Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 217–233
www.elsevier.com/locate/entcs

Guaranteeing Correctness Properties of a Java Card Applet

Lars-Åke Fredlund

Box 1263, 164 29 Kista,
Swedish Institute of Computer Science, Sweden
e-mail: fred@sics.se

Abstract

The paper describes an experiment in which a framework for model checking Java byte code, combined with the application of runtime monitoring techniques through code rewriting, was used to guarantee correctness properties of a Java Card applet.

Keywords: Java byte code, runtime monitoring, code rewriting, Java Card.

1 Introduction

The Java Card platform [8] is a platform for building multi-application smart cards. It is based on a subset of Java which omits features such as concurrency through threads, garbage collection, and many API functions. However, to support multiple applications co-existing on the same card (e.g., both a purse applet and a loyalty applet), there is a notion of an applet. Java Card applets are implemented by extending the Java Card API class `javacard.framework.Applet`. Briefly an implementation is required to provide a method `install` which is called upon installation of an applet, methods `select` and `deselect` for selection/deselection of a particular applet on a card, and the "main" method `process` which is called by the card runtime environment (operating system) upon receiving an event from the card environment intended for that applet. An applet can also implement a method `getShareableInterfaceObject` for permitting other applets on the same card to call it.

doi:10.1016/j.entcs.2004.01.033

Unfortunately the Java Card programming platform provides weak support for separating applets. For instance nothing in the standards prevents a malicious or badly written applet from allocating all persistent memory on a card (the little there is), and since the standard does not require garbage collection this is a very undesirable state-of-affairs. Similar concerns exists for inter-applet calls, although they are controlled by a rather weak firewall mechanism. Thus there are significant dangers with permitting new applets onto a functioning smart card, and as a result one of the chief innovations of Java Card, i.e., multiple applications co-existing on the same card, is in practise not used much at all.

To improve upon this situation the formal design techniques group of SICS have been using fully automatic and low-cost (in terms of execution speed and memory usage) verification methods that could, potentially, be used by an on (or off) card runtime system to determine at load time whether a new applet should be permitted onto a card with pre-existing applets or not. In a first experiment, reported in [3], we analysed inter-method calls of multi-applet Java Card smart cards using model checking of Java byte code. In this paper we extend the treatment to memory allocation concerns. In case the safety of an applet cannot be proved using model checking, we as a complementary technique instrument a compiled applet with a runtime monitor to guarantee that it adheres to a safe memory allocation policy.

To provide a semantics foundation for the analysis of Java Card applets we use the abstract notion of a program graph, capturing the control flow of programs with procedures/methods, and which can be efficiently computed. The behaviour of such program graphs is defined through the notion of pushdown systems, which provide a natural execution model for programs with methods (and possibly recursion), and for which completely automatic model checkers for LTL exist, e.g., Moped [7]. The details of the translation are elaborated in section 3.1, and sections 3.2, 3.3 and 3.4 describes the logic and our use of the Moped model checking tool in further detail.

The example considered in the paper is a real applet submitted by Schlumberger. The applet is monolithic, and does not communicate with other applets. In section 4 we formalise and attempt to verify the property that no memory is allocated by an applet (after a personalization process) using model checking. As the satisfaction of this property is shown to depend critically on properties of data, which requires reasoning using less coarse abstractions than the ones implemented in the call graph extraction tool, we consider in section 5 the complementary use of runtime monitoring techniques to guarantee the property.

2 Constructing Method Call Graphs

We use an external static analysis tool, Soot [12], adapted to Java Card [1], to generate call graphs which abstract from everything (such as data variables, and parameters to method calls) but the presence and order of method calls inside method bodies. The analysis tool performs a safe over-approximation (with regards to preservation of LTL safety properties) in the sense that call edges may be present in the result call graph even if the corresponding calls cannot be invoked at runtime, but the opposite does not hold. For instance, when the static analysis cannot determine which class method is invoked in a method call, typically due to subtyping, then a call edge is generated to a target method in every possible class, thus increasing the nondeterminism in the generated call graph. The static analysis tool generates graphs with information about exceptional behaviours. In this work exceptional edges, and nodes, are translated into non-deterministic constructs thus effectively increasing the non-determinism in program behaviour in a conservative fashion.

The call graph generation is also conservative with respect to the Java Card firewall mechanism, which is not considered during static analysis. That is, a method call that at runtime will fail the security checks of the Java Card runtime environment will nevertheless invariably be included in the method call graphs. To refine the analysis, and to permit analysis of Java Card API usage, the API classes of SUN's Java Card Development Kit (version 2.1.2) are optionally included in the method call generation.

The result of the generation process is a set of method call graphs, representing the methods that may be called after the runtime environment invokes one of the (public) callable applet methods: `install` which is called during installation of an applet, methods `select` and `deselect` for applet selection/deselection, `getShareableInterfaceObject` for permitting inter-applet calls, and the main processing method `process` which is invoked once for every (user) interaction with a Java Card applet.

As we in this case study want to observe the allocation of memory by an applet the standard method call graph generation process has been augmented to additional include information about invocations of the `new` and `newarray` Java virtual machine (byte code) instructions. An instance of the `new` instruction will be represented in a method call graph as a call to the new (synthetic) method `Events.newInst` and a `newarray` instruction as a call to the method `Events.newarrayInst`.

[1] to handle, for instance, the absence of the `java.lang.Class` class

2.0.1 Method Call Graphs

The methods M are partitioned into classes C, which are themselves partitioned into packages P. We assume the usual Java naming conventions with fully qualified names, i.e., a class has a name *Package.identifier* and a method has a name *Class.identifier*.

Definition 2.1 [Method Graph, adapted from [9]] A *method graph* is a tuple

$$m \triangleq (V_m, \rightarrow_m, \lambda_m, \mu_m)$$

such that:

(i) V_m are the *program points* of m,

(ii) $\rightarrow_m \subseteq V_m \times V_m$ are the *transfer edges* of m, and

(iii) $\lambda_m : V_m \rightarrow T$ designates to each program point of m a *program point type* from the set $T \triangleq \{\mathsf{entry}, \mathsf{seq}, \mathsf{call}, \mathsf{return}\}$.

(iv) $\mu_m : V_m \rightarrow \wp(M)$ designates to each program point of type call of m a non-empty set of methods.

We assume the program point sets V_m to be pairwise disjoint. The program points of the program is the set $V \triangleq \bigcup_{m \in M} V_m$.

The program point type indicates whether (`entry`) a node is the entry point of a method, (`seq`) a node in which no method call or return takes place, (`call`) a node from which a method call takes place, or (`return`) a node in which the execution of the method finishes and control flow returns to the calling method.

The method call graphs extracted from a Java Card applet using the modified Soot tool has the following invariant properties: (i) there is exactly one entry program point per method; (ii) there is exactly one return program point per method; (iii) nondeterminism in a call node is due to lack of precision in resolving the target of a method call (due to subtyping), never due to the occurrence of two distinct calls (sequential method calls are always separated by a transfer edge).

For convenience, we introduce the predicates

$$\begin{aligned} v : t &\triangleq \lambda_m(v) = t \text{ for } t \in T \\ v : \mathsf{loc}\, m &\triangleq v \in V_m \\ v : \mathsf{entry}\ m &\triangleq v : \mathsf{entry} \ \wedge\ v : \mathsf{loc}\, m \\ v : \mathsf{return}\ m &\triangleq v : \mathsf{return} \ \wedge\ v : \mathsf{loc}\, m \\ v : \mathsf{class}\ c &\triangleq \exists m.\ v : \mathsf{loc}\, m \ \wedge\ m \in c \\ v : \mathsf{package}\ p &\triangleq \exists c.\ v : \mathsf{class}\ c \ \wedge\ c \in p \end{aligned}$$

For the sake of this case study we further define a predicate v : api, which holds if the program point v occurs in a method in a Java Card API package (one of `java.lang`, `javacard.framework`, `visa.openplatform`, `javacard.security` or `javacardx.crypto`).

3 Model Checking Method Call Graphs

3.1 Pushdown Systems

Pushdown systems provide a natural execution model for programs with recursion. They form a well-studied class of infinite-state systems for which many important problems like equivalence checking and model checking are decidable [2].

Definition 3.1 [PDS, from [6]] A *pushdown system (PDS)* is a tuple

$$\mathcal{P} \ \triangleq\ (P, \Gamma, \Delta)$$

where:

(i) P is a finite set of *control locations*;

(ii) Γ is a finite set of *stack symbols*;

(iii) $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^\star)$ is a finite set of *rewrite rules* $\langle p, \gamma\rangle \rightarrow \langle q, \sigma\rangle$.

The set $P \times \Gamma^\star$ are the *configurations* of $\mathcal{P}$. If $\langle p, \gamma\rangle \rightarrow \langle q, \sigma\rangle$ is a rewrite rule of $\mathcal{P}$, then for each $\omega \in \Gamma^\star$ the configuration $\langle q, \sigma \cdot \omega\rangle$ is an *immediate successor* of the configuration $\langle p, \gamma \cdot \omega\rangle$. A *run* of $\mathcal{P}$ is a sequence $\rho = \langle p_0, \sigma_0\rangle \langle p_1, \sigma_1\rangle \langle p_2, \sigma_2\rangle \cdots$, such that for all i, $\langle p_{i+1}, \sigma_{i+1}\rangle$ is an immediate successor of $\langle p_i, \sigma_i\rangle$.

We now define how a set of methods M induces a PDS.

Definition 3.2 [Induced PDS, formalising [7]] A set of methods M *induces* a PDS

$$\mathcal{P} \stackrel{\Delta}{=} (P, \Gamma, \Delta)$$

as follows:

(i) P consists of the single control location p;

(ii) Γ is the set V of program points;

(iii) Δ is the set $\bigcup_{m \in M} \bigcup_{v \in V_m} \mathrm{Prod}(v)$, where $\mathrm{Prod}(v)$ is a set of rewrite rules:

$$\begin{cases} \{\langle p, v\rangle \rightarrow \langle p, \epsilon\rangle\} & \text{if } v : \mathsf{return} \\ \{\langle p, v\rangle \rightarrow \langle p, v'\rangle \mid v \rightarrow_m v'\} & \text{if } v : \mathsf{entry} \\ & \text{or } v : \mathsf{seq} \\ \bigcup_{m' \in \mu_m(v)} \left\{ \langle p, v\rangle \rightarrow \langle p, v' \cdot v''\rangle \mid v' : \mathsf{entry}\ m', v \rightarrow_m v'' \right\} & \text{if } v : \mathsf{call} \end{cases}$$

The rewrite rules of the pushdown system can be interpreted as simply manipulating the calling stack of the program from which the PDS was obtained. Given a configuration $c \equiv \langle p, v \cdot \sigma\rangle$ let $\mathsf{point}\ (c) \stackrel{\Delta}{=} v$.

3.2 Specification Language

Our specification language is linear temporal logic (LTL), with program point predicates p as atomic propositions but omitting the type predicate $v : t$. The choice of linear temporal logic as the specification language, instead of for instance the modal μ-calculus for which the model checking problem for our encoding into pushdown systems is also efficiently decidable, was solely motivated by the existence of the efficient model checker Moped [7] for LTL.

The operators of the logic are the standard ones. If ϕ and ψ are formulas then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\mathcal{X}\ \phi$ and $\phi\ \ \mathcal{U}\ \ \psi$. The meaning of formulas is defined with respect to runs of infinite length $r \equiv c_0 c_1 c_2 \ldots$. We let r_i denote the suffix of r starting in configuration c_i. Then satisfaction $r \models \phi$ of a formula ϕ by a run r is defined as:

$$
\begin{array}{lll}
r \models p & \text{iff} & \mathsf{point}\ (c_0) : p \\
r \models \neg\phi & \text{iff} & \text{not } r \models \phi \\
r \models \phi \wedge \psi & \text{iff} & r \models \phi \text{ and } r \models \psi \\
r \models \phi \vee \psi & \text{iff} & r \models \phi \text{ or } r \models \psi \\
r \models \mathcal{X}\,\phi & \text{iff} & r_1 \models \phi \\
r \models \phi\ \mathcal{U}\ \psi & \text{iff} & \text{there is an } i \geq 0 \text{ such that } r_i \models \psi \text{ and } r_j \models \phi \text{ for all } 0 \leq j < i
\end{array}
$$

Henceforth let false abbreviate $p \wedge \neg p$ for some atomic predicate p, true abbreviate $\neg\mathsf{false}$, $\phi \Rightarrow \psi$ abbreviate $\neg\phi \vee \psi$, and $\mathsf{next}\ \phi$ abbreviate $\mathcal{X}\,\phi$ and $\phi\ \mathsf{until}\ \psi$ abbreviate $\phi\ \mathcal{U}\ \psi$. Further define $\mathsf{eventually}\ \phi \stackrel{\Delta}{=} \mathsf{true}\ \mathcal{U}\ \phi$ and $\mathsf{always}\ \phi \stackrel{\Delta}{=} \neg(\mathsf{eventually}\ \neg\phi)$. The weak until operator $\phi\ \mathsf{weakuntil}\ \psi$ abbreviates $\phi\ \mathsf{until}\ \psi \vee \mathsf{always}\ \phi$. Finally let $\mathsf{never}\ \phi \stackrel{\Delta}{=} \mathsf{always}\ \neg\phi$.

Given a PDS *pds* let the notation $m \vdash \phi$ express the judgment that all runs starting in the entry program point of the method m satisfy ϕ. More formally:

Definition 3.3 [Model Checking a Method Call] Given a PDS *pds* with the single control location p and a method m, the judgment $m \vdash \phi$ is valid iff for every run r of the PDS *pds′* from the initial configuration $\langle p, m\_\texttt{init}\rangle$, $r \models \phi$ holds, where v is the entry program point of method m (i.e. $v : \mathsf{entry}\ m$), *pds′* is the PDS *pds* extended with the fresh stack symbols $m\_\texttt{init}$ and $m\_\texttt{loop}$, and the two rewrite rules $\langle p, m\_\texttt{init}\rangle \rightarrow \langle p, v \cdot m\_\texttt{loop}\rangle$ and $\langle p, m\_\texttt{loop}\rangle \rightarrow \langle p, m\_\texttt{loop}\rangle$ to achieve infinite runs.

The definition of a judgment $m \vdash \phi$ is motivated by the Moped tool which implements an algorithm for checking an initial configuration with a stack of size at most one against an LTL formula.

3.3 Specification Patterns

As in the Bandera project [4] specification patterns are used to facilitate formulating correctness properties. These specification patterns concern temporal properties of method invocations, and are either *temporal patterns* or *judgment patterns* concerning the invocation of a particular method. Below a set of patterns that we have defined, and which are commonly used, are given.

To express that *within the call of a method m the property ϕ holds* the judgment pattern

$$\mathsf{Within}\ m\ \phi \stackrel{\Delta}{=} m \vdash \phi$$

is used. The property that *a call to m_1 never triggers method m_2* is:

$$m_1 \text{ never triggers } m_2 \;\equiv\; \mathsf{Within}\ m_1\ (\mathsf{never\ loc}\, m_2)$$

Next define the temporal patterns (formulas) (i) m_2 **after** m_1, i.e., m_2 can only be called after a call to m_1; (ii) m_2 **through** m_1, i.e., m_2 can only be called from m_1; (iii) m_2 **from** m_1, i.e., m_2 can only be called directly from m_1; and (iv) m_1 **excludes** m_1, i.e., when m_1 is called this excludes the possibility that m_2 will later be called:

$$m_2 \text{ after } m_1 \quad \stackrel{\Delta}{=} (\mathsf{never\ loc}\, m_2)\ \mathsf{weakuntil\ loc}\, m_1$$

$$m_1 \text{ excludes } m_2 \stackrel{\Delta}{=} (\mathsf{eventually\ loc}\, m_1) \;\Rightarrow\; \mathsf{never\ loc}\, m_2$$

$$m_2 \text{ from } m_1 \quad \stackrel{\Delta}{=} \mathsf{always}\ (\neg\,(\mathsf{loc}\, m_1 \vee \mathsf{loc}\, m_2) \Rightarrow \mathsf{next}\ \neg\mathsf{loc}\, m_2) \;\wedge\; \neg\mathsf{loc}\, m_2$$

$$m_2 \text{ through } m_1 \stackrel{\Delta}{=} \begin{array}{l} \neg\mathsf{loc}\, m_2\ \mathsf{weakuntil\ loc}\, m_1 \\ \wedge \left(\begin{array}{l} \mathsf{always\ return}\ m_1 \Rightarrow \\ \quad \mathsf{next}\ (\neg\mathsf{loc}\, m_2\ \mathsf{weakuntil\ loc}\, m_1) \end{array} \right) \end{array}$$

The intuitive idea of the formulation of m_2 **from** m_1 is to express that the current program point can be in method m_2 only because of a direct call from m_1, or because it was already in m_2, and initially the program point is not in m_2. The above patterns can be combined with the **Within** pattern. For example,

$$\mathsf{Within}\ m_1\ (m_3\ \mathsf{after}\ m_2)$$

expresses that during a call to m_1 the method m_3 will be called only after calling m_2.

An alternative technique for expressing correctness properties of behaviours of programs of stack-based languages is to use stack inspection techniques [9]. Essentially these techniques express constraints on the set of all possible run-time stacks. Note however that for instance the **after** property above cannot directly be coded as a stack inspection property since the calls to m_1 and m_2 need not be concurrent.

3.4 A Tool for Model Checking Pushdown Systems

The Moped tool [7] can check a pushdown system, from an initial configuration, against an LTL formula where the atomic predicates consists of a set of atomic symbols that checks the identity of the top stack symbol or the control location (i.e., simply checks name equality). In case the LTL formula is

falsified a reduced pushdown system constructed from the original one, that also falsifies the LTL formula, is presented as diagnostic information.

To represent the non-identity atomic predicates (e.g., package, entry, ...) as "Moped LTL formulas" a number of options are possible. Consider for instance the package atomic predicate. A direct representation of the predicate in Moped LTL would consist of a disjunction over all the program points in any class in the package.

An alternative representation strategy is to enrich the translation from a call graph to a pushdown system. Since Moped provides boolean variables we could represent the current package identity encoded in a set of boolean variables in the pushdown system. These variables would then be updated for every rewrite rule that crosses package boundaries. Finally the representation of the package predicate itself would consist of a simple boolean condition.

We have instead opted to extend the Moped tool with atomic predicates that can match a control location, or the top stack symbol, against a regular expression. These predicates check the syntactic shape of the symbol being tested. Consider the naming of program points of a method m by the call graph construction. Its entry program point will be named m_entry, its (unique) return program point will be named m_exit, and all other program points in m are of the form m_n where n is a natural number.

With these conventions in place the atomic predicates can be represented in "regular expression Moped" as indicated below:

$$\begin{aligned}
\mathsf{loc}\ m &\triangleq m_.* \\
\mathsf{entry}\ m &\triangleq m_\texttt{entry} \\
\mathsf{return}\ m &\triangleq m_\texttt{exit} \\
\mathsf{class}\ c &\triangleq c\backslash..*_.* \\
\mathsf{package}\ p &\triangleq p\backslash..*\backslash..*_.*
\end{aligned}$$

In the encoding it is assumed that the dot symbol '.' has to be quoted using a backslash character inside a regular expression to represent itself, rather than representing any character. Wildcards can be used in a regular expression to achieve a limited form of quantification over program points.

4 The SLB example: Code and Correctness Properties

This paper studies an applet developed by SchlumbergerSema, henceforth referred to as the SLB applet study. In total the source code of the applet

comprises around 765 lines of Java Card source code. As an example benchmark figure the generation of the calling graph for the `process` methods of the applet, and translation of the callgraph to a push down system, takes roughly 3 seconds [2] . The resulting push down system has approximately 540 rules.

As an example we first encode the property of absence of communication with other applets. A sufficient condition (but not strictly necessary) for establishing that the SLB applet does not attempt to communicate with another applet [3] is to check whether it ever calls a method in another package. The property that verifies that it does not can be stated as (note that the applet package is named `com.schlumbergersema.slb` and that API calls and calls to the synthetic methods in `Events` are permitted):

$$\begin{array}{l}\mathsf{Within}\ \texttt{com.schlumbergersema.slb.Main.process}\\ \quad \mathsf{always}\ \ (\mathsf{package}\ \texttt{com.schlumbergersema.slb}) \vee \mathsf{api} \vee \mathsf{package}\ \texttt{Events}\end{array}$$

Similarly a sufficient condition for the absence of incoming calls to the applet is that the applet does not define a `getShareableInterfaceObject` method, which is checkable when the API is abstracted away.

In an accompanying document [10] a set of desirable security properties for the SLB applet were given, including information flow control properties, memory allocation control properties, error prediction properties, and consistency control properties. In this paper we mainly focus on the problem of checking and guaranteeing that proper memory allocation security procedures are followed. The document specifies, that,

The SLB applet does not allocate memory after the personalization process.

Unfortunately the document does not specify what constitutes a completed personalization process. In the first approximation of the correctness property we consider personalization to be completed upon the first call of the `process` method (this method can only be called after the completion of a call to the `install` method). The corresponding property is:

$$\begin{array}{l}\mathsf{Within}\ \texttt{com.schlumbergersema.slb.Main.process}\\ \quad \mathsf{never}\ \ (\mathsf{loc}\,\texttt{Events.newInst}) \vee (\mathsf{loc}\,\texttt{Events.newArrayInst})\end{array}$$

That is, during the call of the `process` method no allocation of memory will be attempted using the byte code instructions `new` or `newarray`. The Moped

[2] on a laptop with an Intel Pentium III Mobile CPU at 1200Mhz

[3] except ones that are defined in the same Java package, as the Java Card standard communication firewall anyway only regulates inter-package method calls

```
p <com_schlumbergersema_slb_Main_process_init>
p <com_schlumbergersema_slb_Main_process_entry m_loop>
p <com_schlumbergersema_slb_Main_process_77 m_loop>
p <com_schlumbergersema_slb_Main_process_78 m_loop>
p <javacard_framework_APDU_getBuffer_entry
   com_schlumbergersema_slb_Main_process_81 m_loop>
p <com_schlumbergersema_slb_Main_process_81 m_loop>
...
p <com_schlumbergersema_slb_Main_processAppendRecord_292
   com_schlumbergersema_slb_Main_process_ret m_loop>
p <com_schlumbergersema_slb_Main_processAppendRecord_293
   com_schlumbergersema_slb_Main_process_ret m_loop>
p <com_schlumbergersema_slb_Main_processAppendRecord_294
   com_schlumbergersema_slb_Main_process_ret m_loop>
p <Events_newArrayInst_entry
   com_schlumbergersema_slb_Main_processAppendRecord_295
   com_schlumbergersema_slb_Main_process_ret m_loop>
...
```

Fig. 1. An excerpt from the counter example

tool can quickly [4] check such a formula; for the SLB applet the result is that the formula does not hold, and a counter example is generated automatically (a reduced system generated from the original one that also fails to satisfy the property). As an example we include the printout in figure 1 which shows the evolution of configurations (a pair of a control location `p` and a stack of symbols) during a call to the `process` method. The names of methods have been mangled, but still the figure does illustrate the problem: a call to the `process` method can, via a call to the `processAppendRecord` method, allocate memory using the `newarray` byte code instruction (note the occurrence of the entry point of `Events.newarrayInst` on top of the "stack" in the final configuration). The corresponding property for allocation using the `new` instruction does hold, i.e., no memory is allocated for non-array objects after calling the `process` method.

Inspecting the byte code of the applet shows that the open platform [5] personalization scheme seems to be used, i.e., the code contains a call to the method `visa.openplatform.OPSystem.setCardContentState`. To check whether allocation occur only before personalization the first property is refined into the property:

$$\text{Within } \texttt{com.schlumbergersema.slb.Main.process}$$
$$\text{always} \left(\begin{array}{l} \text{loc } \texttt{visa.openplatform.OPSystem.setCardContentState} \Rightarrow \\ \\ \text{never loc } \texttt{Events.newArrayInst} \end{array} \right)$$

That is, during any call to `process` new arrays are never allocated after

[4] in less than a second, on the same hardware as was used to generate the call graph
[5] `http://www.globalplatform.com`

the personalization method has been called. The Moped tool confirms that this property holds of the SLB applet. Unfortunately it isn't quite strong enough to guarantee absence of allocation after personalization (since a later call to `process` is still allowed to allocate an array).

The property below expresses a stronger property that during any call to `process`, any array allocation must always finish with a call to the `setCardContentState` method:

$$\text{Within } \texttt{com.schlumbergersema.slb.Main.process}$$
$$\text{always} \left(\begin{array}{l} \text{loc } \texttt{Events.newArrayInst} \Rightarrow \text{eventually} \\ \quad \text{loc } \texttt{visa.openplatform.OPSystem.setCardContentState} \end{array} \right)$$

Trying to verify this property with Moped unfortunately fails and a counterexample is generated.

To summarise the state-of-affairs: during any call to `process` there is a possibility that personalization takes place, and if it does, no more memory is allocated. However, there exists also the possibility that an array is allocated during an invocation of the `process` method but that afterwards no personalization takes place. This need not necessarily indicate a bug; it could be that the applet keeps a state between invocations of the `process` method by the runtime environment which records whether personalization has taken place, i.e., the allocation property is data dependent and these data dependencies have been abstracted away during call graph generation thus generating false positives.

Clearly the call graph generation process can be refined, using static analysis techniques, to take data dependencies into account. However, determining which abstractions are required to conclusively prove, or disprove, the property is in general not a task which we can expect to be able to solve automatically. In the next section we instead explore an alternative technique to guarantee the memory allocation property.

5 Monitoring Memory Allocation

Clearly the memory allocation property is a safety property, and we can thus guarantee the desired memory allocation property by implementing a runtime monitor [11] to control the execution of the applet.

The monitor has an obligation to preserve the (monitored) safety property and to halt the execution of the (monitored) applet whenever it detects that the applet is about to violate its safety property. For checking the memory allocation control property a very simple two state runtime monitor suffices

that keeps track of whether personalization has occurred, and if it has not, permits memory allocations, and if it has, disallows them. The applet combined with such a runtime monitor may not meet all requirements regarding behaviour (e.g., progress properties), but it will not violate its safety property.

5.1 Implementation Details

A monitor can be implemented in different ways: in a runtime system if it is accessible, or as a separate thread/process, or the code of the runtime monitor can be directly inlined with the monitored application. For the Java Card platform the choice of implementation method is obvious given the lack of threads, and the lack of access to API libraries. The applet code has to be physically combined with the runtime monitor code, using the technique of code instrumentation. Any operations in the applet code that could violate the monitor has to be preceded with monitor code that checks whether the operation in question is safe. Thus for the case study we monitor calls to the API method `visa.openplatform.OPSystem.setCardContentState` and we monitor memory allocation using the byte code instructions `new` and `newarray` from applet code (in this study we do not consider indirect memory allocations by Java Card API methods). As Java byte code is well-structured, and a byte code verifier guarantees that calling conventions and such are adhered to, it turns out that Java is very well suited for implementing runtime monitoring through code instrumentation; see for example Erlingsson and Schneider [5] for a discussion.

We have implemented a facility for experimenting with the automatic instrumentation of Java Card programs using the Soot [12] tool; in fact the same tool that was used for call graph extraction. A significant advantage of using the Soot tool is that it provides a well-defined high-level abstract view of Java (Card) byte code methods. A method is guaranteed to have single entry and return program points, the runtime stack is abstracted away in favour of assignments to local variables, to be composed of a select few instructions (rather than arbitrary Java byte code instructions) partly due to the absence of stack instructions, and for implementing program transformations there is ample support for inserting new byte code instructions in the middle of a method body.

To continue the SLB case study we implemented a generic transformer in Soot that when it recognizes a `new` instruction (or `newarray`), or a call to the personalization method, inserts an appropriate call to the runtime monitor. The code of the transformer (for the `new` instructions) is:

...

```
MonitorClass = Scene.v().loadClassAndSupport("Monitor");
personalize = MonitorClass.getMethod("void personalize()");
allocating = MonitorClass.getMethod("void allocating()");
...
protected void internalTransform
               (Body body, String phaseName, Map options)
{
  Chain units = body.getUnits();
  ...
  Iterator stmtIt = units.snapshotIterator();
  while(stmtIt.hasNext()) {
    Stmt s = (Stmt) stmtIt.next();
    ...
    else if (containsNewExpr(s)) {
     // We found a new expression, instrument it!
      units.insertBefore
      (Jimple.v().newStaticInvokeExpr(allocating), s);
    }
  }
}
...
boolean containsNewExpr(Stmt s) {
  if (s instanceof AssignStmt &&
   ((AssignStmt) s).getRightOp() instanceof NewExpr) return true;
  return false;
}
```

The monitor routine has a state variable, `personalized`, which determines whether allocation is permitted:

```
public class Monitor {
  private static boolean personalized = false;

  public static void personalize() {
    personalized = true;
  }
  public static void allocating() {
    if (personalized)
      ISOException.throwIt(ISO7816.SW_UNKNOWN);
  }
}
```

Next Java byte code was generated from the intermediate representation by Soot, generating ultimately a new applet that was combined with the monitor code. In the process of generating byte code from the intermediate representation a number of problems peculiar to the Java Card platform had to be resolved. In our experiment Soot was used as a Java compiler, and to be able to run the resulting (instrumented) Java Card applet (in, for instance, Sun's Java Card simulation tool) it is necessary to "convert it" [6] using SUN's Java Card Development Kit. Unfortunately the converter tool is targeted towards converting code produced by the Sun Java compiler. To produce byte code output acceptable to the converter tool we had to alter the Soot compiler to (i) generate code that Sun's converter would recognise as obeying the restrictions regarding the use of short integers, and (ii) to generate object initialization code that follows the rather harsh restrictions put forward in Java Card documentation.

Once these, and a number of minor additional problems with the compilation to Java Card were settled, we were able to simulate the instrumented SLB applet in Sun's Java Card simulator. Guided by an inspection of the source code of the applet we were able to design a sequence of APDU messages (events received from the card environment resulting in calls to the `process` method) which, had the safety preserving monitor not being inlined, would have resulted in the applet violating its memory allocation policy. That is, it would have attempted to allocate fresh memory even after personalization. The applet essentially allocates an array of records lazily, and signals personalization already when the first record has been allocated, even though later allocations can occur and it thus violates its own stated policy. In retrospect we believe it would have been exceedingly difficult to discover such a problem using solely a combination of model checking and automatic data abstraction.

The impact in terms of (instrumented) applet code size, of the in-memory size of the applet, and of execution slow-down is negligible. The only code additions, except the inclusion of the small monitor class itself, is the calls to the monitor class signalling personalization and memory allocation. However the number of locations where such calls occur are expected to be few.

5.2 A Second Memory Allocation Property

As a followup to the first experiment with runtime monitoring we decided to monitor also the weaker property of bounded memory allocation, i.e., that there is a bound on the amount of memory the applet allocates. To permit the monitoring of this property the memory control transformer was easily

[6] e.g., to check that only short integer arithmetic is used

extended to also count the amount of memory allocated (for a `new` or `newarray` instruction) by analysing the class hierarchies in Soot. That is, the fields of an object to be allocated are computed (including fields of superclasses), and the size of representing each field in the runtime system is estimated (the size is not predetermined by the Java standard). Thus a `new` instruction in the code was prefixed with a call to a monitor method, with as parameter the size of the allocation request (the size of the object, or the size of the array), and which is responsible for halting the execution of the applet if a predetermined allocation bound will be exceeded by the execution of the `new` instruction.

Clearly a property such as the bounded allocation property is a good example of a class of applet properties that can either be implemented in a card runtime system by a card manufacturer, or, as we show in this example, it can equally well be ensured by combining any applet code to be loaded onto a card with a well-defined runtime monitor guaranteeing the property. Other examples of such monitorable properties interesting for Java Card are, for instance, specific applet-to-applet communication disciplines that are stricter than the firewall mechanism provided by the Java Card standard.

6 Conclusions and Future Work

The paper has studied how to guarantee important security properties for a typical Java Card application using light-weight formal methods, e.g., using model checking and by implementing runtime monitors to forcibly ensure safety properties. The model checking work uses a framework for automatic model checking of temporal constraints on method calls in Java Card applets. The framework has been realised by combining a class–based static analysis tool with an automatic model checker for pushdown system and linear temporal logic. The runtime monitor experiment is promising but needs to be properly formalised. We would like to (i) develop a generic code rewriting function that given a safety property in temporal logic inserts probes into the code to keep the monitor state updated, and (ii) to use an available operational semantics for Java Card such as e.g. [1] to develop a proof that the code rewriting function respects the operational semantics of Java Card, and the semantics of the temporal logic. We would also like to study the feasibility of using code rewriting of applets as a systematic technique to implement card specific policies for card issuers.

Further information, including prototype implementations, regarding the work presented in this paper can be found in the web page `http://www.sics.se/fdt/projects/vericode/jcave.html`.

Acknowledgement

The research has been conducted within the VerifiCard project with financial support from the IST programme of the European Union. I would like to thank Gennady Chugunov and Dilian Gurov for their their contributions to this work, with regards to tool support and theory development for the model checking part, and thank Pablo Giambiagi for interesting discussions on the potential of runtime monitoring techniques.

References

[1] Barthe, G., G. Dufay, L. Jakubiec, S. M. de Sousa and B. Serpette, *A Formal Executable Semantics of the JavaCard Platform*, in: D. Sands, editor, *Proceedings of ESOP'01*, LNCS **2028** (2001), pp. 302–319.
URL ftp://ftp-sop.inria.fr/lemme/Gilles.Barthe/esop01.ps.gz

[2] Burkart, O., D. Caucal, F. Moller and B. Steffen, *Verification on infinite structures*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, North Holland, 2000 pp. 545–623.

[3] Chugunov, G., L. Fredlund and D. Gurov, *Model checking of multi-applet JavaCard applications*, in: *Fifth Smart Card Research and Advanced Application Conf. (CARDIS'2002)* (2002), pp. 87–96.

[4] Corbett, J., M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu and H. Zheng, *Bandera: extracting finite-state models from Java source code*, in: *International Conference on Software Engineering*, 2000, pp. 439–448.

[5] Erlingsson, U. and F. B. Schneider, *SASI enforcement of security policies: A retrospective*, in: *Proceedings of the New Security Paradigms Workshop*, 1999.

[6] Esparza, J., D. Hansel, P. Rossmanith and S. Schwoon, *Efficient algorithms for model checking pushdown systems*, in: *Proc. CAV'00*, Lecture Notes in Computer Science **1855** (2000), pp. 232–247.

[7] Esparza, J. and S. Schwoon, *A BDD-based model checker for recursive programs*, Lecture Notes in Computer Science **2102** (2001), pp. 324–336.

[8] *JavaCard 2.1.1 Documentation*, Technical report, Sun Microsystems (2000), http://java.sun.com/products/javacard/\discretionary{-}{}{}specs.html#211 .

[9] Jensen, T., D. L. Metayer and T. Thorn, *Verification of control flow based security properties*, in: *IEEE Symposium on Security and Privacy*, 1999.

[10] SchlumbergerSema, *Java Card applet security properties*, internal deliverable, VerifiCard Project, task 6.3.

[11] Schneider, F. B., *Enforceable security policies*, Technical Report TR99-1759, Cornell Univ (1999).

[12] Vallée-Rai, R., L. Hendren, V. Sundaresan, P. Lam, E. Gagnon and P. Co, *Soot - a Java optimization framework*, in: *Proceedings of CASCON 1999*, 1999.