ENFORCING HIGH-LEVEL SECURITY PROPERTIES FOR APPLETS

Mariela Pavlova[1], Gilles Barthe[1], Lilian Burdy[1], Marieke Huisman[1] and Jean-Louis Lanet[2]
[1] *INRIA Sophia Antipolis, France and* [2] *INRIA Dir DRI, France*

Abstract Smart card applications often handle privacy-sensitive information, and therefore must obey certain security policies. Typically, such policies are described as high-level security properties, stating for example that no pin verification 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 directly be expressed in such languages. Therefore, this paper proposes a method to translate high-level security properties into JML annotations. The method synthesises appropriate annotations and weaves them throughout the application. In this way, security policies can be validated using existing tools for JML. The method is general and applies to a large class of security properties.

To validate the method, it has been applied to several realistic examples of smart card applications. This allowed us to find violations against the documented security policies for some of these applications.

Keywords: Smart devices, security, specification, verification

1. Introduction

Nowadays, most efforts in smart card security focus on adequate countermeasures against hardware attacks. However, logical attacks, caused by *e.g.* illegal control flow or uncaught exceptions, form a new major threat for security and privacy. An example of such an attack is a malicious GSM applet that performs illegal calls to the method `sendSMS`.

To ensure user confidence, smart card application providers therefore have to guarantee the dependability of their software. This can be achieved by following certification procedures, such as "Common Criteria[1]", focusing on security aspects. But such procedures are relatively heavy, and they are also concerned with aspects unrelated to software security. Therefore, industry often prefers to do a more lightweight analysis or software audit.

Such an analysis typically consists in a manual deep code review, for which no tool support is available. Therefore, this is a costly procedure, and there is no formal guarantee of its results. The quality of this analysis can be improved by using program verification techniques. Therefore, industry is investigating how these techniques can be used to provide high quality software. For example, in the context of smart cards, program verification has been successfully used to verify functional properties of applications, discovering subtle programming errors that remain undetected by intensive testing [3, 5].

Unfortunately, the cost of employing program verification techniques remains an important obstacle for most industrials. Our experiences, which are confirmed by two recent road-maps for smart card research[2], 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[3] tries to tackle these problems.

To reduce the difficulty of learning a specification language, the Java Modeling Language (JML)[4] [6] 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).

While JML is easily accessible to Java developers, actually writing the specifications of 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 these annotations, *e.g.* Daikon [11] and Houdini [12] use heuristic methods to produce annotations for simple safety and functional invariants. However, these tools cannot be guided by the user—they do not require any user input—and in particular cannot be used to synthesise annotations from realistic security policies.

The main contribution of this paper is 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 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 Next we propagate these annotations to all methods directly or indirectly invoking the methods that form the core of the security policy, thus *weaving* the security policy throughout the application.

The need for such a propagation phase stems from the fact that we are interested in doing static verification. We need tool support for the propagation, because a typical security property may involve methods from different classes, as illustrated below. The annotations that we generate all use JML static ghost variables: special specification-only variables, that can be modified via a special ghost-assignment annotation. Since we use only static ghost variables, the properties are independent of the particular class instances available.

The annotations we generate can be checked with existing verification tools *e.g.* JACK (Java Applet Correctness Kit) [7], Jive [17], Krakatoa [15], Loop [2] and ESC/Java [14]. We use JACK as it provides the best compromise between soundness, efficiency, scalability and usability.

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 itself, strict security policies—expressed as high-level security properties—must be obeyed. Such properties are high-level in the sense that they have impact on the whole application and are not restricted to single classes. Below we will present several examples. 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, related to 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. For all these properties encodings into JML annotations exist.

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.* [16]). 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 pin verification 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 would not be finished. The last property excludes pin verification within a transaction. If this would be allowed, one could abort the transaction every time a wrong pin code has been entered. 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. Even though the specification of the Java Card API prescribes that the retry counter for pin verification cannot be rolled back, in general one has to check this kind of properties.

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, but that guarantee the overall well-functioning of an application. 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 for different provers, including the AtelierB prover[5] and Simplify[6]. 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 [6] uses a Java-like syntax to write predicates, extended with several specification-specific constructs, such as `\forall`, `\exists` *etc.* Method specifications consist of preconditions (`requires`), postconditions (`ensures`), and exceptional postconditions (`exsures`), *i.e.* the condition that has to hold upon abnormal termination of a method. We can also specify so-called `assignable` clauses, stating which variables may be modified by a method. Class invariants (keyword `invariant`) describe properties that have to be preserved by each method.

To write more abstract and implementation-independent specifications, JML provides several means of abstraction. One of these are 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 their value. Using invariants they can be related to concrete variables.

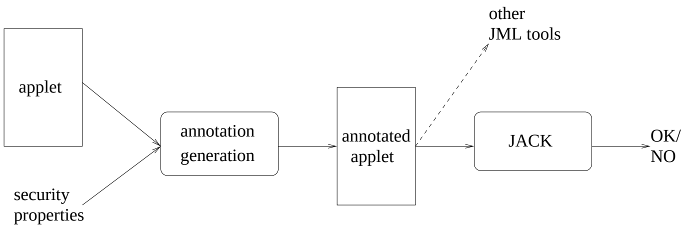

Figure 1. Tool set for verifying high-level security properties

A large class of security properties can be expressed using static ghost variables of primitive type; these are typically used to keep track of the control state of the application (including the ones presented in Section 2). Therefore, here we only study the propagation of annotations containing static ghost variables of primitive type. However, our propagation technique easily can be generalised to concrete (static) variables, as long as we do not have to handle aliasing.

To give an example JML specification, we show a fragment of the core-annotation for the **No nested transactions** property. A static ghost variable `TRANS` 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 TRANS == 0; @*/
```

The method `beginTransaction` is annotated as follows.

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

Since the method is native, one cannot describe its body. However, if it had been non-native, an annotation `//@ set TRANS = 1;` would have been generated, to ensure that the method satisfies its specification.

3.2 Architecture

Figure 1 shows the general architecture of the tool set for verifying high-level security properties. 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 a 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 mod, pre, post and excpost, propagating assignable clauses, 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 statements: statement composition, method calls, conditional and `try-catch` statements and special set-annotations. We assume the existence of domains MethName of method names, Stmt of Java Card statements, Expr of Java Card expressions, and Var of static ghost variables, and functions call and body, denoting a method call and body, respectively.

All functions are defined as mutual recursive functions on method names, statements and expressions. When a method call is encountered, the implementation will check whether annotations already have been generated for this method (either by synthesising or weaving). If not it will recursively generate appropriate annotations. 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).

Propagation of assignable clauses. First we define a function mod that propagates assignable clauses for static ghost variables.

Definition 1 (**mod**) *We define functions* $\mathsf{mod}\colon \mathsf{MethName} \to \mathcal{P}(\mathsf{Var})$*,* $\mathsf{mod}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Var})$*, and* $\mathsf{mod}\colon \mathsf{Expr} \to \mathcal{P}(\mathsf{Var})$ *by rules like (where* $m, n : \mathsf{MethName}$*,* $s_1, s_2 : \mathsf{Stmt}$*,* $c : \mathsf{Expr}$ *and* $x : \mathsf{Var}$*):*

$$
\begin{array}{rcl}
\mathsf{mod}(m) & = & \mathsf{mod}(\mathsf{body}(m)) \\
\mathsf{mod}(s_1;s_2) & = & \mathsf{mod}(s_1) \cup \mathsf{mod}(s_2) \\
\mathsf{mod}(\mathsf{call}(n)) & = & \mathsf{mod}(n) \\
\mathsf{mod}(\texttt{if}\ (c)\ s_1\ \texttt{else}\ s_2) & = & \mathsf{mod}(c) \cup \mathsf{mod}(s_1) \cup \mathsf{mod}(s_2) \\
\mathsf{mod}(\texttt{try}\ s_1\ \texttt{catch}\ (E)\ s_2) & = & \mathsf{mod}(s_1) \cup \mathsf{mod}(s_2) \\
\mathsf{mod}(\texttt{set}\ x = c) & = & \{x\}
\end{array}
$$

Propagation of preconditions. Next, we define a function pre for propagating 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, and which does not contain any set annotations.

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

Core-annotations are synthesised for `beginTransaction` and `commitTransaction`. The annotations for `beginTransaction` are shown in Section 3.1 above, while `commitTransaction` requires `TRANS == 1` and ensures `TRANS == 0`. As we assume that `TRANS` is not modified by the code that precedes the call to `beginTransaction`, the only way the precondition of this method 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` (`TRANS == 1`) has to be established by the postcondition of `beginTransaction`, because the variable `TRANS` is modified by this method. Thus, preconditions containing only unmodified variables should be propagated. Propagating pre- or postconditions can be considered as passing on a method contract. Method bodies can only pass on contracts for variables they do not modify; once they modify a variable it is their duty to ensure that the necessary conditions are satisfied.

We assume the existence of a domain Pred of predicates using static ghost variables only, and function fv, returning the set of free variables.

Definition 2 (**pre**) *We define* $\mathsf{pre}\colon \mathsf{MethName} \to \mathcal{P}(\mathsf{Pred})$*,* $\mathsf{pre}\colon \mathsf{Stmt} \to \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Pred})$*, and* $\mathsf{pre}\colon \mathsf{Expr} \to \mathcal{P}(\mathsf{Var}) \to \mathcal{P}(\mathsf{Pred})$ *by rules like (where* $m, n : \mathsf{MethName}$*,* $s_1, s_2 : \mathsf{Stmt}$*,* $c : \mathsf{Expr}$*,* $V : \mathcal{P}(\mathsf{Var})$ *and* $x : \mathsf{Var}$*):*

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

In the rules defining pre on Stmt and Expr, the second argument denotes the set of static ghost variables that have been modified so far. When calculating the precondition for a method, we calculate the precondition of its body, assuming that so far no variables have been modified. 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. Finally, a set annotation does not give rise to any propagated precondition.

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 these functions 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 functions compute the following annotations.

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 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 whether there is a relation between our propagation functions and well-known program transformation calculi as the weakest precondition (wp) and strongest postcondition (sp).

The conceptual difference between our propagation functions and standard program transformation calculi is that, given method m our functions extract a method contract for m, while the program transformation calculi compute the proof obligations that, given all method contracts, allow to decide whether the implementation of m is correct.

A formal relation between pre and (a variant of) the wp-calculus can be established. Since we only consider ghost variables, we need to consider an abstract version of the wp: $\mathsf{wp}^{\#}$, which does not consider concrete variables. Most rules of this abstract wp-calculus are unchanged, but rules as for the conditional statement cannot consider the outcome of the conditional expression.

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

The abstract wp-calculus is sound, 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 3 *For any statement s, and any predicates P and Q, containing static ghost variables only, we have:*

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

Now we can prove a correspondence between pre and $\mathsf{wp}^{\#}$.

Theorem 4 (Correspondence) *For any statement s, its abstract weakest precondition is equivalent to the calculated precondition, in conjunc-*

tion with a universally quantified expression F.

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

This property formalises the conceptual difference described above: the function pre extracts the "external" part of the $\mathsf{wp}^{\#}$ (the method contract), while the quantified expression F corresponds to the "internal" proof obligations. The proofs of both properties proceed by structural induction. We believe similar equivalences can be proven for the function post and the sp-calculus. However, we are not aware of any adaptation of the sp-calculus to Java, therefore we did not study this.

4. Results

For several realistic examples of Java Card applications, we checked whether they 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 class `JCSystem` of the Java Card API. As explained above, a static ghost variable `TRANS` is used to keep track of whether there is a transaction in progress. Section 3.1 presents 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) TRANS == 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 if exceptions are thrown, they have to be caught within the 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 `Pin` in the standard Java Card API) with a precondition, requiring that no transaction is in progress.

```
/*@ requires TRANS == 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[7], and the PACAP case study[8], developed by Gemplus. Both examples have been explicitly developed as test cases for different formal techniques, 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 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 very easy to forget some of them.

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

```
void appExchangeCurrency(...) {
  ...
  /*@ exsures (Exception) TRANS == 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 is found automatically, without any problems.

5. Related Work

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

Proof-carrying code [18] provides another appealing solution to enforce security policies statically, but it 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.* [1, 4, 20]. Security automata provide another means to specify security policies and to monitor program executions. Different forms of automata (edit automata, truncation automata, insertion automata, *etc.*) have been proposed, to prevent or react against violations of security policies, see *e.g.* [19, 22, 13, 10]. 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 card's limited resources.

6. Conclusions

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, and a compiler that translates 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. 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 with temporal logic [21].

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 recursion and multi-threading.

Notes

1. `http://commoncriteria.com/`
2. `http://www.ercim.org/reset` and `ftp://ftp.cordis.lu/pub/ist/docs/ka2/`
3. Java Card is a dialect of Java, tailored explicitly to smart card applications.
4. `http://www.jmlspecs.org`
5. `http://www.atelierb.societe.com/`
6. `http://research.compaq.com/SRC/esc/Simplify.html`
7. `http://www.trusted-logic.fr`
8. `http://www.gemplus.com/smart/r_d/publications/case-study`

References

[1] 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.

[2] 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.

[3] 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(4):369–398, 2002.

[4] 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.

[5] 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. To appear in *Science of Computer Programming.*

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and

W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *ENTCS*. Elsevier, 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, 1998.

[10] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2003. Available as Technical Report 2003-1916.

[11] 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.

[12] 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.

[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] 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.

[15] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

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

[17] 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.

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

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

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

[21] 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.

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