Memory Usage Estimation for Java Cards ⋆

Gerardo Schneider⋆⋆

IRISA/CNRS, Campus de Beaulieu F-35042 Rennes, France
{gerardo@irisa.fr}

1 Introduction

Embedded systems, i.e. special-purpose computer systems built into larger devices, are widespread and can be found in simple gadgets like coffee machines and in more complex ones like mobile phones and smart cards. 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 applet could be installed in smart cards destroying data, disclosing private information over the network or performing non-authorised credit card transactions.

The massive use of such small communicating devices is imminent. Hence, it is essential that their platforms, as well as the applets that will run on them, can provide some minimal security guarantee 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 a crucial issue, mainly due to their limitation on size.

Concerning memory management on Java smart cards, we quote Chen: "Because memory in a smart card is very scarce, neither persistent nor transient objects should be created willy-nilly" [3, Section 4.4]. Chapter 13 of the same book provides programming tips on how to optimise applet creation considering the memory limitations. Here are some of the recommendations for applet programmers related to our work: "You should also limit nested method invocations, which could easily lead to stack overflow. In particular, applets should not use recursive calls." ([3, Section 13.3]). Later, on Section 13.4, it says: "An applet should always check that an object is created only once". Even though the above tips are followed by industry when programming applets and are in general respected, mistakes regarding memory usage –like the creation of an object inside a loop– both accidentally or intentionally, may have non-desirable consequences. In fact, there is nothing in the standards which prevents a(n) (intentionally) badly written applet to allocate all persistent memory on a card. Hence, the detection of recursive methods and loops in byte codes for embedded systems (smart cards in particular) is a crucial issue to take into account.

As far as we are aware there is no available tool for detecting the dynamic creation of objects inside cycles and/or recursive functions for Java smart cards. The byte code verifier [4], for instance, does not verify properties related to memory usage. There are, undoubtedly, many ways of writing algorithms for detecting syntactic loops, mutually recursive methods and over-approximating the dynamic memory used for an assembler-like language. In this work, however, we propose a constraint-based algorithm, presented as a set of rules –one for each instruction– for solving the above-mentioned memory-related concerns. The formalism chosen was not governed by mere aesthetic reasons: it is compositional and it allows to obtain a certified analyser using the

⋆ Presented at NWPT'2004 (The 16th Nordic Workshop on Programming Theory). Uppsala (Sweden), October 6–8, 2004.

⋆⋆ Partially supported by the *RNTL* French project, CASTLES (*Conception d'Analyses Statiques et de Tests pour le Logiciel Embarqué Sécurisé*).

program-as-proof paradigm (i.e., following the Curry-Howard isomorphism) to extract a program directly from the proof of its correctness. One feature of our algorithm is that it works directly on the byte code and there is no need to build extra data structures (e.g., a control-flow graph). Moreover, the byte code considered is arbitrary, i.e. we do not have any assumption about the byte code being produced by a "good" compiler. Our approach includes both intra- and inter-procedural analysis. Furthermore, the formalism used is compatible with existing works done at IRISA [2], which will allow to reuse the constraint solver, the fix-point algorithm and some of the lattice libraries "implemented" in Coq [7].

The language being considered is JCVMLe [8], a *byte code* language that models the Java Card Virtual Machine Language. It manipulates (dynamic) objects and arrays and besides the usual stack and register operations it comprises interesting features like virtual method calls, (mutually) recursive methods, (un)conditional jumps and subroutines. We assume there is no garbage collector, which is the case for Java Card upto version 2.1. 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, not during execution.

Our technique lies within the scope of *static analysis* [6], which analyses some run-time properties of a program without executing it. The implementation of static analysis is generally based on the solution of constraints through the computation of fix-points.

2 A constraint-based algorithm

Given a program P, we associate a set of constraints to each program instruction (applying rules like the ones shown in Fig. 1). Essentially, our algorithm add program points (m, pc) to a certain context Γ for each instruction (m, pc) : `new` o, if it is not in a loop nor in a mutually recursive method. If such instruction is inside a cycle and/or a (mutually) recursive method, then $<!>_{(m,pc)}$ is added to Γ instead.

The least solution to the set of constraints associated to P is obtained by computing the least fix-point of a certain functional F arising from such set. By a corollary of Tarski's Fixed Point theorem, the solution may be obtained as the limit of the stabilising sequence $(F^n(\bot))_{n\in\mathbb{N}}$.

Because of lack of space we only present in this abstract some of the rules of our main algorithm and we explain only the first rule (see Fig. 1). If the current instruction at program counter pc in method m is `new` o, and the instruction is inside a loop (detected by the predicate *Loop*) or a (mutually) recursive method (predicate *Rec*) then the rule propagates the *warning* type $<!>_{(m,pc)}$ to $\Gamma(m, pc + 1)$. Thus, at program point $(m, pc + 1)$, Γ will contain at least everything it had at (m, pc), plus $<!>_{(m,pc)}$. The content of Γ is thus "propagated" by the constraints associated to each instruction, as defined by each rule. The rules for `invokevirtual` and subroutine calls are similar to the `invokedefinite`; for any other instruction, $\Gamma(m, pc) \sqsubseteq \Gamma(m, pc + 1)$.

We have defined, in a similar way, rules for computing the predicate *Loop* –which detects intra-procedural cycles and instructions reachable from such cycles through method calls– and *Rec*, which computes all the mutually recursive methods. In fact, the technical difficulties are mainly presented in the computation of these last two predicates. In particular, detecting loops in a byte code program is more difficult than in a high-level language like Java, mainly due to the lack of structure, allowing jumps from (and to) the middle of a cycle.

3 Final Discussion

Besides the advantages mentioned in the introduction, the modularity of our algorithm allows the analyser itself, as well as the predicates *Loop* and *Rec*, to be reused by other constraint-based analysers as predicates (programs) which have been proved correct.

$$\frac{(m,pc) : \texttt{new}\ o \qquad Loop(m,pc) \lor Rec(m,pc)}{\Gamma(m,pc) \cup \{<!>_{(m,pc)}\} \sqsubseteq \Gamma(m,pc+1)} \qquad \frac{(m,pc) : \texttt{new}\ o \qquad \neg Loop(m,pc) \land \neg Rec(m,pc)}{\Gamma(m,pc) \cup \{(m,pc)\} \sqsubseteq \Gamma(m,pc+1)}$$

$$\frac{(m,pc) : \texttt{if}\ c\ \texttt{goto}\ pc'}{\begin{array}{l}\Gamma(m,pc) \sqsubseteq \Gamma(m,pc+1)\\ \Gamma(m,pc) \sqsubseteq \Gamma(m,pc')\end{array}} \qquad \frac{(m,pc) : \texttt{invokedefinite}\ m'}{\begin{array}{l}\Gamma(m,pc) \sqsubseteq \Gamma(m',1)\\ \Gamma(m',\mathrm{END}_{m'}) \sqsubseteq \Gamma(m,pc+1)\end{array}}$$

$$\frac{(m,pc) : \texttt{return}}{\Gamma(m,pc) \sqsubseteq \Gamma(m,\mathrm{END}_m)} \qquad \frac{(m,pc) : \texttt{goto}\ pc'}{\Gamma(m,pc) \sqsubseteq \Gamma(m,pc')}$$

Fig. 1. Rules of the main algorithm

Current Work. We have proved the termination of the main algorithm and of the two predicates $Loop$ and Rec. To prove termination, we have defined (finite) lattices defined on the range of the program counter numbers and method names. We are currently working on a hand-written proof of soundness w.r.t. to an abstraction of the operational semantics.

Future Work. Our approach allows the rules to be fed into the proof assistant Coq [1]. We intend to prove the correctness of our algorithm in the constructive logic of Coq and to use its extraction mechanism to automatically obtain a certified analyser. Besides its application on smart cards, we believe we can apply our analyser to mobile phone technology, where the detection of loops and recursive methods in applets is crucial, to avoid for instance the saturation of communication by sending SMS' indiscriminately.

Related Work. Our approach is inspired by the work in [2] where the technique sketched here has been applied with success for extracting a data flow analyser for Java card byte code programs. Even though the motivations and the techniques are not the same, it is worth mentioning the work done in the Mobile Resource Guarantees project [5], which applies ideas from proof-carrying code for solving the problem of resource certification for mobile code,

References

1. 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.
2. D. Cachera, T. JeMobile Resource Guarantees projectnsen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *ESOP*, LNCS, pages 385–400, 2004.
3. Zhiqun Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison Wesley, 2000.
4. Xavier Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.
5. MRG. Mobile Resource Guarantees project. See `http://groups.inf.ed.ac.uk/mrg/`.
6. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.
7. David Pichardie. Coq sources of the development corresponding to the paper [2]. See `http://www.irisa.fr/lande/pichardie/CarmelCoq/`.
8. I. Siveroni and C. Hankin. A proposal for the JCVMLe operational semantics. Research report SECSAFE-ICSTM-001-2.2, October 2001.