Formal Verification of a Commercial Smart Card Applet with Multiple Tools*

Bart Jacobs[1], Claude Marché[2], and Nicole Rauch[3]

[1] University of Nijmegen, The Netherlands
[2] PCRI, LRI (CNRS UMR 8623), INRIA Futurs, Université Paris-Sud, France
[3] University of Kaiserslautern, Germany

Abstract. This paper presents a major Java source code verification case study that was carried out within the European VerifiCard project. It involves a realistic smart card applet from the company SchlumbergerSema that has been verified with several tools in parallel, in order to assess the state of the art in formal verification. The paper describes part of the verification—using the static checker ESC/Java2 and the verifiers Jive, Loop and Krakatoa—and reports on the experiences and outlook.

1 Introduction

The European project VerifiCard has been running for almost three years from 2001 to 2004. Its aim was to apply formal techniques in the area of Java-based smart cards, both at the byte code and source code level, in order to provide the smart card industry with tools and techniques for higher levels of certification—typically within the Common Criteria framework. The VerifiCard project did not focus on the use of one particular tool, but involved the parallel use of several tools in order to stimulate coordination and comparison. At the start of the project it was felt that the area was too young and immature to restrict to one particular technique or tool. This approach is reflected in the current paper.

Within this framework the VerifiCard participant SchlumbergerSema[4] has provided the project with source code of one of its applets, together with a set of security properties that it wants certainty about. It is a commercially developed applet that has been sold to customers, and therefore, several aspects of it (and of the required security properties) have to remain confidential. We thus refer to the applet simply as the "SLB applet".

For the specification of Java source code properties the language JML [2] is developing into a (*de facto*) standard—in part as a result of the activities within the VerifiCard project. This paper describes our experiences in applying tools for JML-based source code verification, namely Jive [14], Loop [9] and Krakatoa [13], as well as the static checking tool ESC/Java [12] (and its successor

* Funded by EU IST project IST-2000-26328-VERIFICARD, `www.verificard.org`

[4] Since jan. 1 2004 SchlumbergerSema's smart card group has become part of Axalto. Here we shall use the old name SchlumbergerSema that was used within the project.

ESC/Java2 [3]). The paper presents a short overview of the methodologies that the different verification tools are based upon, demonstrates how the tools have been applied to the case study, and what their respective achievements in the verification of the applet are—namely the detection of some bugs.

The paper is organised as follows. The SLB applet and its specification are introduced in Section 2. The subsequent five sections present the various tools that have been applied to the applet and the results achieved with them. Finally, Section 8 draws some conclusions.

2 Outline of the SLB Applet

This section presents some background information about the applet under investigation. The confidential nature of the applet imposes some restrictions on what can be said. However, these restrictions do not really affect the explanations of the scientific aspects of the investigation.

The applet has been provided by SchlumbergerSema "as is", that is without any additional documentation. The code itself does contain some rudimentary comments, but they are mostly concerned with specific details, and not with the big picture. We had to reconstruct that ourselves. This first step of the work, done by Hans Meijer and Bart Jacobs at Nijmegen, was to write a JML specification for the SLB applet. The specification was based purely on code inspection. The unclarities that emerged were discussed with SchlumbergerSema. At this stage the specification was only typechecked, and not verified. It was then distributed to the four verification teams, who adapted it to their own needs.

2.1 Structure of the Code

The main part of the SLB applet is a 2-dimensional array of bytes, representing a rudimentary file system. One immediate problem is that Java Card does not allow multi-dimensional arrays. The trick used in the applet is to define an array:

```
Object[] o_Records;
```

Within the applet, each object in `o_Records` will then store a byte array, called a *record*. This intention can be made explicit in a JML class invariant, as:

```
o_Records[i] instanceof byte[] &&
((byte[])o_Records[i]).length == record_length && ...
```

where `i` lies in an appropriate range. As stated, these records all have the same length, described here as a JML model field `record_length`. The first entry `o_Records[i][0]`, if non-zero, is used to indicate the length of the data in the record `o_Records[i]`.

All this makes the class invariant an essential part of the specification: it expresses the basic ideas that the applet developers had in mind. These ideas are not explicit in the code, nor in the documentation (that we have seen). At the same time, the class invariant is one of the most complicated parts of the

specification, and is sometimes needed to establish additional safety properties. The class invariant complicates the verification, because for each method it must be shown that the invariant is maintained. But in the different verifications described below, only the LOOP approach actually used the entire invariant. A detailed description of the class invariant is therefore postponed to Section 5.

The JML specification developed at this stage involved not only a class invariant, but also specifications for all methods in the applet class. A central method (as in any Java Card applet) is the `process` method: based on a case analysis using the APDU's class and instruction entries, it passes the card's incoming byte sequence (called APDU for application data unit) to one of the private methods `processSelectCmd`, `processReadRecord`, `processDeleteRecord`, `processPutData`, or `processAppendRecord`. Additionally, the `process` method involves a debug option. The private methods called by `process` typically start with low-level byte operations on the APDU's byte array, to perform appropriate checks or extract the relevant data. The associated JML specifications incorporate the corresponding checks, and are thus also low-level and rather verbose.

The following actual fields of the SLB applet are most relevant:

```
static final short OFF_DATA_IN_RECORD  = (short)01;
static final byte  LEN_RECORD_LEN_BYTE = (byte)01;
static final short MAX_LEN_OPTIONAL_DATA = (short)10;
static final short MAX_LEN_OPTIONAL_DATA_AND_HEADER
    = MAX_LEN_OPTIONAL_DATA + (short)3;
static final short SIZE_OPTIONAL_DATA_BUFFER
    = (short)3 * MAX_LEN_OPTIONAL_DATA_AND_HEADER;
// === Data entries
byte by_NbRecords;              // number of existing data entries
byte by_MaxNbRecord;            // max number of data entries
byte by_MaxSizeRecord;          // max size of a record
byte[] bya_OptionalData;
Object[] o_Records;
```

We introduced the following additional JML *model fields*, which are specification-only variables [10,11], typically used as convenient abbreviations or to provide a higher level of abstraction with respect to the actual fields.

```
/*@ public model final short OFF_DATA_IN_APDU;
  @   represents OFF_DATA_IN_APDU
              <- (short)ISO7816.OFFSET_CDATA;
  @ public model final short aid_off_in_data;
  @   represents aid_off_in_data
              <- (short)(OFF_DATA_IN_RECORD + 5);
  @ public model final short aid_off_in_apdu;
  @   represents aid_off_in_apdu
              <- (short)(OFF_DATA_IN_APDU + 5);
  @ public model short record_count;
  @   represents record_count <- (short)(by_NbRecords & 0xFF);
  @ public model short max_record_count;
  @   represents max_record_count
              <- (short)(by_MaxNbRecord & 0xFF);
```

```
  @ public model short record_length;
16 @ // with first byte describing length of subsequent records
  @    represents record_length
18 @    <- (short)(LEN_RECORD_LEN_BYTE+(by_MaxSizeRecord&0xFF));
  @*/
```

Various methods will be discussed below in separate sections.

2.2 Verification Aims

SchlumbergerSema provided us with a list of security properties to be checked. In this paper, we focus on the property describing error prediction. It is requested that "No exception other than an ISOException should be thrown at toplevel as a result of invoking an applet entry point". The following sections present the approaches of the different tools towards securing this property.

2.3 Production and Evaluation

Even though SchlumbergerSema is remarkably open in providing us access to actual production code, it is (understandably) secretive about many details, in order to protect its commercial interests. What we understand is that the applet under consideration has gone through the internal test and evaluation procedures within SchlumbergerSema before being sold to customers. We have no information about the nature and depth of these procedures, but we understand that there is no formal specification and verification involved. Hence, the verification methods described in this paper extend the internal approach at SchlumbergerSema, and their results are therefore of interest in order to possibly improve the current practice.

3 ESC/Java2 Approach

ESC/Java2 stands for *Extended Static Checker for Java Version 2*, a static checker which has originally been developed at Compaq SRC [12] and which is now being extended at the University of Nijmegen [3]. It has been applied to this case study in order to relate its checking approach to the verification approach followed by the other tools. ESC/Java2 is fully automatic (push-button). A drawback is that it is neither sound nor complete. Still, it provides valuable results which are demonstrated below.

3.1 Checking Technique

One approach of using ESC/Java2 is to add a JML specification to an applet and to apply ESC/Java2 to the specified applet. An alternative approach is to develop a specification interactively by invoking ESC/Java2 on an unspecified applet and by removing the warnings step by step by adding appropriate annotations. Remaining warnings can be confirmed or refuted by a verification

```
private void processDeleteRecord(APDU oApdu) {
  byte[] byaApdu = checkIncomingData(oApdu, true, false);
  byte byMode = MODE_DELETE_UNSET ;
  switch(byaApdu[ISO7816.OFFSET_P2]&0x07) {
    case 0x00: // --- modes delete by AID or refresh
      switch(byaApdu[ISO7816.OFFSET_P1]) {
        case 0x00:
          byMode = MODE_DELETE_BY_AID; break;
        case 0x01:
          byMode = MODE_DELETE_BY_REFRESH; break;
        default:
          ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
      } break;
    case 0x04: // --- mode delete by record number
      byMode = MODE_DELETE_BY_NUMBER; break;
    default:
      ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
  }
  if((byaApdu[ISO7816.OFFSET_P2]>>3) != 1)
    ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);
  byte[] byaRecordData = null;
  if (byMode == MODE_DELETE_BY_NUMBER) { /* ... */ }
  else if (byMode == MODE_DELETE_BY_AID) {
    for(byte i = 0; i < by_NbRecords; i++) {
      byaRecordData = (byte[])(o_Records[i]);
      if (Util.arrayCompare(byaApdu, ISO7816.OFFSET_CDATA,
            byaRecordData, (short)(OFF_DATA_IN_RECORD+6),
            byaRecordData[OFF_DATA_IN_RECORD+5]) == 0)
      {   /* ... */    }
    }
  } else if (byMode == MODE_DELETE_BY_REFRESH) { /* ... */ }
}
```

Fig. 1. SLB applet, method `processDeleteRecord` (slightly shortened)

system. This produces a lightweight specification which captures the applet's requirements to run without errors. In a subsequent step, the user can develop a functional specification. The approach presented here is not intended to produce a functional specification automatically.

3.2 The `processDeleteRecord` Method

This section introduces the method `processDeleteRecord` of the SLB applet. Figure 1 shows part of the method's source code. During the project, ESC/Java2 has been applied to the unspecified SLB applet in order to check the security property described in Section 2.2. We interactively generated a lightweight specification (see Fig. 2) which captures all conditions that are required by the

```
requires oApdu != null && oApdu._APDU_state == 1;
signals (ArrayIndexOutOfBoundsException u)
 (\exists short i; (0 <= i && i < by_NbRecords) ==>
  ((short)(ISO7816.OFFSET_CDATA +
  ((byte[])(o_Records[i])))[OFF_DATA_IN_RECORD+5])
 > oApdu.buffer.length));
signals (APDUException a) true;
signals (ISOException i)  true;
```

Fig. 2. Lightweight ESC/Java2 specification of `processDeleteRecord` method

```
/*@ requires src != null && srcOff+length <= src.length &&
  @          dst != null && dstOff+length <= dst.length &&
  @          srcOff >= 0 && dstOff >= 0 && length >= 0;
  @ signals (NullPointerException) src==null || dst==null;
  @ signals (ArrayIndexOutOfBoundsException) length < 0
  @         || srcOff < 0 || srcOff+length > src.length
  @         || dstOff < 0 || dstOff+length > dst.length;
  @*/
public static final /*@ pure @*/ byte arrayCompare(byte[] src,
        short srcOff, byte[] dst, short dstOff, short length)
```

Fig. 3. Method `Util.arrayCompare()`: Excerpt of JML specification.

method to run without exceptions. It is especially interesting that ESC/Java2 also points out some possible exceptions: `APDUExceptions` and `ISOExceptions`, which can be triggered from outside the applet. For example, the method `APDU.setIncomingAndReceive`, which is invoked in `checkIncomingData` (see Fig. 1, line 2), throws an `APDUException` if an I/O error occurred. These exceptions cannot be avoided by setting up requirements to the method, which is clear from the generated specification: it states that these exceptions are thrown unconditionally (see Fig. 2, lines 7 – 8). They must explicitly be caught in the applet or allowed at the top level.

Another reported exception is an `ArrayIndexOutOfBoundsException` which can occur in the call to Java Card's `Util.arrayCompare` method (see Fig. 1, lines 26 – 28). This method's last parameter accepts the length of the interval in which the two arrays are compared. If this length is negative or too large, or if one of the offsets passed as arguments are too large, an `ArrayIndexOutOfBoundsException` is thrown (see Figure 3). Section 4 treats the verification of this exception.

4 Jive Approach

Jive is a verification system developed at the Universities of Hagen and Kaiserslautern. It is an interactive Hoare logic theorem prover with a special graphical

user interface that uses an associated general purpose theorem prover (the user can choose between Isabelle/HOL [17] and PVS [16]). Internally, JIVE operates on Diet Java Card, a desugared subset of Java Card which is very close to the original language. For the verification, a Hoare-style programming logic is used. The pre- and postconditions of the Hoare triples are given in a predicate logic notation. The ability to read JML specifications is currently being added.

Proofs are performed by applying the Hoare rules—either manually or via so-called strategies which are Java programs that automatically apply several Hoare rules. Predicate logical statements over the pre- and postconditions (usually implications that stem from strengthening or weakening steps performed in the Hoare logic) are proven interactively in the associated prover.

One of JIVE's main strengths is its dedicated user interface. At all times, the user has full visual control over the whole proof process. A description of JIVE's architecture is given in [14].

4.1 Verification Technique

In order to prove the given security property with JIVE, one needs to provide a specification that describes the absence of any exception other than an `ISOException` or `APDUException` in the poststate of the `process` method.

Two techniques can be used to prove such a specification. The "standard" approach is to perform a *backward proof* by starting at the method implementation and the given specification and by developing a proof by applying the Hoare-logic rules in backward direction, e.g. by using a weak(est) precondition generation strategy [18]. This approach is useful if there is nothing known about the correctness of the code. But if one already suspects an error in the code, another approach comes in handy: the *assertion-supported forward proof*. In this approach an assertion is inserted into the code which can be pushed through the code [19], again e.g. by means of a weak(est) precondition mechanism. This technique will be explained in more detail below.

4.2 The `processDeleteRecord` Method

To verify the exception in the method `processDeleteRecord` which was indicated by ESC/JAVA2 (see Sect. 3), we use the *assertion-supported forward proof* that was briefly sketched above. This strategy allows us to insert an assertion just before a statement s which is suspected to throw an exception. This assertion is then propagated to the beginning of the method body by means of weak(est) precondition generation. There are two possibilities for the assertion: One can either insert a formula P_{ass} which is fulfilled by all states that do not raise an exception at s, or one can insert a formula $P_{ass}^{\neg}$ which is fulfilled by all states that definitely throw an exception at s. These formulae are propagated to preconditions P_{calc} and $P_{calc}^{\neg}$, respectively. How do these generated preconditions relate to the precondition P_{spec} which is part of the method specification?

Let us regard the first case. We know that if the precondition P_{calc} holds, then no exception can occur at s. If P_{calc} is a weakest precondition, i.e. if there are

neither loops nor method invocations in the code before s, then the implication $P_{spec} \Rightarrow P_{calc}$ must hold, otherwise we know that an exception can occur at s. If P_{calc} is not weakest, we can still try to prove the implication $P_{spec} \Rightarrow P_{calc}$. If it holds, we know again that no exception can occur, but if it does not hold, we do not know anything. We can only assume that P_{spec} may be too weak and should be strengthened (e.g. by adding P_{calc}). Thus, if we suspect that an exception occurs at s, and if there are loops or method invocations in the code before s, this may not be a good strategy because we might not get a usable result. Therefore, let us take a look at the second case. We know that each state which fulfills the formula $P^{\neg}_{calc}$ is guaranteed to raise an exception at s. Therefore, we need to find a state Σ that fulfills both the given precondition P_{spec} and the generated precondition $P^{\neg}_{calc}$, in other words, the formula $P_{spec}(\Sigma) \wedge P^{\neg}_{calc}(\Sigma)$ must be valid. The state Σ is called a *counterexample* to the assumption that no exception is thrown at s.

Regarding the `processDeleteRecord` method and the `ArrayIndexOutOfBoundsException` that ESC/Java2 reported, this means that the assertion $P^{\neg}_{ass}$, which guarantees for all fulfilling states that an exception is thrown in the subsequent statement, contains the following:

```
ISO7816.OFFSET_CDATA + byaRecordData[OFF_DATA_IN_RECORD+5]
                     > byaApdu.length
```

Here, it is easy to construct a counterexample as the array `byaRecordData` can contain arbitrary data (see Section 3); therefore, we use the second strategy to verify the possible occurrence of an exception. A more detailed description of the methodology can be found in [19].

5 LOOP Approach

After the initial phase described in Section 2 in which the specification was written, the verification with the LOOP tool started[5]. This resulted in a number of changes in the specification: First, only during actual verification is the specification really tested. A small number of subtle mismatches was found, leading to improvements in the specification. Second, the specification was optimised in its formulation to make it more suitable for verification. For instance, method calls were removed from specifications and replaced by explicit descriptions. This lowers the level of abstraction, but facilitates the verification with the LOOP tool.

In Section 2 we have already seen part of the classwide JML specification, namely the model fields. This part will be extended first with the invariant. Then, Subsection 5.2 will describe the specification and verification of one method in greater detail. The class invariant that we formulated is as follows.

[5] At Nijmegen the (original) ESC/Java tool was not applied to the applet, because the invariant (involving a complicated universal quantification) was too difficult to be handled by ESC/Java.

```
/*@ invariant
  @   bya_FCI != null  && bya_FCI.length == 23  &&
  @   LEN_RECORD_LEN_BYTE + DEF_MAX_ENTRY_SIZE
  @        >= aid_off_in_data + 1 &&
  @   LEN_RECORD_LEN_BYTE <= OFF_DATA_IN_RECORD &&
  @   0 <= by_NbRecords &&  0 <= by_MaxNbRecord  &&
  @   0 <= record_count && record_count <= max_record_count &&
  @   0 <= record_length && record_length > aid_off_in_data &&
  @   bya_OptionalData != null  && o_Records != null  &&
  @   bya_OptionalData.length == SIZE_OPTIONAL_DATA_BUFFER  &&
  @   o_Records.length == max_record_count  &&
  @   (\forall short i; 0 <= i && i < max_record_count ==> (
  @       (i >= record_count && o_Records[i] == null)  ||
  @       (i <  record_count && o_Records[i] != null &&
  @        o_Records[i] instanceof byte[] &&
  @        ((byte[])o_Records[i]).length == record_length &&
  @        ((byte[])o_Records[i])[aid_off_in_data] >= 0 &&
  @        ((byte[])o_Records[i])[aid_off_in_data] <
  @             record_length - aid_off_in_data &&
  @        (((byte[])o_Records[i])[0] != 0 ==>
  @            OFF_DATA_IN_RECORD +
  @                    (((byte[])o_Records[i])[0] & 0xFF)
  @             <= record_length))));
  @*/
```

The part of this invariant before the forall-quantification makes the bounds of the various (model) fields explicit. The forall-assertion describes the directory `o_Records` as an appropriate 2-dimensional array of bytes. It contains information that is not explicit in the code nor in the associated comments (provided by SchlumbergerSema), but was crucial for the proper understanding of the applet.

5.1 Verification Technique

The Loop tool is a compiler that takes Java+JML input and translates it to logical theories for the PVS [16] theorem prover. The generated PVS-theories incorporate the resulting proof obligations: each JML method specification becomes a PVS predicate that must be proven for the associated translated method. See [9] for a recent overview and further references. These translated method specifications incorporate the class invariants in their pre- and postconditions.

The actual verification happens in the backend theorem prover PVS. A user interacts with PVS by typing appropriate proof commands to guide the theorem prover. High level strategies are available, providing much automation.

Typical of the Loop tool is the use of a shallow embedding: Java programs become actual functions in PVS, on the basis of an underlying semantics. These functions can be evaluated symbolically, corresponding to execution in Java. The Loop verifications can take place in different ways.

– Symbolic evaluation (aka. semantical proof). This can be very efficient in proofs, but only works for simple Java program fragments, not involving iterations (such as while and for loops).
– Hoare logic [8]. In this way one can step through the Java code, at each point proving appropriate assertions. This provides more abstraction, but has the disadvantage that the user has to explicitly provide the intermediate assertions. However, this works well for larger method bodies, because they can be broken up into smaller manageable parts. The intermediate assertions can be written directly in JML, and are also translated by the LOOP tool.
– Weakest precondition reasoning [7]. It provides several (forward and backward) strategies to automatically prove assertions for code fragments. This works well for relatively small pieces of code, and can thus be combined efficiently with Hoare logic.

The fact that the entire Java+JML input is represented in PVS—and not broken up by the translation tool—has both advantages and disadvantages: an advantage of breaking up the proof goals inside PVS is that the rules for doing so (coming from Hoare and WP logic) are provably sound; a disadvantage is that size becomes an issue. Indeed, in the applet verification we found that the bottleneck in the verification was the theorem prover PVS. It has problems dealing with the large proof goals resulting from the larger methods (esp. `processAppendRecord` and `processDeleteRecord`). Currently, these scalability problems are being discussed and solved with the PVS development team.

One recent addition to the LOOP tool is the bitvector semantics [6] to handle Java Card's bounded arithmetic (`byte`, `short`) in a precise manner. This semantics is heavily used in the SLB applet case study, with its many low-level operations on bytes (such as masking and shifting).

5.2 The `processAppendRecord` Method

The `processAppendRecord` method has the following header, with explanation as provided by SchlumbergerSema.

```
/** Add a new record to the directory. The directory
  * is configured for a maximum number of directory entries
  * (set in the install command)
  * Entries once added can be removed, corresponding space can
  * be re-used again.  ...
  */
private void processAppendRecord(APDU oApdu) { ... }
```

The first step of the method is to put the byte array contained in the parameter APDU into a local variable `byte[] byaApdu`. The contents of this byte array are then examined, to see if they contain the right values to append a record (also contained in this byte array) to the directory (`o_Records`). If such an examination does not provide a positive result, an `ISOException` is thrown. These exceptions show up in the method's JML method specification, as `signal` clauses.

As part of these examinations an AID lookup is performed, of the form:

```
if (JCSystem.lookupAID(byaApdu, ADF_OFFSET,
                       byaApdu[ADF_LEN_OFFSET]) == null)
  ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
```

In the verification of this code fragment the method specification of `lookupAID` from the API class `JCSystem` is used. This specification looks as follows.

```
/*@ public behavior
  @   requires true;
  @   assignable \nothing;
  @   ensures true;
  @   signals (NullPointerException) buffer == null;
  @   signals (IndexOutOfBoundsException)
  @       offset < 0 || length < 0 ||
  @       offset + length > buffer.length;
  @*/
public static AID lookupAID( byte[] buffer, short offset,
                             byte length )
```

We see that an `IndexOutOfBoundsException` is thrown in case the parameter `length` is negative. During the verification of the `processAppendRecord` method we were not able to prove that this requirement holds for the byte value `byaApdu[ADF_LEN_OFFSET]`. Indeed, this requirement is not checked during the examinations at the beginning of the method. This is a bug: an APDU sent to the card can generate an exception that is not caught, and thus cause the application to halt, with status word "no precise diagnostic"[6]. However, this is not a security breach, because the applet remains in a consistent state, since the invariant still holds when such an exception appears.

A similar bug shows up later on in the code:

```
for(byte i=0 ; i<by_NbRecords ; i++) {
  byaRecordData = (byte[])(o_Records[i]) ;
  if ((byaRecordData[0] != 0x00)  // if not previously erased
    && Util.arrayCompare(byaApdu, ADF_OFFSET, byaRecordData,
        (short)(OFF_DATA_IN_RECORD+6),byaApdu[ADF_LEN_OFFSET])
    == 0)
    ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
```

We see that the `arrayCompare` method from Java Card's `Util` class is called. Its last parameter describes the length of the comparison interval (in two byte arrays). As we have seen in Section 3, if this length is negative or too large, an `IndexOutOfBoundsException` is thrown. This may actually happen in the applet because there are no appropriate checks on the value `byaApdu[ADF_LEN_OFFSET]` of the incoming APDU.

In conclusion, these two bugs appear in the JML specification of the method `processAppendRecord` in the fragment:

[6] We have been able to recreate this exception on an actual smart card running this applet. It was the first bug found in the course of this case study.

```
@  signals (IndexOutOfBoundsException) // This is a real bug!
2@       (oApdu.buffer[ADF_LEN_OFFSET] < 0     // for lookupAID
@     || oApdu.buffer[ADF_LEN_OFFSET] >=
4@          record_length - aid_off_in_data) // for ArrayCompare
```

These bugs were discovered because the proof could not proceed at the above mentioned program points.

6 Krakatoa Approach

6.1 Verification Technique

The Krakatoa tool [13] proceeds roughly as follows: from the JML-annotated source code of the API, and the program in consideration, and some method in that program, source scripts for the Coq proof assistant [20] are generated automatically. They provide a modeling of the Java memory heap for the program (depending on the collection of classes and fields); Coq definitions corresponding to the specifications of all constructors and methods of the program; a set of proof obligations, which are not in fact generated directly but by using an external tool called Why [4,5]; and a so-called *validation* of the method considered, that is a Coq program equivalent to it, of the form $\forall args\ \forall heap_{in}, pre \Rightarrow \exists result\ \exists heap_{out}, post$.

The application of the Krakatoa approach to the SLB applet case study took some benefit from the achievements of the Loop team. We actually used JML specifications they wrote both for the Java Card API and the applet, performing only a few minor changes to optimise the specifications for the Krakatoa tool as well as to avoid some *ad hoc* additions related to the Loop tool. We focus here on the `processPutData` method.

6.2 The `processPutData` Method

An excerpt of the annotated source of this method is shown in Figure 4. The changes (w.r.t. Loop specifications) we made are marked by appropriate comments. The first addition is in the precondition, line 2 of the listing. It appears to be needed for performing the proof of the normal postconditions, because otherwise `oApdu.buffer` could be modified, and everything would go wrong. After discussion with the Loop team, it appears they did not need this assumption because they were able to derive it from the JML class invariant of the `APDU` class: they put a very large minimal bound for the length of APDU buffers, so that these arrays can be provably different just because their lengths are different.

The second addition is in the modifiable clause on line 4 of the listing: mentioning `ISOException.systemInstance.theSw[0]` is needed because this location is modified when an `ISOException` is thrown. This was not required by Loop, because, as before, they simplified the JML specifications of the `ISOException` class, to get rid of such annoying and essentially irrelevant details.

The third change, line 7 of the listing, is the use of the pure method `getIndex` to replace the expression `>>4` used in Loop, which is OK only because of the

```
/*@ requires oApdu != null
  @          && byaOptionalData == bya_OptionalData
  @          && byaOptionalData != oApdu.buffer; //added
  @ modifiable byaOptionalData[*],
  @            ISOException.systemInstance.theSw[0]; // added
  @ ensures  (\forall short i; 0 <= i &&
  @             i < sh(oApdu.buffer[ISO7816.OFFSET_LC]) + 3;
  @       byaOptionalData[getIndex // instead of >>4 with LOOP
  @            (Util.getShort(oApdu.buffer,ISO7816.OFFSET_P1))
  @                  * MAX_LEN_OPTIONAL_DATA_AND_HEADER + i]
  @                  == oApdu.buffer[ISO7816.OFFSET_P1+i]);
  @ signals (ISOException e) true;
  @ signals (APDUException e) true;
  @*/
private void processPutData(APDU oApdu,byte[] byaOptionalData)
{ ... (some verification of input data skipped)
  byte byIndex = (byte)0;
  /*@ assignable ISOException.systemInstance.theSw[0];
    @ ensures 0 <= byIndex && byIndex <= 2 &&
    @         byIndex == getIndex(Util.getShort(oApdu.buffer,
    @                                 ISO7816.OFFSET_P1));
    @ signals (ISOException e) true;
    @*/
   switch (Util.getShort( byaApdu, ISO7816.OFFSET_P1 )) {
      case TAG_FCI_ISSUER_DISCRETIONARY_DATA :
          byIndex = 0 ; break ;
      case TAG_ISSUER_CODE_TABLE_INDEX :
          byIndex = 1 ; break ;
      case TAG_LANGUAGE_PREFERENCE :
          byIndex = 2 ; break ;
      default :
          ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
   }
   Util.arrayCopy( byaApdu, ISO7816.OFFSET_P1, byaOptionalData,
          (short)(byIndex * MAX_LEN_OPTIONAL_DATA_AND_HEADER),
          (short)(shLC+(short)3)); // +3 for TAG(2) and LEN(1)
}

/*@ ensures 0 <= \result && \result <= 2
  @  && (tag == TAG_FCI_ISSUER_DISCRETIONARY_DATA
                ==> \result == 0)
  @  && (tag == TAG_ISSUER_CODE_TABLE_INDEX ==> \result == 1)
  @  && (tag == TAG_LANGUAGE_PREFERENCE ==> \result == 2);
  @*/
private static /*@ pure @*/ byte getIndex(short tag);
```

Fig. 4. Excerpt of `processPutData` method, annotated

specific values of the `TAG_*` constants. For LOOP it is more convenient to have executable specifications because of its symbolic evaluation feature, but with KRAKATOA it is better to use more abstract specifications: we introduced a JML *pure* method `getIndex` to capture the logical behavior of the switch statement.

The last change, lines 17–21 of the listing, is an annotation for the switch statement, which is not standard JML but an extension allowed in KRAKATOA. This is because we met scalability issues: the amount of generated COQ source is quite large; and much effort has been made to reduce such problems. For instance, the annotation for the `switch` statement above reduced the size of the proofs with a factor four, because in some sense it factorizes all four cases of the `switch`. This clearly suggests that such an extension of JML is useful for theorem proving (indeed this extension is now under consideration).

7 Results and Experiences

In this case study, several exceptions were detected by the LOOP tool and ESC/JAVA2. These are `SystemExceptions`, `ISOExceptions`, `APDUExceptions`, `TransactionExceptions` and `ArrayIndexOutOfBoundsExceptions`. SchlumbergerSema's security policy only accepts `ISOExceptions` at the top level. But `SystemExceptions`, `APDUExceptions` and `TransactionExceptions` are inavoidable in the applet because they can be initiated externally (see Sect. 3.2 for an example). Here, the two tools clearly pointed out that SchlumbergerSema's security policy is too strict. But of course, the `ArrayIndexOutOfBoundsExceptions` are the most interesting exceptions. The LOOP and JIVE tools verified that these exceptions may occur in the `processAppendRecord` and `processDeleteRecord` methods because the applet uses data which has been passed from the card accepting device (CAD) to the applet in an APDU to index an array, and which has not been checked to be within the array bounds. Usually, this does not pose a problem, but if the applet is being attacked, malformed APDUs will be likely to be sent to it, which should not cause an exception to occur. To our knowledge, this exception cannot be used to exploit the applet nor to permanently damage the contained data. But the applet programmers would nevertheless be well advised to remove this cause of abrupt termination.

The `processPutData` method, as it is annotated in Figure 4, has been fully verified by the KRAKATOA tool. It was shown that no exception can be thrown other than `ISOException` or `APDUException`, and that the method satisfies its normal postcondition and preserves a class invariant.

Writing the specifications was a substantial part of the effort. Ideally, the specifications are already there, written by the programmers themselves. However, just writing specifications without checking them may lead to mismatches. Part of this case study's success is due to the support of JML by all tools except JIVE; only small changes had to be performed to adapt to KRAKATOA and ESC/JAVA2.

This case study has shown that the LOOP approach is capable of analyzing a non-trivial amount of Java Card source code (in the order of hundred's of

lines), and to detect errors that had previously gone unnoticed. KRAKATOA also demonstrated its ability to prove non-trivial properties of Java Card applets. This is clearly a success.

For KRAKATOA and LOOP, performance and scalability bottlenecks occurred within the backend theorem provers Coq and PVS, respectively. They slowed down the effort considerably. Improvements are currently under investigation.

The LOOP and KRAKATOA approaches require user interaction with PVS and Coq provers respectively, and knowledge of both the back-end prover and the Java modeling introduced by the respective tool is mandatory. But for an experienced user, proving the obligations generated from one method is not so hard: indeed, much of our time has been spent on improving the specification, the modeling, etc. to make proofs easier. Nevertheless, more automation of proofs is required via dedicated strategies/tactics in the back-end prover. Also, the option of further integration of handling of proof obligations by either an automated prover like Simplify (used by ESC/JAVA2) or an interactive prover (like Coq or PVS) is currently under study.

The validation of suspected errors (e.g. those detected by ESC/JAVA2) is not as costly as a full verification. Although a partial verification as e.g. performed by the JIVE tool does not guarantee that the areas that ESC/JAVA2 reports as correct are indeed error-free, it still is a recommendable approach to software verification because the tradeoff between cost and correctness that comes with it seems to be fair enough.

It is difficult to give exact measurements of time because this case study was used to optimize the tools, so that it is hard to tell what the pure verification time is. Additionally, the JML specifications had to be written in the beginning and improved during verification. All we can say that if no major problems arise, then the verification of individual methods is a matter of days.

8 Conclusions

The SLB applet case study described in this paper has pushed the development of the Java verifiers (JIVE, LOOP, KRAKATOA) to make them powerful enough to handle non-trivial source code. (The development from ESC/JAVA to ESC/-JAVA2 is independent.) Moreover, it has led to a unification in the area and a common focus on JML as specification language.

The semantical complexity of languages like Java and JML means that code verification is still very complicated and requires a substantial amount of user interaction. Nevertheless, the current tools are capable of detecting non-trivial bugs that were not spotted with conventional techniques (testing and code inspection). The case study has made it clear where the complexities are, and where improvements are needed. Here is a brief analysis.

Static checking with ESC/JAVA2 has become a very powerful push-button technique that is so simple to use that good Java developers should be able to use it. This means that they can write their own JML specifications together with

the implementation, and check it while they are developing code. The theorem-prover based verification techniques are still complicated and labor-intensive, so that they should be reserved for the cases that ESC/Java2 cannot handle, probably by people in research departments or outsiders. The positive feedback of SchlumbergerSema on this case study envisages an approach along these lines.

As an aside, separate from verification: SchlumbergerSema (and other companies like Gemplus) are enthusiastic about the use of JML. They see as one of the next challenges to relate their high-level security requirements to the relatively low-level specifications in JML.

It is in the nature of Java Card applets that they communicate with the outside world via byte sequences (in APDUs). As a result, much of the implementation and specification involves low-level details that are hard to write and easy to get wrong. A more abstract approach is badly needed, for instance based on Java Card RMI (see also [15]), as has recently been added in Java Card 2.2.

When we compare the verification tools used in this paper[7] we see two fundamental differences: First, Loop and Krakatoa are front-ends to theorem provers in which the interactive verification takes place on the translated program code. In contrast, the user of the Jive tool spends most of the time interacting with the original program code. Second, verifications with the Krakatoa and Jive tools decompose the proof goal into many separate verification conditions. The Loop tool generates one single proof obligation, which is decomposed (using provably sound rules) within the back-end theorem prover. The first approach leads to very many small obligations, and the latter to one big obligation. Both approaches may lead to too much data to manage. Among these alternative approaches there is (maybe unfortunately) not a single one that emerges as "best". An interesting perspective is to be able to make these tools, including ESC/Java2, cooperate.

In conclusion, we see a bright future for Java source code verification (based on JML specifications) in a 2-step approach, where theorem-prover based verification should be smoothly integrated with static checking, and be used to handle the difficult left-overs (like in [1]): Especially methods with loops or recursion or sources of remaining warnings are likely to still contain errors and are therefore good candidates for full verification.

Acknowledgements

Many people contributed to the design of the JML specifications and the proofs of obligations: we gratefully thank Cees-Bart Breunesse, Marek Gawkowski, Hans Meijer, Martijn Oostdijk, Christine Paulin, Erik Poll, Anne Schultz, Xavier Urbain, and Martijn Warnier. Further, we like to thank Boutheïna Chetali and Olivier Ly from SchlumbergerSema for their help and feedback.

[7] Besides the three verification approaches used in this paper there is also the deep embedding of Java into Isabelle/HOL [17] developed by the VerifiCard partner in Munich. Such a deep embedding is ideal for proving meta-properties (like type-safety), but too complicated for the verification of concrete programs.

References

1. C.-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. Technical Report NIII-R0316, University of Nijmegen, 2003.
2. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. 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)*, number 80 in Electr. Notes in Theoretical Computer Science. Elsevier, Amsterdam, 2003. `www.elsevier.nl/locate/entcs/volume80.html`.
3. ESC/Java2. Open source extended static checking for java version 2 (esc/java 2) project. Security of Systems Group, Univ. of Nijmegen `www.cs.kun.nl/ita/research/projects/sos/projects/escjava.html`.
4. J.-C. Filliâtre. The Why certification tool. http://why.lri.fr/.
5. J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4), 2003.
6. B. Jacobs. Java's integral types in PVS. In *FMOODS 2003 Proceedings*, Lecture Notes in Computer Science. Springer, 2003.
7. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 2004.
8. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In *Proc. FASE*, LNCS 2029. Springer, 2001.
9. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Techn. Rep. NIII-R0318, Comput. Sci. Inst., Univ. of Nijmegen. To appear in the LNCS proceedings of the International Symposium of Software Security (ISSS), Tokyo, 2003.
10. G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Business and Systems*. Kluwer, 1999.
11. G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual (draft). 2003.
12. K. Leino, G. Nelson, and J. Saxe. ESC/Java's User's Manual. Technical Report 2000-002, Compaq Systems Research Center, 2000.
13. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2), 2004.
14. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS00*, LNCS 276, 2000.
15. M. Oostdijk and M. Warnier. On the combination of Java Card Remote Method Invocation and JML. Technical Report NIII-R0321, University of Nijmegen, 2003.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
17. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.
18. N. Rauch and A. Poetzsch-Heffter. Predicate transformation as a proof strategy. In *Proc. 4th ECOOP Workshop: FTfJP, Tech. Rep. NIII-R0204*. Computing Science Department, University of Nijmegen, 2002.
19. A. Schultz. Verification of Java Card-applets. Master's thesis, Universität Kaiserslautern, 2003.
20. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, Jan. 2004. http://coq.inria.fr.