in the call graph: $Anc(m)$;
2. Determine the set of methods that are reachable from a mutually recursive method: $MutRecR$;
3. Compute the set of potential predecessors of a program point pc in a method m: $Pred(m, pc)$;
4. Determine the set of methods that may be called from intra-procedural loops: $LoopCall$;
5. Combining all these results ($Unbounded(P)$): phases 1 to 4 are used to detect if a new object creation may occur in a loop, leading to a potentially unbounded memory usage.

Notice that step 3 is the only intra-procedural computation. In the following, we describe the rules for obtaining each of the above-mentioned sets and explain informally how they are related to the operational semantics. This relationship is formalised in Section 4 which proves the correctness of the analysis.

$$\frac{(m, pc) : \texttt{invokevirtual}\ m_{\text{ID}} \qquad m' \in \text{implements}(P, m_{\text{ID}})}{Anc(m) \cup \{m\} \subseteq Anc(m')}$$

Fig. 2. Rule for Anc

$$\frac{m \in Anc(m)}{\{m\} \subseteq MutRecR} \qquad \frac{Anc(m) \cap MutRecR \neq \emptyset}{\{m\} \subseteq MutRecR}$$

Fig. 3. Rules for $MutRecR$

3.1 Computing Ancestors of a Method (Anc)

Anc associates to each method name the set of potential ancestors of this method in the call graph. The type of Anc is thus $\text{methodName} \rightarrow \wp(\text{methodName})$. Fig. 2 shows the rule corresponding to the `invokevirtual` instruction for computing the set $Anc(m')$: for each method m' which may be called by a method m, it determines that the set of ancestors of m' must contain m as well as all the ancestors of m. The function $implements$ is a static over-approximation of the dynamic method lookup function. It returns all possible implementations of a given method with name m_{ID} relative to a program P. We do not specify it in further detail. No constraint is generated for any other instruction different from `invokevirtual` since we are here interested only in the method call graph.

Intuitively, given a trace, if the current method being executed is m, then $Anc(m)$ contains all the methods appearing in the current stack frame.

3.2 Determining Mutually Recursive Methods ($MutRecR$)

$MutRecR$ contains the mutually recursive methods as well as those reachable from a mutually recursive method. Fig. 3 shows the rules used to compute the set $MutRecR$: if m is in the list of its ancestors, then it is mutually recursive, and all the descendants of a mutually recursive method are reachable from a mutually recursive method. The result of the computation of $MutRecR$ can be seen as a marking of methods: methods reachable from mutually recursive methods may be called an unbounded number of times within the execution of an inter-procedural loop. Instructions in these methods may be executed an unlimited number of times. For an example, see Fig. 4: methods are represented with rectangles, thin arrows represent local jumps (`goto`), and thick arrows represent method invocations. Shaded methods are those in $MutRecR$.

Intuitively, given a trace where the current method being executed is m, if $m \notin MutRecR$, then m does not appear in the current stack frame, and all methods in this stack frame are distinct.

3.3 Computing Predecessors of a Program Point ($Pred$)

Given a method m, $Pred(m, pc)$ contains the set of predecessors of the program point pc in the intra-procedural control flow graph of method m. The type of $Pred$ is thus $\text{methodName} \times \text{progCount} \rightarrow \wp(\text{progCount})$. Fig. 5 shows the rules (one for each

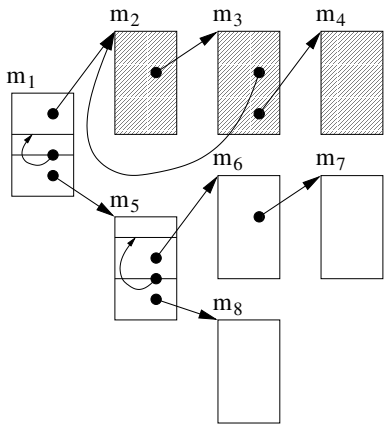

Fig. 4. Example of mutually recursive reachable methods

$$\frac{(m,pc) : \texttt{instr}}{Pred(m,pc) \cup \{pc\} \subseteq Pred(m,pc+1)} \qquad \frac{(m,pc) : \texttt{if } pc'}{\begin{array}{l} Pred(m,pc) \cup \{pc\} \subseteq Pred(m,pc+1) \\ Pred(m,pc) \cup \{pc\} \subseteq Pred(m,pc') \end{array}}$$

$$\frac{(m,pc) : \texttt{goto } pc'}{Pred(m,pc) \cup \{pc\} \subseteq Pred(m,pc')}$$

Fig. 5. Rules for $Pred$

instruction) used for defining $Pred$. For instructions that do not induce a jump (`instr` stands for any instruction different from `if` and `goto`), the set of predecessors of a program point pc, augmented with pc itself, is transferred to its direct successor $pc+1$. For the `if` instruction, the two branches are taken into account. For a `goto` instruction, the set of predecessors of the current program point pc, augmented with pc itself, is transferred to the target of the jump.

To relate $Pred$ to the execution traces, we need to define the notion of *current execution* of a method: the current execution of a method m in a trace $tr' = tr ::: \langle\langle h, \langle m, pc, l, s\rangle, sf\rangle\rangle$ is the set of all program points (m, pc') appearing in a maximal suffix of tr' that does not contain a program point where a call to m is performed. Intuitively, given a trace, $Pred(m, pc)$ represents the set of all programs points pc' appearing in the current execution of m.

3.4 Determining Method Calls Inside Loops ($LoopCall$)

The $LoopCall$ set contains the names of the methods susceptible to be executed an unbounded number of times due to intra-procedural loops. Fig. 6 shows the rules used for computing $LoopCall$. The first rule says that if a method m' is possibly called by a method m at program point pc, and if pc is within an intra-procedural loop of m (pc is in the set of its predecessors), then m' may be called an unbounded number of times.

$$\frac{(m, pc) : \texttt{invokevirtual}\ m_{\text{ID}} \quad m' \in \text{implements}(P, m_{\text{ID}}) \quad pc \in Pred(m, pc)}{\{m'\} \subseteq LoopCall}$$

$$\frac{(m, pc) : \texttt{invokevirtual}\ m_{\text{ID}} \quad m' \in \text{implements}(P, m_{\text{ID}}) \quad m \in LoopCall}{\{m'\} \subseteq LoopCall}$$

Fig. 6. Rules for $LoopCall$

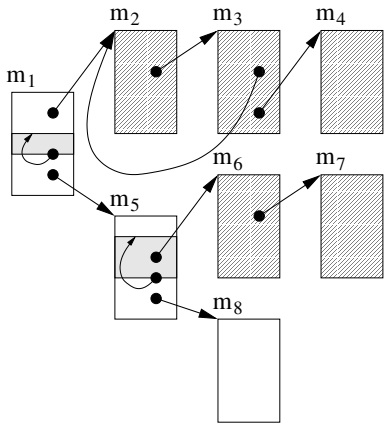

Fig. 7. Marking methods called from inside an intra-procedural loop

Furthermore, if m may be called an unbounded number of times and m calls m', then this property is inherited by m'.

Intuitively, given a trace tr where the method currently being executed is m, if $m \notin LoopCall$ then for each method m' at point pc' performing a call to m, (m', pc') appears only once in the current execution of m'. For an example of the result of this phase of the algorithm, see Fig. 7. The newly shaded methods m_6 and m_7 are in $LoopCall$ because of the call from within the loop in method m_5.

3.5 The Main Predicate ($Unbounded(P)$)

So far, the constraints we defined yield an algorithm that detects inter- and intra-procedural loops of a given program P. We now can specialise this algorithm for determining if the memory usage of our program is certainly bounded. The final result consists in a predicate $Unbounded(P)$ which is computed by the rule depicted in Fig. 8. This rule sums up the previous results, by saying that if a new object creation may

$$\frac{(m, pc) : \texttt{new}_o\ cl \qquad m \in MutRecR \vee m \in LoopCall \vee pc \in Pred(m, pc)}{Unbounded(P)}$$

Fig. 8. Rule for $Unbounded(P)$

occur inside a loop (directly or indirectly, as described by the sets Anc, $LoopCall$, $MutRecR$ and $Pred$) then $Unbounded(P)$ is true.

4 Correctness

The correctness proof follows a classic abstract interpretation approach in which we show that the information computed by the constraints is an invariant of the trace semantics of a program P. For each previously defined function or set X (Anc, $MutRecR$, $LoopCall$, $Pred$ and $Unbounded(P)$) we use the following schema:

1. Prove that all the domains are lattices and that they have no infinite, strictly increasing chains (ascending chain condition).
2. Determine a set of constraints for defining X.
3. Define a concretisation function γ_X in order to relate concrete domains (sets of traces) and abstract domains (X).
4. Prove that all partial traces of a given program are correctly approximated by X, i.e., that $\forall t \in [\![P]\!],\ t \in \gamma_X(X)$. This result is a consequence of the classical characterisation of $[\![P]\!]$ as the least element of the following set:

$$\left\{ S \in \wp(\text{Trace}) \;\middle|\; \mathcal{S}_{init} \subseteq S \ \wedge \begin{array}{l} \forall t_1, t_2 \in \text{Trace}, \\ \text{if } t_1 \in S \text{ and } t_1 \rightarrow t_2 \\ \text{then } t_2 \in S \end{array} \right\}$$

   We must prove the following two intermediary lemmas:

$$\textit{For any trace } t_1 \in [\![P]\!], \textit{ if } t_1 \in \gamma_X(X) \textit{ and } t_1 \rightarrow t_2, \textit{ then } t_2 \in \gamma_X(X). \quad (1)$$

$$\textit{For any trace } t \in \mathcal{S}_{init},\ t \in \gamma_X(X). \quad (2)$$

5. Analyse a given applet P, which consists then of 1) constructing the set of constraints associated to the program 2) solving this system with a classic fixed point iteration whose termination is ensured by the lattice ascending chain condition.

Steps 1 to 4 are proof-theoretical while step 5 is algorithmic. All these steps are performed in the Coq proof assistant. Steps 1, 2 and 5 benefit from the framework proposed in [3] and thus no new proof is required. We only need to prove steps 3 and 4, for which the property (1) represents the core of the work:

Lemma 1. *For any trace $t_1 \in [\![P]\!]$, if $t_1 \in \gamma_X(X)$ and $t_1 \rightarrow t_2$, then $t_2 \in \gamma_X(X)$.*

We now define the concretisation functions γ_X for Anc, $MutRecR$, $Pred$ and $LoopCall$.

Anc. The concretisation function for Anc formalises the fact that m' calls m (directly or indirectly) in a trace t by examining the call stack of each element in t:

$$\begin{array}{l} \gamma_{Anc} : (\text{methodName} \rightarrow \wp(\text{methodName})) \longrightarrow \wp(\text{State}^+) \\ \qquad X \mapsto \left\{ t \in \text{State}^+ \;\middle|\; \begin{array}{l} \text{for all } \langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \text{ in } t \\ \text{for all } m' \text{ appearing in } sf,\ m' \in X(m) \end{array} \right\} \end{array}$$

$MutRecR$. Given a method name m and a partial trace t, we say that "m is ever executed with a safe callstack in t" (which is denoted by the SafeCallStack(m, t) predicate) iff for all $\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle$ in t, m does not appear in sf and all methods in sf are distinct.

The concretisation function for $MutRecR$ is then defined by:

$$\gamma_{MutRecR} : \wp(\text{methodName}) \longrightarrow \wp(\text{State}^+)$$
$$X \mapsto \left\{ t \in \text{State}^+ \;\middle|\; \begin{array}{l} \text{for all } m \in \text{methodName, if } m \notin X, \\ \text{then SafeCallStack}(m, t) \text{ holds} \end{array} \right\}$$

$Pred$. The associated concretisation function is

$$\gamma_{Pred} : (\text{methodName} \times \text{progCount} \to \wp(\text{progCount})) \longrightarrow \wp(\text{State}^+)$$
$$X \mapsto \left\{ t \in \text{State}^+ \;\middle|\; \begin{array}{l} \text{for all prefix } t' ::: \langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \text{ of } t, \\ \text{if SafeCallStack}(m, t) \text{ then } current(t', m) \subseteq X(m, pc) \end{array} \right\}$$

where $current(t', m)$ is the set of program points which appear in the current execution of m relative to the trace t'.

$LoopCall$. Given two method names m and m', and a partial trace t, we use the predicate OneCall to state that m is called at most once within each invocation of m'. Formally, OneCall is defined by OneCall(m, m', t) iff for all prefix $t' ::: \langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle$ of t, and for all positions (m', pc') where a call to m is performed, pc' occurs only once in the corresponding current execution of m'.

The concretisation function for $LoopCall$ is then defined by:

$$\gamma_{LoopCall} : \wp(\text{methodName}) \longrightarrow \wp(\text{State}^+)$$
$$X \mapsto \left\{ t \in \text{State}^+ \;\middle|\; \begin{array}{l} \text{for all prefix } t' ::: \langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle \text{ of } t, \\ \text{if SafeCallStack}(m, t) \text{ and } m \notin X, \\ \text{then for all } m' \text{ in methodName,} \\ \text{OneCall}(m, m', t) \text{ holds} \end{array} \right\}$$

To prove the correctness of $Unbounded(P)$ we need to prove the following lemma:

Lemma 2. *If for all program point (m, pc) where an instruction* `new` *is found we have $m \notin MutRecR \cup LoopCall$ and $pc \notin Pred(m, pc)$, then there exists a bound $Max_{\texttt{new}}$ so that*

$$\forall t \in [\![P]\!],\ |t|_{\texttt{new}} < Max_{\texttt{new}}$$

where $|t|_{\texttt{new}}$ counts the number of `new` *instructions which appear in the states of the trace t.*

To establish the above result we first prove an inequality relation between the number of executions of the different methods. We write $Exec(m, t)$ for the number of executions of a method m found in a trace t. Similarly, $Max_{\texttt{invoke}}(m)$ is the maximum number of `invokevirtual` instructions which appear in a method m. Let $m \in Call(m')$ denote that m' calls m.

Lemma 3. *For all methods m, if $m \notin MutRecR \cup LoopCall$ then for all $t \in [\![P]\!]$,*

$$Exec(m, t) \leq \sum_{m \in Call(m')} Exec(m', t) \cdot Max_{\texttt{invoke}}(m').$$

Using this lemma we prove that the number of executions of the method m in the trace t is bounded, as expressed in the following lemma.

Lemma 4. *There exists a bound Max_{exec} such that for all methods m which verify $m \notin LoopCall \cup MutRecR$, we have*

$$\forall t \in [\![P]\!],\ Exec(m,t) \leq Max_{exec}.$$

To conclude the proof of Lemma 2 we need to prove the following result, establishing that if a method is not (mutually) recursive, nor reachable from a mutually recursive one and it is not in a intra-method cycle, then the number of `new` instructions is bounded.

Lemma 5. *Given a method m which verifies $m \notin MutRecR \cup LoopCall$, if for all program points (m, pc) in m where an instruction* `new` *is found, $pc \notin Pred(m, pc)$ holds then*

$$\forall t \in [\![P]\!],\ |t|^m_{\texttt{new}} \leq Exec(m,t)$$

where $|t|^m_{\texttt{new}}$ counts the number of instructions `new` *which appears in the states of the trace t in the method m.*

Lemma 2 follows then from the following inequality:

$$\forall t \in [\![P]\!],\ |t|_{\texttt{new}} = \sum_m |t|^m_{\texttt{new}} \leq MethodMax_P \cdot Max_{\texttt{invoke}}$$

where $MethodMax_P$ is the number of methods in program P.

The correctness of our analysis is a corollary of Lemma 2:

Theorem 1. $\neg Unbounded(P) \Rightarrow \exists Max_{\texttt{new}},\ \forall t \in [\![P]\!],\ |t|_{\texttt{new}} < Max_{\texttt{new}}$.

5 Coq Development

The following section gives an overview of the structure of the Coq development. It is meant to give an intuition for how the development of a certified analyser can be done methodologically [3] and to serve as a first guide to the site [13] from which the analyser and the Coq specification and proofs can be downloaded, compiled and tested.

The formalisation of Java Card syntax and semantics is taken form an existing data flow analyser formalised in Coq [3]. The analysis consists in calculating the sets Anc, $MutRecR$, $Pred$ and $LoopCall$ that are indexed by program methods and program points. This naturally leads to a representation as arrays of sets, defined in the following way using Coq modules:

```
Module MAnc := ArrayLattice(FiniteSetLattice).
Module MMutRec := FiniteSetLattice.
Module MPred := ArrayLattice(ArrayLattice(FiniteSetLattice))
Module MLoopCall := FiniteSetLattice.
Module MUnbounded := BoolLattice.
```

This leads to a type for *eg.* $Pred$ that is dependent on the actual program P to analyse. Once the program P is supplied, we construct the actual set $Pred$, properly indexed by the methods and program points of P.

Each of the four type of sets gives rise to a specific kind of constraints. For example, the constraints defining the set $Pred$ are given the following definition

```
Inductive ConstraintPred : Set :=
 C4: MethodName -> progCount -> progCount  ->
     (FiniteSetLattice.Pos.set -> FiniteSetLattice.Pos.set)
        -> ConstraintPred.
```

Thus, each constraint is constructed as an element of a data type that for a given method `m` and two instructions at program points `pc` and `pc'` provides the transfer function that links information at one program point to the other. The actual generation of constraints is done via a function that recurses over the program, matching each instruction to see if it gives rise to the generation of a constraint.The following definition corresponds to the Coq formalisation of the constraint rules depicted on Fig. 5.

```
Definition genPred (P:Program) (m:MethodName) (pc:progCount)
                   (i:Instruction) : list ConstraintPred :=
  match i with
   return_v => nil
 | goto pc' => (C4 m pc pc' (fun s => (add_set pc s)))::nil
 | If pc' => (C4 m pc pc' (fun s => (add_set pc s)))::
             (C4 m pc (nextAddress P pc)
                      (fun s => (add_set pc s)))::nil
 | _ => (C4 m pc (nextAddress P pc)
                 (fun s => (add_set pc s)))::nil
```

The result of the constraint generation is a list of constraints that together specify the sets Anc, $Pred$, $MutRecR$ and $LoopCall$. When calculating the solution of the constraint system, we use the technique that the resolution of a constraint system can be done by interpreting each constraint as a *function* that computes information to add to each state and then increment the information associated with the state with this information. Formally, for each constraint of the form $f(X(m, pc_1)) \sqsubseteq X(m, pc_2)$ over an indexed set X (such as $Pred$), we return a function for updating the indexed set by replacing the value of X at (m, pc_2) by the value $f(X(m, pc_1))$.

```
Definition F_Pred (c:ConstraintPred) :
                      MPred.Pos.set -> MPred.Pos.set :=
 match c with
  (C4 m pc1 pc2 f) => fun s => update s m pc2 (f (s m pc1))
```

The resolution of the constraints can now be done using the iterative fix-point solver, as explained in [3]. The fix-point solver is a function of type

```
(l: (L → L) list) → (∀f ∈ l, (monotone L f)) →
  ∃x:A, (∀f ∈ l, (order L (f x) x)) ∧
        (∀ y:A (∀f ∈ l, (order L (f y) y)) ⇒ (order L x y))
```

Subject	**number of lines**
syntax + semantics	1000
lattices + solver	3000
Anc, $MutRecR$, $Pred$, $LoopCall$ correctness	1300
$Unbounded(P)$correctness	2500
constraint collecting, monotonicity	1200
total	9000

Fig. 9. Proof effort for the development

that will take a list of monotone functions over a lattice `L` and iterate these until stabilisation. The proof of this proposition (*ie.* the inhabitant of the type) is a variant of the standard Knaster-Tarski fix-point theorem on finite lattices that constructs (and hence guarantees the existence of) a least fix-point as the limit of the ascending chain $\bot, f(\bot), f^2(\bot), \ldots$.

5.1 Correctness Proof in Coq

The remaining parts of the proof effort are dedicated to the correctness of the memory usage analysis. Two particular points connected with the correctness proof are worth mentioning:

- The correctness of $Unbounded(P)$ requires much more work than the proof of the various partial analyses. This is not surprising because of the mathematical difficulties of the corresponding property: counting proofs are well-known examples of where big gaps can appear between informal and formal proofs.
- In many of the proofs involved in the construction of the analyser, there is one case for each byte code instruction. Most of the cases are dealt with in the same way. For the methodology to scale well, the proof effort should not grow proportional to the size of the instruction set. This is true already for the relatively small Carmel instruction set (15 instructions) and in particular for the real Java Card byte code language (180 instructions).

For the latter point, it was essential to use the Coq tactic language of proof scripts (called *tactics* in Coq) which allows to apply the same sequence of proof steps to different subgoals, looking in the context for adequate hypothesis. In this way, most of our proofs are only divided in three parts: one case for `invokevirtual`, one case for `return` and one case (using an appropriate tactics applied on several subgoals) for the other instructions. With such a methodology, we can quickly add simple instructions (like operand stack manipulations) without modifying any proof scripts.

The extracted analyser is about 1000 lines of OCaml code while the total development is about 9000 lines of Coq. The following table gives the breakdown of the proof

effort measured in lines of proof scripts[2]. Fig. 9 summarises the proof effort for each part of the certified development of the analyser.

6 Complexity and Benchmarks

The computation of the final result of the algorithm from the constraints defined above is performed through well-known iteration strategies. Let N denote the number of methods and I_m the number of instructions in method m. The computation of the sets Anc, $MutRecR$ and $LoopCall$ consists in a fix-point iteration on the method call graph, that is at most quadratic in N. The computation of $Pred$ for a given method m requires at most $I_m \times (|$ number of jumps in $m \mid +1)$ operations. The computation of $Unbounded(P)$ requires $\sum_m I_m \leq N \times \max\{I_m\}$ operations and in the worst case to save I_m line numbers for each instruction (i.e., I_m^2). The algorithm may be further optimised by using a more compact representation with intervals but we have not implemented this.

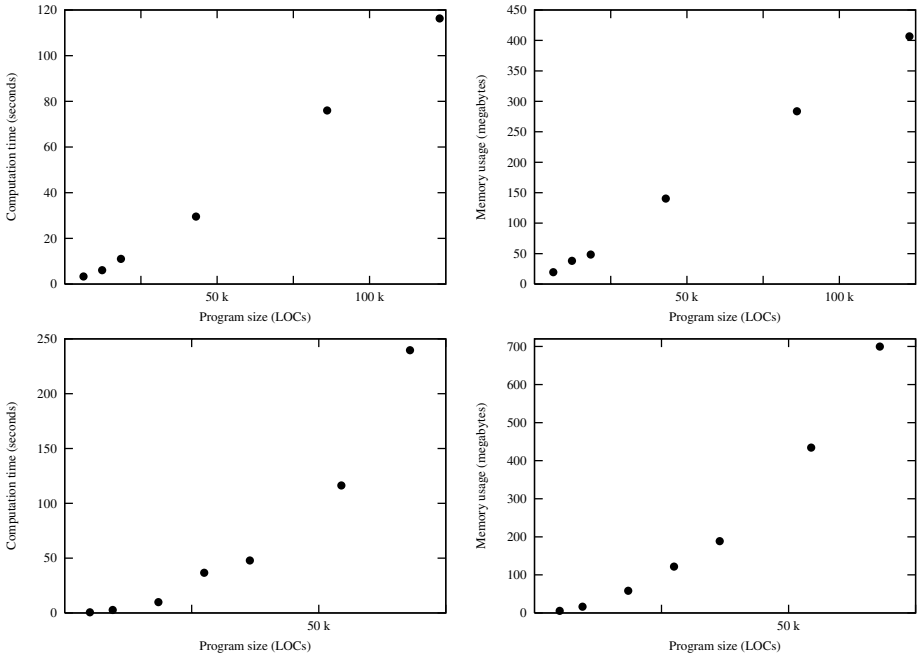

Fig. 10. Performance measures. The first row corresponds to a variable number of methods with a fixed number of lines per method, while the second corresponds to a fixed number of methods with a variable number of lines per method

[2] Note that the size of a Coq development can change significantly from one proof script style to another. The same proofs could have consumed two or three times more script lines if the capabilities of the proof tactics language were not exploited. Thus, it is the relative size of the proofs that is more important here.

Fig. 10 gives benchmarks for the performance of the extracted program. These measure have been performed with a randomly byte code program generator. Given two parameters N and l, this program generates a well formed Carmel program with N methods, each of them containing $6 \cdot l$ lines of byte code. Each group of 6 lines handles a call to a randomly chosen method, a `goto` and a `if` instruction with an apparition probability of $1/5$. Hence we can easily measure the performance of our extracted program on big Carmel programs. The first row of Fig. 10 corresponds to a variable number of methods with a fixed number of lines per method, while the second corresponds to a fixed number of methods with a variable number of lines per method. These benchmarks show a linear performance in the first case (both in computation time and memory requirements), and a quadratic performance in the latter.

As the benchmarks show, the extracted program performs very well, in particular when taking into account that no modification on the extracted code was necessary.

7 Related Work

Hofmann [7] has devised a type system for bounded space and functional in-place update. In this system, a specific $\diamond$-type is used to indicate heap cells that can be overwritten. A type system for a first-order functional language defines when the reuse of heap cells due to such type annotations is guaranteed not to alter the behaviour of the program. Inspired by this work and by Typed Assembly Language of Morrisett et al. [12], Aspinall and Campagnoni [1] have defined heap-bounded assembly language, a byte code language equipped with specific *pseudo-instructions* for passing information about the heap structure to the type system. The type system use linearity constraints to guarantee absence of aliasing. Together, this allows to prove the sound reuse of heap space in the presence of kinds of heap cells (integers, list cells, *etc*).

Crary and Weirich [5] define a logic for reasoning about resource consumption certificates of higher-order functions. The certificate of a function provides an over-approximation of the execution time of a call to the function. The logic only defines what is a correct deduction of a certificate and has no inference algorithm associated with it. The logic is about computation time but could be extended to measure memory consumption.

The most accurate automatic, static analysis of heap space usage is probably the analysis proposed by Hofmann and Jost [8] that operates on first-order functional programs. The analysis both determines the amount of free cells necessary before execution as well as a safe (under)-estimate of the size of a *free-list* after successful execution of a function. These numbers are obtained as solutions to a set of linear programming (LP) constraints derived from the program text. Automatic inference is obtained by using standard polynomial-time algorithms for solving LP constraints. The correctness of the analysis is proved with respect to an operational semantics that explicitly keeps track of the memory structure and the number of free cells.

The Hofmann-Jost analysis is more precise than the analysis presented here but is too costly to be executed on most embedded devices, in particular smart cards. Rather, its use lies in the generation of certificates that can then be checked on-card. A simi-

lar distinction can been observed in on-card byte code verification where the on-card verifier of Casset et al. [4] relies on certificates generated off-card, whereas the verifier described by Leroy [10] imposes slight language restrictions so that the verifier can execute on-card.

A similar (but less precise) analysis to ours is presented in [14]. The analysis is shown to be correct and complete w.r.t. an abstraction of the operational semantics. One difference with our work is the computation of $Pred$, which keeps track only of the program points pc of the branching commands instead of all the visited method program points, decreasing the space complexity. However, in such work the proofs are done manually and the semantics being considered is *total* in contrast with the *partial* semantics used in our work; this could make the formal proof in Coq much more difficult.

The certification of our analysis was done by formalising the correctness proof in the proof assistant Coq. Mechanical verification of Java analysers have so far mainly dealt with the Java byte code verifier [2, 9, 4]. The first exception is the work reported in [3] on formalising an interprocedural data flow analyser for Java Card, on which part of the formalisation of the present analysis is based. The framework proposed in [3] allows us to concentrate on the specification of the analysis as a set of constraints and on the correctness of this system with respect to the semantics of the language (see Section 4). The lattice library and the generic solver of [3] were reused *as is* to extract the certified analyser.

8 Conclusion

We have presented a constraint-based analysis for detecting unbounded memory consumption on embedded devices such as Java Card smart cards. The analysis has been proved correct with respect to an operational semantics of Java byte code and the proof has been entirely formalised in the theorem prover Coq, providing the first certified memory usage analysis. The analysis can be used in program processing tools for verifying that certain resource-aware programming styles have been followed. An important contribution of the paper is to demonstrate how such an analysis can be formalised entirely inside a theorem prover. To the best of our knowledge, this is the first time that a resource usage analysis has undergone a complete formalisation with machine-checkable correctness proof. Still, several aspects of the analysis merit further development:

- By using the formula established in Lemma 3, we could in principle compute an over-approximation of the number of `new` instructions performed during any execution of the program and thereby produce an estimation of the memory usage. However, it is unclear whether this algorithm can be expressed in the constraint-based formalism used here; a specific proof effort would be required for this extension.
- From a programming language perspective, it would be interesting to investigate how additional restrictions on the programming discipline could be used to lower the complexity of the analysis, in the style of what was used in [10]. For example, knowing that the byte code is a result of a compilation of Java source code immediately gives additional information about the structure of the control flow graph.

– A challenge in the smart card setting would be to refine the algorithm to an implementation of a certified on-device analyser that could form part of an enhanced byte code verifier for protecting the device against resource-consumption attacks. The main challenge here is to optimise the memory usage of the analysis which is currently too high. Recent work on verification of C code in Coq [6] could be of essential use here. Techniques for an actual implementation can be gleaned from [10] as well as from [14] in order to optimise the computation of $Pred$.

References

1. David Aspinall and Andrea Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31(3–4):261–302, 2003.
2. Gilles Barthe, Guillaume Dufay, Line Jakubiac, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proc. ESOP'01*. Springer LNCS vol. 2028, 2001.
3. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proc. ESOP'04*, number 2986 in Springer LNCS, pages 385–400, 2004.
4. Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *Proc. of IEEE Int. Conference on Dependable Systems & Networks (DSN)*, 2002.
5. Karl Crary and Stephanie Weirich. Resource bound certification. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL'00)*, pages 184–198. ACM Press, 2000.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Proc. ICFEM 2004*, number 3308 in Springer LNCS, pages 15–29, 2004.
7. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
8. Martin Hofmann and Stefan Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of 30th ACM Symp. on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.
9. Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.
10. Xavier Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Smart card programming and security, (E-Smart 2001)*, pages 150–164. Springer LNCS vol. 2140, 2001.
11. Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, Trusted Logic SA, May 2001.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
13. David Pichardie. Coq sources of the development. `http://www.irisa.fr/lande/pichardie/MemoryUsage/`.
14. Gerardo Schneider. A constraint-based algorithm for analysing memory usage on Java cards. Technical Report RR-5440, INRIA, December 2004.
15. Igor Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Algebraic Programming*, 58(1-2), 2004.