INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Enforcing High-Level Security Properties For Applets

Mariela Pavlova — Gilles Barthe — Lilian Burdy — Marieke Huisman — Jean-Louis Lanet

N° 5061

December 2003

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Enforcing High-Level Security Properties For Applets

Mariela Pavlova , Gilles Barthe , Lilian Burdy , Marieke Huisman , Jean-Louis Lanet

Thème 2 — Génie logiciel
et calcul symbolique
Projet Everest

Rapport de recherche n° 5061 — December 2003 — 18 pages

Abstract: Smart card applications often handle privacy-sensitive information, and therefore must obey certain security policies. Such policies are usually described by high-level security properties, stating for example that no authentication must take place within a transaction.

Behavioural interface specification languages, such as JML (Java Modeling Language) have been successfully used to validate functional properties of smart card applications. However, high-level security properties cannot be expressed directly in such languages. Therefore, this paper proposes a method to translate high-level security properties into JML annotations. The method proceeds by synthesising appropriate annotations and weaving them throughout the application. In this way, security policies can be validated using the various existing tools for JML. The method is general and applies to a large class of security properties.

To illustrate its applicability, we applied the method to several realistic examples of smart card applications. This allowed us to find violations against the documented security policies for some of these applications.

Key-words: security, applets, Java Card, specification generation

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Vérification automatisée de politiques de sécurité pour des applications carte à puces

Résumé : Les applications destinées aux cartes à puce sont souvent employées pour réaliser des opérations sensibles du point de vue de la sécurité. En cela, elles doivent se conformer à des politiques de sécurité souvent décrites par des propriétés de haut niveau. De telles propriétés indiquent par exemple qu'aucune authentification ne doit avoir lieu au cours d'une transaction, ou qu'aucune exception autre qu'une exception référencée dans la norme ISO ne peut arriver jusqu'à l'utilisateur.

Des langages de spécifications, tel que JML (Java Modeling Language) ont été employé avec succès pour valider des applications s'exécutant sur des cartes à puce. Cependant, les propriétés de sécurité de haut niveau ne peuvent pas, le plus souvent, etre directement exprimées dans ces languages. Cet article propose donc une méthode pour traduire automatiquement les propriétés de haut niveau en spécifications JML. La méthode consiste à propager à travers une application des annotations synthétiques appropriées à une propriété. Ainsi dans une seconde étape, la politique de sécurité peut etre validée en utilisant les outils de validation existants pour JML. La méthode est générale et peut etre appliquée à une grande classe des propriétés de sécurité.

Elle a notamment été appliquée sur plusieurs exemples réalistes d'applications pour carte, et a réellement permis de trouver des violations de politiques de sécurité.

Mots-clés : sécurité, applets, Java Card, génération des spécifications

1 Introduction

Program verification techniques are increasingly being advocated by industry as a way to provide high quality software. In particular in the context of smart cards they already have been successfully used to verify functional properties and interoperability (*i.e.* platform-independence) of applications, and also to discover subtle programming errors that remain undetected by intensive testing [4, 6]. However, current techniques are often not appropriate to verify typical security policies for applets (smart card applications), expressed as high level rules, such as "no authentication must take place within a transaction", or "an applet can be personalised only once".

Moreover, the cost of employing program verification techniques remains an important obstacle for most industrials. Our experiences, which are confirmed by two recent roadmaps for smart card research [12, 1], show that the difficulty of learning a specification language whose internals may be obscure to programmers, and the large amount of work required to formally specify and verify applications constitute major obstacles to the use of program verification techniques in industry. Therefore, recent work on formal methods for Java and Java Card[1] tries to tackle these problems.

To reduce the difficulty of learning a specification language, the Java Modeling Language (JML) [16] has been designed as an easily accessible specification language. It uses a Java-like syntax with some specification-specific keywords added. JML allows developers to specify the properties of their program in a generalisation of Hoare logic, tailored to Java. By now, it has been generally accepted as *the* behavioural interface specification language for Java (Card).

For the verification of Java (Card) programs, several tools are available – based on Hoare logic [15] or weakest preconditions calculus [14] – using (variations of) JML as specification language. We mention *e.g.* JACK (Java Applet Correctness Kit) [7], Jive [21], Krakatoa [19], Loop [3] and ESC/Java [18]. These tools vary in the amount of user interaction required, but also in the level of correctness they provide. Jive, Krakatoa and Loop are sound, but require much user interaction, while ESC/Java is automatic, but unsound. For smart card industry, both soundness and automation are major concerns. One needs to be able to give firm correctness guarantees, while automation provides scalability and usability. JACK is the tool which addresses these issues best, combining soundness with a high degree of automation, therefore we use it in our work.

In contrast, the problem of actually writing the specifications remains largely unaddressed. Specifying a smart card application is labour-intensive and error-prone, as it is easy to forget some annotations. There exist tools which assist in writing annotations, *e.g.* Daikon [10] and Houdini [11], but these use heuristic methods and do not require any user input. Moreover, they typically only produce annotations for simple safety and functional invariants, and they cannot be used to synthesise realistic security policies.

Therefore, we propose a method that, given a security policy, automatically annotates a Java (Card) application, in such a way that if the application respects the annotations then

[1] Java Card is a dialect of Java, tailored explicitly to smart card applications.

it also respects the security policy. The generation of annotations proceeds in two phases: synthesising and weaving.

1. Based on the security policy we *synthesise* core annotations, specifying the behaviour of the methods directly involved.

2. We propagate appropriate annotations for all methods directly or indirectly invoking the methods that form the core of the security policy, thus *weaving* the security policy through the application.

For example, suppose that the security policy prescribes that the synthesised core-annotations contain precondition P and postcondition Q for method m. When weaving the annotations, every method n that calls method m will also get precondition P and post-condition Q, *unless* we find that the implementation of n establishes P before calling m, or breaks Q after returning from m. Next, the same approach will be used for all methods calling n, *etc.* Once the whole application is annotated, using JACK one can verify automatically whether the application respects the security policy. This whole process might seem trivial, but doing it manually is labour-intensive and error-prone. Our method provides a sound, automatic and cost-effective way for checking security policies.

The annotations that we generate all use JML's static ghost variables that are special specification-only variables. JML also defines a special ghost-assignment annotation for these variables. Since we use only static ghost variables, the properties that we express are independent of the particular class instances available. We have defined special weakest precondition and strongest postcondition calculi, considering static ghost variables only, and we have proven that our algorithms for weaving the annotations correspond exactly to these calculi.

To show the usefulness of our approach, we applied the algorithm to several realistic examples of smart card applications. When doing this, we actually found violations against the security policies documented for some of these applications.

This paper is organised as follows. Section 2 introduces several typical high-level security properties. Next, Section 3 presents the process to weave these properties throughout applications. Subsequently, Section 4 discusses the application of our method to realistic examples. Finally, Sections 5 and 6 present related work and draw conclusions.

2 High-level Security Properties for Applets

Over the last years, smart cards have evolved from proprietary into open systems, making it possible to have applications from different providers on a single card. To ensure that these applications cannot damage the other applications or the card, strict security policies – expressed as high-level security properties – must be obeyed. Below we will present several examples of such security properties. Such properties are high-level in the sense that they have impact on the whole application and are not restricted to single classes. It is important to notice that we restrict our attention to source code-level security of applications.

The properties that we consider can be divided in several groups, dealing with different aspects of smart cards. First of all there are properties dealing with the so-called *applet life cycle*, describing the different phases that an applet can be in. Many actions can only be performed when an applet is in a certain phase. Second, there are properties dealing with the transaction mechanism, the Java Card solution for having atomic updates. Further there are properties restricting the kind of exceptions that can occur, and finally, we consider properties dealing with access control, limiting the possible interactions between different applications. For each group we present some example properties. However, we would like to emphasise that there exist many more relevant security properties for smart cards, for example specifying memory management, information flow and management of sensitive data. Identifying all relevant security properties for smart cards, and expressing them formally, is an important ongoing research issue.

Applet life cycle A typical applet life cycle defines phases as loading, installation, personalisation, selectable, blocked, and dead (see *e.g.* [20]). Each phase corresponds to a different moment in the applet's life. First an applet is loaded on the card, then it is properly installed and registered with the Java Card Runtime Environment. Next the card is personalised, *i.e.* all information about the card owner, permissions, keys *etc.* is stored. After this, the applet is selectable, which means that it can be repeatedly selected, executed, and deselected. However, if a serious error occurs, for example there have been too many attempts to verify a pin code, the card can get blocked or even become dead. From the latter state, no recovery is possible.

In many of these phases, restrictions apply on who can perform actions, or on which actions can be performed. These restrictions give rise to different security properties, to be obeyed by the applet.

> **Authenticated initialisation** Loading, installing and personalising the applet can only be done by an authenticated authority.
>
> **Authenticated unblocking** When the card is blocked, only an authenticated authority can execute commands and possibly unblock it.
>
> **Single personalisation** An applet can be personalised only once.

Atomicity A smart card does not include a power supply, thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification prescribes the use of a transaction mechanism to control synchronised updates of sensitive data. A statement block surrounded by the methods `beginTransaction()` and `commitTransaction()` can be considered atomic. If something happens while executing the transaction (or if `abortTransaction()` is executed), the card will roll back its internal state to the state before the transaction was begun.

To ensure the proper functioning and prevent abuse of this mechanism, several security properties can be specified.

No nested transactions Only one level of transactions is allowed.

No exception in transaction All exceptions that may be thrown inside a transaction, should also be caught inside the transaction.

Bounded retries No authentication may happen within a transaction.

The second property ensures that the `commitTransaction` will always be executed. If the exception is not caught, the `commitTransaction` would be ignored and the transaction will not be finished. The last property excludes authentication within a transaction. If this would be allowed, one could abort the transaction every time a wrong authentication attempt has been made. As this rolls back the internal state to the state before the transaction was started, this would also reset the retry counter, thus allowing an unbounded number of retries.

Exceptions Raising an exception at the top level can reveal information about the behaviour of the application and in principle it should be forbidden. However, sometimes it is necessary to pass on information about a problem that occurred. Therefore, the Java Card standard defines so-called ISO exceptions, where a pre-defined status word explains the problem encountered. These exceptions are the only exceptions that may be visible at top-level; all other exceptions should be caught within the application.

Only ISO exceptions at top-level No exception should be visible at top-level, except ISO exceptions.

Access control Another feature of Java Card is an isolation mechanism between applications: the firewall. The firewall ensures that several applications can securely co-exist on the same card, while managing limited collaboration between them: classes and interfaces defined in the same package can freely access each other, while external classes can only be accessed via explicitly shared interfaces. Inter-application communication via shareable interfaces should only take place when the applet is selectable, in all other phases of the applet life cycle only authenticated authorities are allowed to access the applet.

Only selectable applications shareable An application is accessible via a shareable interface only if it is selectable.

3 Automatic Verification of Security Properties

As explained above, we are interested in the verification of high-level security properties that are not directly related to a single method or class of the application, but that guarantee its overall well-functioning. Writing appropriate JML annotations for such properties is tedious and error-prone, as they have to be spread all over the application. Therefore, we propose a way to construct such annotations automatically. First we synthesise core-annotations for methods directly involved in the property. For example, when specifying that no nested

transactions are allowed, we annotate the methods `beginTransaction`, `commitTransaction` and `abortTransaction`. Subsequently we propagate the necessary annotations to all methods (directly or indirectly) invoking these core-methods. The generated annotations are sufficient to respect the security properties, *i.e.* if the applet does not violate the annotations, it respects the corresponding high-level security property.

Whether the applet respects its annotations can be established with any of the existing tools for JML. We use JACK [7], which generates proof obligations accepted by the AtelierB prover[2] and Simplify[3]. Both are automatic verifiers for first-order logical formulae. Since for most security properties the annotations are relatively simple – but there are many – it is important that these verifications are done automatically, without any user interaction. The results in Section 4 show that for the generated annotations all correct proof obligations can indeed be automatically discharged.

Before presenting the overall architecture of our tool set and outlining the algorithm for propagation of annotations, we briefly present a few JML keywords, that are relevant for the examples presented here.

3.1 JML in a nutshell

JML [16] uses a Java-like syntax to write predicates, extended with several specification-specific constructs, such as `\forall`, `\exists` *etc.* Method specifications are given using the keywords `requires` (preconditions), `ensures` (postconditions) and `exsures` or `signals` (exceptional postconditions, *i.e.* the condition that has to hold upon abnormal termination of a method). For methods we can also specify which variables may be modified, using a so-called `assignable` clause. Class invariants, describing properties that have to be preserved by each method are denoted using the keyword `invariant`.

To make specifications more abstract and implementation-independent, JML provides several means of abstraction. One of these are the so-called ghost-variables, which are visible only in specifications. Their declaration is preceded by the keyword `ghost`. A special assignment annotation `set` allows to update its value. Using invariants they can be related to concrete variables.

A large class of security properties can be expressed using static ghost variables with primitive type only (including the ones presented in Section 2). Since in this paper we restrict our attention to these properties, we only study annotations containing static ghost variables. The static ghost variables are typically used to keep track of the control state of the application.

To give an example JML specification, we show a fragment of the core-annotation for the **No nested transactions** property. A static ghost variable `TRANSACT` is declared that keeps track of whether there is a transaction in progress. It is initialised to 0, denoting that there is no transaction in progress.

```
/*@ static ghost int TRANSACT == 0; @*/
```

[2] See `http://www.atelierb.societe.com/`.

[3] See `http://research.compaq.com/SRC/esc/Simplify.html`.

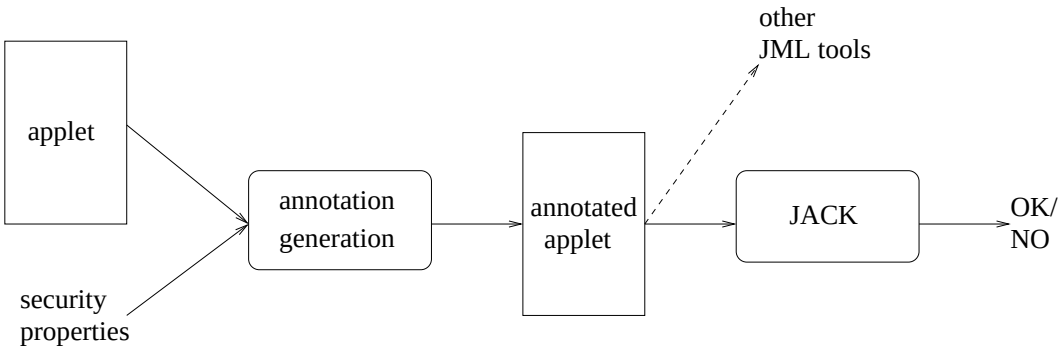

Figure 1: Tool set for verifying high-level security properties

The method `beginTransaction` is annotated as follows.

```
/*@ requires TRANSACT == 0;
  @ assignable TRANSACT;
  @ ensures TRANSACT == 1; @*/
public static native void beginTransaction()
                          throws TransactionException;
```

Since the method is native, one cannot describe its body. However, if it had been a non-native method, its body would have to contain a special assignment `//@ set TRANSACT = 1;` to ensure that the method itself satisfies its annotations.

3.2 Architecture

Figure 1 shows the general architecture of the tool set we provide for verifying high-level security properties. In principle, our annotation generator can be used as a front-end for any tool accepting JML-annotated Java (Card) applications. As input we have a security property and a Java Card applet. The output is a JML Abstract Syntax Tree (AST), using the format as defined for the standard JML parser. When pretty-printed, this AST corresponds to an JML-annotated Java file. From this annotated file, JACK generates appropriate proof obligations to check whether the applet respects the security property.

3.3 Automatic Generation of Annotations

Section 4 presents example core-annotations for some of the security properties presented in Section 2, here we focus on the weaving phase, *i.e.* how the core-annotations are propagated throughout the applet. We define functions `pre`, `post` and `excpost`, propagating preconditions, postconditions and exceptional postconditions, respectively. These functions have

been defined and implemented for the full Java Card language, but to present our ideas, we only give the definitions for a representative subset of Java Card statements: statement composition, method calls, conditional and `try-catch` statements. We assume the existence of the domains MethName of method names, Stmt of Java Card statements, and Expr of Java Card expressions, respectively. Moreover, we assume the existence of functions call and body, denoting a method call and body, respectively.

Propagation of preconditions First, we will present the definition of the function pre that is used to propagate preconditions. This function analyses a method body in a sequential way – from beginning to end – computing which preconditions of the methods called within the body have to be propagated. To understand the reasoning behind the definition, we will first look at an example. Suppose we are checking the **No nested transactions** property for an application, which contains a method `m`, whose only method calls are those shown.

```
void m() {
  ... // some internal computations
  JCSystem.beginTransaction();
  ... // computations within transaction
  JCSystem.commitTransaction();
}
```

To check this property, core-annotations are synthesised for `beginTransaction` and `commitTransaction`. The annotations for `beginTransaction` are presented in Section 3.1 above, while `commitTransaction` requires `TRANSACT == 1` and ensures `TRANSACT == 0`. As we assume that no other methods are called in `m`, the only way the precondition of `beginTransaction` can hold, is by requiring that it already holds at the moment `m` is called. Thus, the precondition of `beginTransaction` has to be propagated. In contrast, the precondition for `commitTransaction` (`TRANSACT == 1`) has to be established by the postcondition of `beginTransaction`, because the variable `TRANSACT` is modified by this method. Propagating the precondition of `commitTransaction` to the preconditions of `m` would not help to guarantee that the precondition of `commitTransaction` holds. Thus, only preconditions expressing properties over unmodified variables should be propagated. Propagating pre- or postconditions can be considered as passing on a method contract. Methods can only pass on contracts for variables they do not modify; if they modify a variable it is their duty to ensure that the necessary conditions are satisfied.

In the definition of the function pre for propagating preconditions, we assume the existence of a function mod returning the set of static ghost variables modified by a statement. As we are only interested in static ghost variables with primitive types, the definition is straightforward and it does not have to consider aliasing. Further, we assume the existence of the domains Var of static ghost variables and Pred of predicates, containing static ghost variables only - with function fv, returning the set of free variables used in a predicate.

We define pre on method names, statements and expressions. These definitions are mutually recursive. Java Card applets typically do not contain (mutually) recursive method

calls, therefore this does not cause any problems. Generating appropriate annotations for recursive methods would require more care (and in general it might not be possible to do without any user interaction).

Definition 1 (pre) *We define*

$$\begin{array}{rcl} \mathsf{pre} & : & \mathsf{MethName} \to \mathcal{P}(\mathsf{Pred}) \\ \mathsf{pre} & : & \mathsf{Stmt} \to \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Pred}) \\ \mathsf{pre} & : & \mathsf{Expr} \to \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Pred}) \end{array}$$

by rules like (where $m, n : \mathsf{MethName}$*,* $s_1, s_2 : \mathsf{Stmt}$*,* $c : \mathsf{Expr}$ *and* $V : \mathcal{P}(\mathsf{Var})$*):*

$$\begin{array}{rcl} \mathsf{pre}(m) & = & \mathsf{pre}(\mathsf{body}(m), \emptyset) \\ \mathsf{pre}(s_1;s_2, V) & = & \mathsf{pre}(s_1, V) \cup \mathsf{pre}(s_2, V \cup \mathsf{mod}(s1)) \\ \mathsf{pre}(\mathsf{call}(n), V) & = & \{p \mid p \in \mathsf{pre}(n) \wedge \mathsf{fv}(p) \cap V = \emptyset\} \\ \mathsf{pre}(\texttt{if}\ (c)\ s_1\ \texttt{else}\ s_2, V) & = & \mathsf{pre}(c, V) \cup \mathsf{pre}(s_1, V \cup \mathsf{mod}(c)) \cup \mathsf{pre}(s_2, V \cup \mathsf{mod}(c)) \\ \mathsf{pre}(\texttt{try}\ s_1\ \texttt{catch}\ (E)\ s_2, V) & = & \mathsf{pre}(s_1, V) \cup \mathsf{pre}(s_2, V \cup \mathsf{mod}(s1)) \end{array}$$

In the rules defining pre on Stmt and Expr, the second argument denotes the set of variables that have been modified so far. When calculating the precondition for a method, we calculate the precondition of its body, assuming that no variables have been modified so far. For a statement composition, we first propagate the preconditions for the first sub-statement, and then for the second sub-statement, but taking into account the variables modified by the first sub-statement. When propagating the preconditions for a method call, we propagate all preconditions of the called method that do not contain modified variables. Since we are restricting our annotations to expressions containing static ghost variables only, in the rule for the conditional statement we cannot take the outcome of the conditional expression into account. As a consequence, we sometimes generate too strong annotations, but in practice this does not cause problems. Moreover, it should be emphasised that this only can make us reject correct applets, but it will never make us accept incorrect ones. Similarly, for the `try-catch` statement, we always propagate the precondition for the `catch` clause, without checking whether it actually can get executed. Again, this will only make us reject correct applets, but it will never make us accept incorrect ones.

Notice that by definition, we have the following property for the function pre (where s is either in Stmt or Expr, and V is a set of static ghost variables).

$$p \in \mathsf{pre}(s, V) \Leftrightarrow (p \in \mathsf{pre}(s, \emptyset) \wedge \mathsf{fv}(p) \cap V = \emptyset)$$

Propagation of postconditions In a similar way, we define functions post and excpost, computing the set of postconditions and exceptional postconditions that have to be propagated for method names, statements and expressions. The main difference with the definition of pre is that they run through a method from the end to the beginning. Moreover, they have to take into account the different paths through the method. For each of these possible paths, we calculate the appropriate (exceptional) postcondition. The overall (exceptional) postcondition is then defined as the disjunction of the postconditions related to the different paths through the method.

Example For the example discussed above, our algorithms generate the following annotations (including the information computed by the function mod).

```
/*@ requires TRANSACT == 0;
  @ assignable TRANSACT;
  @ ensures TRANSACT == 0; @*/
void m() {
  ... // some internal computations
  JCSystem.beginTransaction();
  ... // computations within transaction
  JCSystem.commitTransaction();
}
```

This might seem trivial, but it is important to realise that similar annotations will be generated for all methods calling `m`, and transitively for all methods calling the methods calling `m` *etc.* Having an algorithm to generate such annotations enables to check automatically a large class of high-level security properties.

3.4 Annotation generation and predicate transformer calculi

A natural question that arises is the relation between the computations done by the functions pre, post and excpost and well-known program transformation methods as the weakest precondition and strongest postcondition calculi.

As explained above, we consider annotations containing static ghost variables only, and in particular we do not take the outcome of conditional expressions in a branching statement into account. Therefore, we cannot show any relationship between the function pre and the standard *wp*-calculus. However, we have defined an abstract version of the *wp*-calculus, denoted as $\mathsf{wp}^{\#}$, which only transforms predicates containing static ghost variables, and we can prove a correspondence between this abstract *wp*-calculus and the function pre, defined above.

The rules for $\mathsf{wp}^{\#}$ are similar to the rules for the standard *wp*-calculus; the main difference is that they do not take conditions into account. For example, the rule for the conditional statement is defined as follows[4].

$$\mathsf{wp}^{\#}(\texttt{if}(c)s_1\ \texttt{else}\ s_2, Q) = \mathsf{wp}^{\#}(c, \mathsf{wp}^{\#}(s_1, Q)) \wedge \mathsf{wp}^{\#}(c, \mathsf{wp}^{\#}(s_2, Q))$$

For our work, the most interesting rule is the rule for method calls, which is similar for the abstract and the traditional weakest precondition calculus. We assume the existence of functions requires and ensures, returning the annotated preconditions and postconditions of m, respectively.

$$\mathsf{wp}^{\#}(\mathsf{call}(m), Q) = \mathsf{requires}(m) \wedge \forall \mathsf{mod}(m).(\mathsf{ensures}(m) \Rightarrow Q)$$

[4] In fact, following [17, 14] we have adapted our abstract *wp*-calculus in order to take exceptional postconditions into account. For clarity, we will ignore this in this paper.

Notice that if the postcondition Q contains predicates containing only variables not mentioned in the modifies clause of m, then these can be taken out of the quantification[5]. Using this, we can prove the following property for the abstract weakest precondition calculus.

$$\forall s\colon \mathsf{Stmt}, Q_1, Q_2\colon \mathsf{Pred}.(\mathsf{mod}(s) \cap \mathsf{fv}(Q_2) = \emptyset) \wedge \mathsf{post}(s) \neq \lambda x.\mathsf{false} \Rightarrow \\ \mathsf{wp}^{\#}(s, Q_1 \wedge Q_2) = \mathsf{wp}^{\#}(s, Q_1) \wedge Q_2$$

This property is crucial for the correspondence between pre and the abstract weakest precondition. Notice that for the standard *wp*-calculus, one can prove a similar property, provided one has an appropriate definition for mod.

When annotating a program with static ghost variables of primitive types only, the abstract *wp*-calculus is sufficient. That is, every program that can be proven correct with the abstract *wp*-calculus, can also be proven correct with the standard *wp*-calculus.

Lemma 1 *For any statement s, and any predicates P and Q, containing static ghost variables only, we have:*

$$\forall P, Q\colon \mathsf{Pred}, s\colon \mathsf{Stmt}.(P \Rightarrow \mathsf{wp}^{\#}(s, Q)) \Rightarrow (P \Rightarrow \mathsf{wp}(s, Q))$$

The converse of this implication does not hold. Consider for example the following method.

```
void m(boolean atomic) {
  if (atomic) JCSystem.beginTransaction();
  // some computations
  if (atomic) JCSystem.commitTransaction();
}
```

Suppose that we wish to compute the weakest precondition for this method to ensure true, *i.e.* it will not throw any exceptions. The abstract weakest precondition will be `TRANSACT == 0 & TRANSACT == 1`, which obviously simplifies to false and thus never can be established. However, the standard weakest precondition calculus will return `atomic ==> TRANSACT == 0`, which is indeed expected.

Finally, we can show the relationship between the abstract weakest precondition and the function pre. The function pre computes the unbounded part of the abstract weakest precondition. Formally, we express this as follows.

Theorem 1 (Correspondence) *For any statement s, its abstract weakest precondition is equivalent to the calculated precondition, in conjunction with a universally quantified expression F.*

$$\exists F\colon \mathsf{Pred}.\mathsf{wp}^{\#}(s, \lambda x.\mathsf{true}) = (\mathsf{pre}(s, \emptyset) \wedge \forall \mathsf{mod}(s).F)$$

The proof uses structural induction and the properties for pre and $\mathsf{wp}^{\#}$ above. We believe similar equivalences can be proven for the function post and the strongest postcondition calculus. However, we are not aware of any adaptation of the *sp*-calculus to Java, therefore we have not studied this any further.

[5] Except when the postcondition of m is equal to false.

4 Results

We checked whether different realistic examples of Java Card applications respect the security properties presented in Section 2, and actually found some violations. This section presents these results, focusing on the atomicity properties.

4.1 Core-annotations for Atomicity Properties

The core-annotations related to the atomicity properties specify the methods related to the transaction mechanism declared in `javacard.framework.JCSystem` of the Java Card API. As explained above a static ghost variable `TRANSACT` is used to keep track of whether there is a transaction in progress. Section 3.1 presented the annotations for method `beginTransaction`, for `commitTransaction` and `abortTransaction` similar annotations are synthesised. After propagation, these annotations are sufficient to check for the absence of nested transactions.

To check for the absence of uncaught exceptions inside transactions, we use a special feature of JACK, namely pre- and postcondition annotations for statement blocks (as presented in [7]). Block annotations are similar to method specifications. The propagation algorithm is adapted, so that it not only generates annotations for methods, but also for designated blocks. As core-annotation, we add the following annotation for `commitTransaction`.

```
/*@ exsures (Exception) TRANSACT == 0; @*/
public static native void commitTransaction()
                          throws TransactionException;
```

This specifies that exceptions only can occur if no transaction is in progress. Propagating these annotations to statement blocks ending with a commit guarantees that these only can raise exceptions, if they do not start a transaction.

Finally, in order to check that only a bounded number of retries of pin-verification is possible, we annotate the method `check` (declared in the interface `javacard.framework.Pin` in the standard Java Card API) with a precondition, requiring that no transaction is in progress.

```
/*@ requires TRANSACT == 0; @*/
public boolean check(byte[] pin, short offset, byte length);
```

4.2 Checking the Atomicity Properties

As mentioned above, we tested our method on realistic examples of industrial smart card applications, including the so-called Demoney case study, developed as a research prototype by Trusted Logic[6], and the PACAP case study[7], developed by smart card producer Gemplus. Both examples have been explicitly developed as test cases for different formal techniques,

[6] `http://www.trusted-logic.fr`
[7] `http://www.gemplus.com/smart/r_d/publications/case-study`

illustrating the different issues involved when writing smart card applications. We used the core-annotations as presented above, and propagated these throughout the applications.

For both applications we found that they contained no nested transactions, and that they did not contain attempts to verify pin codes within transactions. All proof obligations generated *w.r.t.* these properties are trivial and can be discharged immediately. However, to emphasise once more the usefulness of having a tool for generating these annotations, in the PACAP case study we encountered cases where a single transaction gave rise to twenty-three annotations in five different classes. When writing these annotations manually, it is easy to forget some of these annotations.

Finally, in the PACAP application we found transactions containing uncaught exceptions. Consider for example the following code fragment.

```
void appExchangeCurrency(...) {
   ...
   /*@ exsures (Exception) TRANSACT == 0; @*/ {
      ...
      JCSystem.beginTransaction();
      try {
         balance.setValue(decimal2);
         ...
      } catch (DecimalException e) {
         ISOException.throwIt(PurseApplet.DECIMAL_OVERFLOW);
      }
      JCSystem.commitTransaction();
   }
   ...
}
```

The method `setValue` that is called can actually throw a decimal exception, which would lead to throwing an ISO exception, and the transaction would not be committed. This clearly violates the security policy as described in Section 2. After propagating the core-annotations, and computing the appropriate proof obligations, this violation can be found automatically, without any problems.

5 Related work

Our approach to enforce security policies relies on the combination of: i) an annotation assistant that generates JML annotations from high-level security properties; ii) a lemma generator that produces proof obligations for annotated applets, using *e.g.* a weakest precondition calculus; and iii) an automated or interactive theorem prover to discharge all proof obligations generated by the lemma generator. Experience suggests that our approach provides accurate and automated analyses that may handle a wide range of security properties.

It is to be contrasted with some other approaches to enforce security policies statically, in particular:

- program analyses, which are fully automatic, but whose results are often hard to exploit due to a very high rate of spurious warnings; and
- type systems, which are implemented by automatic type inference engines, but usually tailored towards a single property, *e.g.* confidentiality or availability, and often too restrictive to be used in practice.

Proof-carrying code [22] provides an appealing solution to enforce security policies statically, but does not directly address the problem of obtaining appropriate specifications for the code to be downloaded. In fact, our mechanism may be used in the context of proof-carrying code as a generator of verification conditions from high-level security properties.

Run-time monitoring provides a dynamic measure to enforce safety and security properties, and has been instrumented for Java through a variety of tools, see *e.g.* [2, 5, 24]. Security automata provide another means to monitor a program execution and to enforce security policies. They have been extensively studied by Schneider, Morrisett, and Walker [23, 26, 13], who propose different forms of automata (edit automata, truncation automata, insertion automata, *etc.*) to prevent or react against violations of security polices. Inspired by aspect-oriented programming, Colcombet and Fradet [8] propose a technique to compose programs in a simple imperative language with optimised security automata. However, run-time monitoring is not an option for smart card applications, in particular because of the smart card's limited resources.

Leaving the application domain of security policies and focusing on program specification and verification techniques, one encounters annotation assistants, such as Daikon and Houdini. But these tools synthesise simple safety annotations and functional invariants, without being guided by user input—in our case, the security properties are user input. Further apart, one encounters testing, which remains the technique most commonly used by the smart card industry to guarantee the quality of applications. However, testing cannot guarantee that an application complies to its security policy, because the program is only tested on sample values; in fact, program verification techniques have been shown to discover subtle programming errors that remain undetected by intensive testing.

6 Conclusion

We have developed a mechanism to synthesise JML annotations from high-level security properties. The mechanism has been implemented as a front-end for tools accepting JML-annotated Java programs; we use it in combination with JACK. The resulting tool set has been successfully applied to the area of smart cards, both to verify secure applications, and to discover programming errors in insecure ones. Our broad conclusion is that the tool set contributes to effectively carrying out formal security analyses, while also being reasonably accessible to security experts without intensive training in formal techniques.

Currently, we are developing solutions to hide the complexity of generating core annotations from the user. To this end, we plan to develop appropriate formalisms for expressing high-level security properties, with compilers that translate properties expressed in these formalisms into appropriate JML core-annotations. Possible formalisms include security automata, for which appealing visual representations can be given, or more traditional logics, such as temporal logic, that allow to specify the behaviour of an application. In the latter case, we believe that it will be necessary to rely on a form of security patterns reminiscent of the specification patterns developed by Dwyer *et al.* [9], and also to consider extensions of JML, *e.g.* JML with temporal logic [25].

Further, we intend to apply our methods and tools in other contexts, and in particular for mobile phone applications. In particular, this will require extending our tools to other Java technologies that, unlike Java Card, feature multi-threading.

References

[1] Roadmap for European Research on Smartcard Technologies. See `http://www.ercim.org/reset`

[2] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *ENTCS*, volume 55(2). Elsevier Publishing, 2001.

[3] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, number 2031 in LNCS, pages 299–312. Springer, 2001.

[4] P. Bieber, J. Cazin, V.Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - second generation of a Java model checker. In *Workshop on Advances in Verification*, 2000.

[6] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal Methods for Smart Cards: an experience report. Technical Report NIII-R0316, NIII, University of Nijmegen, 2003.

[7] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Formal Methods (FME'03)*, number 2805 in LNCS, pages 422–439. Springer, 2003.

[8] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of POPL'00*, pages 54–66. ACM Press, 2000.

[9] M. Dwyer, G. Avrunin, and J. Corbett. Property Specification Patterns for Finite-state Verification. In *2nd Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[10] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[11] C. Flanagan and K.R.M. Leino. Houdini, an annotation assistant for ESC/Java. In J.N. Oliveira and P. Zave, editors, *Formal Methods Europe 2001 (FME'01): Formal Methods for Increasing Software Productivity*, number 2021 in LNCS, pages 500–517. Springer, 2001.

[12] European Commission Information Society Directorate General. Research for the smart card of 2010, 2001. Available from `ftp://ftp.cordis.lu/pub/ist/docs/ka2/`.

[13] K. Hamlen, G. Morrisett, and F.B. Schneider. Computability classes for enforcement mechanisms. Technical Report 2003-1908, Department of Computer Science, Cornell University, 2003.

[14] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 2003. To appear.

[15] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lecture Notes in Computer Science, pages 284–299. Springer-Verlag, 2001.

[16] JML Specification Language. `http://www.jmlspecs.org`.

[17] K.R.M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[18] K.R.M. Leino, G. Nelson, and J.B. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000.

[19] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. *Journal of Logic and Algebraic Programming*, 2003. To appear.

[20] R. Marlet and D. Le Métayer. Security properties and Java Card specificities to be studied in the SecSafe project, 2001. Number: SECSAFE-TL-006.

[21] J. Meyer and A. Poetzsch-Heffter. An architecture of interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in LNCS, pages 63–77. Springer, 2000.

[22] G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[23] F. B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, October 1999.

[24] L. Tan, J. Kim, and I. Lee. Testing and monitoring model-based generated program. In *Proceeding ofRV'03*, volume 89 of *ENTCS*. Elsevier, 2003.

[25] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST'02)*, number 2422 in LNCS, pages 334–348. Springer, 2002.

[26] D. Walker. A Type System for Expressive Security Policies. In *Proceedings of POPL'00*, pages 254–267. ACM Press, 2000.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399