Implementing a Formally Verifiable Security Protocol in Java Card

Engelbert Hubbers, Martijn Oostdijk, and Erik Poll

Nijmegen Institute for Information and Computing Sciences, University of Nijmegen,
P.O. Box 9010, 6500GL, Nijmegen, The Netherlands
{hubbers,martijno,erikpoll}@cs.kun.nl

Abstract This paper describes a case study in refining an abstract security protocol description down to a concrete implementation on a Java Card smart card. The aim is to consider the decisions that have to be made in the development of such an implementation in a systematic way, and to investigate the possibilities of formal specification and verification in the design process and for the final implementation.

1 Introduction

Security protocols play a crucial role in pervasive computing, e.g. in ensuring authentication of different devices communicating over networks, or encryption of communications between these devices. There has been a lot of work on reasoning about security protocols over the past years, for example BAN logic [3] or state exploration based analysis using model checkers [12]. Still, there is a big gap between the abstract level at which such protocols are typically studied and the concrete level at which they are implemented. This is unsatisfactory since ultimately we are interested in properties of the concrete implementation.

This raises several questions: Which choices have to be made in the process of implementing a protocol and how do these affect the security of the implementation? Which properties of the abstract description also hold for a concrete implementation? What additional properties have to be worried about if for instance one of the agents participating in the protocol is running on a smart card and can therefore be subject to sudden loss of power at any moment?

Our aim is to investigate possible notations, techniques, and tools that can help in answering these questions. Rather than trying to make more precise what is meant by "the security of an implementation", the approach taken in this paper is to consider the kind of properties that we know how to specify and verify with today's tools and to see how these can contribute to secure protocol implementations.

This paper discusses a case study in refining a security protocol from the abstract description down to an actual implementation, where one of the agents is implemented on a smart card, using Java Card, a "dialect" of Java for programming smart cards. We investigate the choices that have to be made in this process by looking at formal descriptions of the protocol at different levels of abstraction and the properties we want to specify and verify.

The relation between the abstract protocol description and the final Java implementation is shown in Fig. 1. Our long term goal is to prove that the implementation of the protocol ensures the security properties we are interested in. For the moment we have to content ourselves with

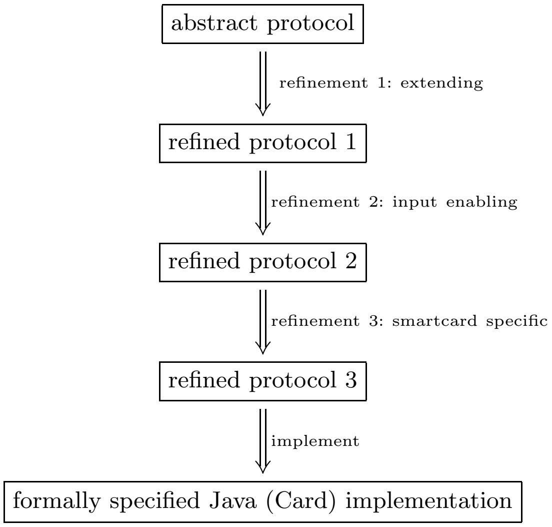

abstract protocol
refinement 1: extending
refined protocol 1
refinement 2: input enabling
refined protocol 2
refinement 3: smartcard specific
refined protocol 3
implement
formally specified Java (Card) implementation

Figure1. Refinement overview

- accurately specifying the different refinements that lead from the abstract protocol to the implementation and making the design decisions underlying these refinements explicit; this is done in Section 3.
- using the formal specification language JML (Java Modeling Language [7]) and two tools that support JML, namely the runtime assertion checker for JML [**?**] and the static checker ESC/Java [13], to ensure that the Java code correctly implements the final refinement of the protocol; this is done in Section 4.

Section 2 first describes the abstract protocol that we want to implement.

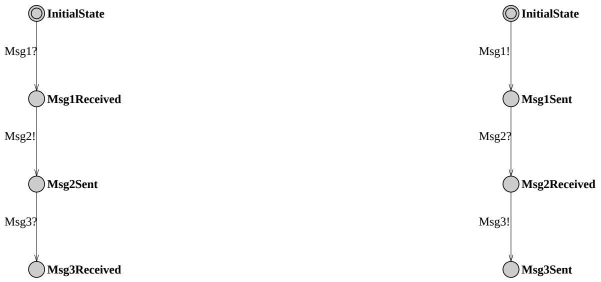
2 The abstract protocol

For this case study we use the protocol for bilateral key exchange (BKE) with public key described in [5, § 6.6.6]. This protocol allows two agents to agree on a session key. One of the agents will be implemented as a so-called smart card applet, i.e. a program executing on a smart card. It could, for example, be running on a mobile phone SIM or a credit card. The other agent will be an off-card application, communicating with the smart card applet via a smart card reader and possibly some network connection. The protocol consists of three messages. In conventional notation for security protocols, it reads as follows:

$$\begin{aligned} &1.\ B \rightarrow A : B, \{N_b, B\}_{K_a} \\ &2.\ A \rightarrow B : \{f(N_b), N_a, A, K\}_{K_b} \\ &3.\ B \rightarrow A : \{f(N_a)\}_K \end{aligned}$$

Here A and B are the two agents, N_a and N_b are the nonces (challenges) from A and B, and K_a and K_b are the public keys of A and B, respectively. The function f is a hash function and $\{\ldots\}_K$ denotes the data $\ldots$ encrypted using key K.

Figure 2 presents an alternative description of the protocol as two simple finite automata, one for each agent. (These automata are almost identical, but in the course of introducing more implementation details the automata for the two agents will become different.) Initial states are indicated by extra circles. All transitions are labeled with messages and either a ?, in case of an incoming message, or ! in case of an outgoing message. This is standard CSP notation.



Principal A

Principal B

Figure2. Abstract BKE protocol

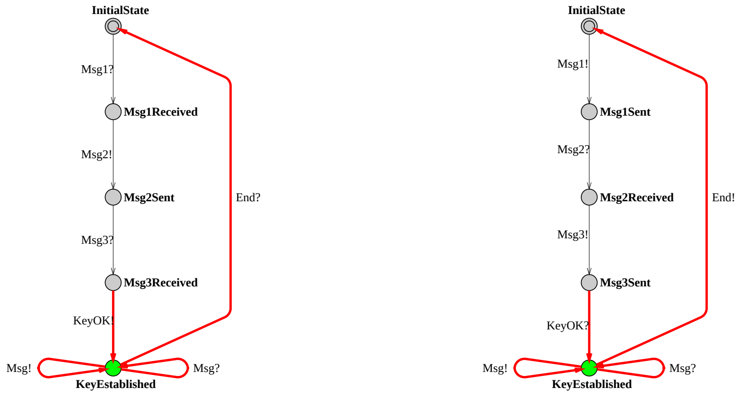
We used Casper [8] in combination with the model checker FDR2 [14] to prove that this protocol does indeed ensure mutual authentication and secrecy of the session key.

The abstract protocol only describes the initial handshake between A and B that establishes a session key. It does not say how this session key is actually used afterwards. For an actual implementation we do of course want to use the session key to encrypt subsequent communications between A and B. Therefore we extend the protocol as follows:

$$1.\ B \rightarrow A : B, \{N_b, B\}_{K_a}$$

$$\begin{array}{rl}
2.\ A \rightarrow B: & \{f(N_b), N_a, A, K\}_{K_b} \\
3.\ B \rightarrow A: & \{f(N_a)\}_K \\
4.\ A \rightarrow B: & \{\texttt{KeyOK}\}_K \\
5.\ B \rightarrow A: & \{\texttt{Msg}\ldots\}_K \\
6.\ A \rightarrow B: & \{\texttt{Msg}\ldots\}_K \\
 & \vdots \\
2n.\ B \rightarrow A: & \{\texttt{Msg}\ldots\}_K \\
2n+1.\ A \rightarrow B: & \{\texttt{Msg}\ldots\}_K \\
2n+2.\ B \rightarrow A: & \{\texttt{End}\}_K
\end{array}$$

Here `KeyOK` is an acknowledgment message sent to agent B in order to make sure that B knows he is allowed to send regular messages. This message is not really needed: if agent A would simply start to send a regular message using K, agent B could know that the suggested key has been accepted. The message `End` ends the session. This extension leads to the automata in Fig. 3.



Principal A Principal B

Figure3. Extended BKE state-transition diagram

3 Refinements

3.1 Anything that can go wrong . . .

Several things can go wrong during a protocol run:

1. We can get an *unsolicited message*. For example, agent A could be in its initial state and receive message `Msg3?` from agent B, whereas it is expecting `Msg1?`. In this case we say that the agents are "out of sync", which is something that could happen as a consequence of messages being lost. Note that Fig. 3 does not specify what should happen if this situation occurs.
2. An *exception may be thrown while processing expected messages*. For instance, an agent may receive an *incorrectly encrypted message*. For example, agent B could receive a first response message[1] of agent A that is not of the required form $\{f(N_b), N_a, A, K\}_{K_b}$. Note that the protocol as is doesn't provide any guarantee for message integrity.
3. An agent may fail to receive any message at all, due to a basic *failure of the communication channel* between the two agents.

Decision 1 *1. Receiving an unsolicited message ends the current session, i.e. an agent receiving an unsolicited message will move to its initial state. The only exception to this is if Agent A receives an unsolicited message* `Msg1?`*; in that case a new session will immediately start and Agent A will go to state* `Msg1Received`*.*

2. In case an exception is thrown, for instance when an agent receives an incorrectly encrypted message*, the agent will go back to its* `InitialState` *(and sends a special message* `XcB!` *back to the other agent).*

3. An agent noticing a failure of the communication channel *will go back to its* `InitialState`*.*

These decisions result in the new state-transition diagrams given in Fig. 4.

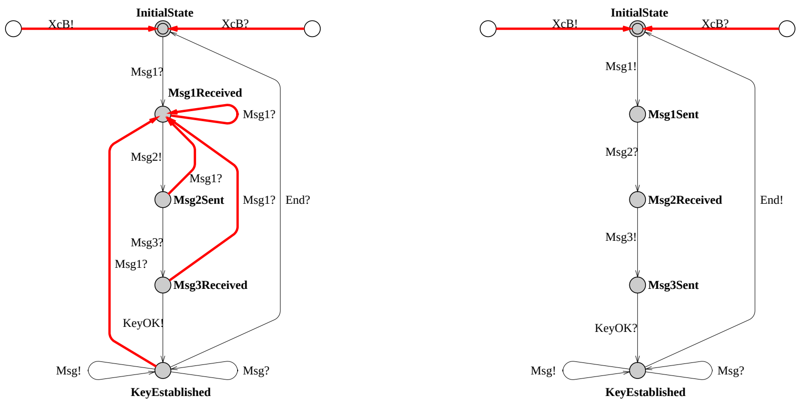
In order to keep these diagrams readable, two abbreviations are introduced. First, dummy states are introduced (indicated in Fig. 4 as the white states in the upper corners). Such a dummy state is to be seen as an abbreviation for all states. So, for example, from each state we have a transition to `InitialState` labeled `XcB?`. Without the upper right dummy state our diagram would be cluttered with five extra arrows.

Second, the label `XcB?` is an abbreviation for "all other messages", i.e. all possible messages that are not mentioned explicitly in the diagram. For example, consider the state `Msg2Sent` of agent A in Fig. 4. Two outgoing transitions, labelled with `Msg3?` and `Msg1?`, are drawn from this state. By the convention discussed above, there is also an implicit transition to `InitialState` labeled `XcB?`. Here `XcB?` now stands for any message other than `Msg3?` and `Msg1?`. So, from `Msg2Sent` we can move to `Msg1Received` by `Msg1?`, to `Msg3Received` by `Msg3?`, and to `InitialState` by any other message (i.e. `Msg?` or `End?` is received.

3.2 Initialization phase

Before the protocol can be used, some initialization has to be performed: each agent has to get its private and public key, and has to know the public key of the

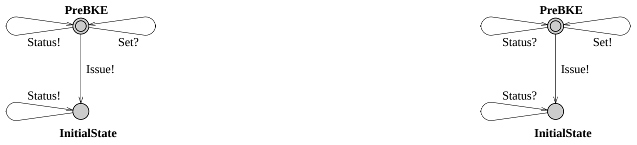
[1] We assume that the messages are labeled so it is clear which message is received. In the Java Card implementation this is typically done by means of the so-called instruction byte.



Principal A Principal B

Figure4. Extended BKE state-transition diagram with exceptional behavior included

other agent. All diagrams above start in the `InitialState` in which we assume the agents know all the relevant keys. In an actual implementation we will have to take care of this initialization phase.



Principal A Trusted Principal

Figure5. Initialization phase

To model this, the automata should be extended with the automata of Fig. 5. Notice how this affects the initial state of the automaton. The initialization phase involves communication with another agent, some trusted principal that tells agent A the public keys of the agents that the applet should be able to communicate with later on. We assume the initialization takes place in a trusted environment, and the smart card applet will ensure that initialization can only take place once by toggling a *personalization flag*. For the sake of the presentation, we will avoid talking about this `PreBKE` state in the diagrams below. However, in the JML specifications ofr the actual code we present later on it will turn up again.

3.3 Applet selection and persistent vs. transient memory

Java Card smart cards are multi-application smart cards, which means that several applets can be installed on one smart card. As a consequence, before we can communicate with a Java Card applet on a smart card, we tell the smart card which applet we want to communicate with. This is done by sending a `select` command to the smart card.

Decision 2 *If the card has been issued, the resulting state after a* `select` *command is always* `InitialState`*. If the card has not been issued, the state will be* `PreBKE`*.*

There are two kinds of memory available on a smart card: there is *persistent* memory, EEPROM, which keeps its value if the card has no power, and there is *transient* memory, RAM, which loses its value as soon as the power supply to the smart card is interrupted. By default, all objects are allocated in persistent memory, but an applet can choose to allocate some fields in transient memory. Such fields will be reset to default initial value, e.g. 0 for numerical fields, by the smart card operating system when the card powers up.

Decision 3 *All session-oriented information such as nonces, the state of the protocol, and the session key are kept in transient memory. The other information like the card's id, public keys and the personalization flag is stored in persistent memory.*

3.4 Card tears

A smart card applet can suddenly lose power due to a so-called *card tear*, e.g. when a card is removed from a card reader (or, in the case of a GSM SIM, when the battery of the mobile phone runs down). What should be the behavior of the smart card applet implementing agent A when a card tear happens? Of course, the applet will not be able to do anything after the card tear happens, as it will stop executing, but it can do something the next time the smart card powers up again and the applet is selected once again.

Decision 4 *It follows from Decisions 2 and 3 that after a card tear, the subsequent powering up, and selection of the applet, the new state is* `InitialState` *and of course all the transient memory is erased. This means that any session in progress is closed.*

Figure 6 later on makes these issues explicit. We have introduced two real states, `CardInserted` and `CardReady`, and one dummy state. As before, the dummy state can be seen as a union of all real states. So, the `CardTear` transition from this new dummy state to the state `CardInserted` can be taking from any state in the diagram.
The name `CardInserted` may seem strange. The `CardTear` transition does not mean that after a card tear this automaton goes immediately to `CardInserted`.

As soon as a card tear happens, the current session or the current automaton is stopped completely. Nothing will happen until the card is re-inserted again. In particular no transitions can be triggered during a card tear. Therefore this `CardTear` transition is only triggered at the re-insertion.

When the card is re-inserted the powering up takes place. In particular the terminal resets the card. The card responds to this by sending an Answer to Reset (ATR). After this the card is ready and waiting for a select command from the terminal.

3.5 Command-response pairs

Communication with a smart card uses the ISO7816 protocol, in which the terminal acts as a master and the smart card as a slave. The terminal sends *commands*, to which the card answers with a *response*. The messages sent between terminal and smart card are called APDUs (Application Protocol Data Unit) in ISO7816, which are just sequences of bytes. In our protocol agent A is implemented as a smart card and B as an application with access to the card terminal.

This means that all outgoing messages from A need to be triggered by an incoming message from B. And vice versa all incoming messages need to be followed by an outgoing message. Of course it would be possible to let agent A respond to all messages from B by sending a status word only. However, it seems more efficient to fill the response APDUs with the expected answers and a status word. For instance `Msg1` will be implemented as a command APDU and `Msg2` will be implemented as the corresponding response APDU.

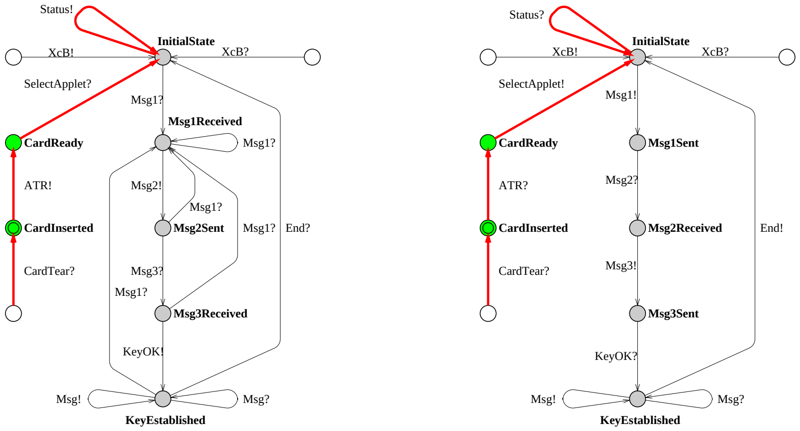
This choice has consequences for the states in the applet. After receiving `Msg1?` the applet will be in the state `Msg1Received`. However, before it tries to do anything else it will try to send back `Msg2!`. If this succeeds the resulting state will be `Msg2Sent`. If this fails the resulting state will be `InitialState`. In particular, this means the applet can never remain in state `Msg1Received` for any length of time, as the transition to this state –by a command APDU– will always be followed immediately by another transition out of this state –by the matching response APDU. This means that it is no longer possible for any incoming unsolicited message to be received in this intermediate state `Msg1Received`. Technically this means that some of the arrows in the diagram for agent A can now be omitted. However, because we used dummy states in our diagrams we do not see this in our representation. Only the interpretation of the notion of dummy state is weakened slightly.

Decision 5 *We need one extra response APDU* `Status!`*: it will act as a response to* `SelectApplet?`*,* `End?` *and* `XcB?`*.*

Below are the command-response pairs that may occur.

Commands	`Msg1?`	`Msg3?`	`Msg?`	`End?`	`XcB?`	`SelectApplet?`
Responses	`Msg2!` `XcB!`	`KeyOK!` `XcB!`	`Msg!` `XcB!`	`Status!`	`Status!`	`Status!`

The way we implemented this affects the meaning of the message `Msg!`. Although it still appears in the diagram for principal A, it is now restricted to being used as an answer to the `Msg?` message. The applet will no longer be able to send `Msg!` on its own initiative! Furthermore, adding the necessary `Status!` response on the applet side implies also adding `Status?` on the terminal side.



Principal A Principal B

Figure6. Extended BKE state-transition diagram with exceptional behavior, card tear recovery, and paired APDUs included

Obviously the changes we have discussed here have an impact on the protocol as we presented it earlier. We need to add a single line:

$$2n+3.\ A \rightarrow B : \{\mathtt{Status}\}_K$$

Fig. 6 shows the corresponding state-transition diagram.

4 Using JML

This section considers the use of the Java Modeling Language (JML, see [7]) and tools that support JML to ensure that our Java (Card) implementation correctly implements the final refinement of the protocol discussed in the previous section. JML is a specification language that can be used to formally specify the behavior of Java programs.

4.1 JML specifications

Fortunately, state-transition diagrams describing the required behavior of the agents can easily be translated into JML. (The only problem is how to deal with specifying the card tear mechanism; this is discussed in Section 4.4.)

In order to describe the rest of the diagram in Fig. 6 we use two instance variables. Namely the instance variable personalized, stored in persistent memory, that keeps track of whether the card has been issued or not, and the instance variable bke_state[0], stored in transient memory, that records the state in the protocol. (In Java Card only arrays can be allocated in transient memory. Therefore bke_state is a transient array of length 1.)

The diagrams of Figures 5 and 6 can be expressed by a combination of JML invariants, constraints, and method specifications.

In JML, as is usual, *invariants* are predicates which should be established by the constructor and preserved by the methods, i.e. invariants should hold after an invocation of a constructor, and both before and after any method invocation). E.g. the invariants in the JML specification below give the possible values of the applet state bke_state[0], and the relation between this state and the personalized flag.

In JML *constraints* are relations that should be respected by all methods, i.e. the pre- and post-state of any method invocation should be in the relation specified by a constraint. E.g. the constraint in the JML specification below specifies that once a card has been personalized, it will remain personalized forever.

```
/*@ invariant
  @   bke_state[0] == PRE_BKE || bke_state[0] == INIT ||
  @   bke_state[0] == MSG1_RECEIVED || bke_state[0] == MSG2_SENT ||
  @   bke_state[0] == MSG3_RECEIVED || bke_state[0] == KEY_ESTABLISHED;
  @ invariant personalized <==> (bke_state[0] != PRE_BKE);
  @ constraint \old(personalized) ==> personalized;
  @*/
```

Based upon the automata in Fig. 5 and 6, and given these constraints and invariants, it is easy to give JML specifications for the methods that specify the desired flow of control. Below we give the method specification of the process method.

```
/*@ behavior
  @ requires true;
  @ ensures (\old(bke_state[0]) == PRE_BKE ==>
  @  (bke_state[0] == \old(bke_state[0]) || bke_state[0] == INIT));
  @ ensures (\old(bke_state[0]) == INIT ==>
  @  (bke_state[0] == \old(bke_state[0]) || bke_state[0] == MSG2_SENT));
  @ ensures (\old(bke_state[0]) == MSG2_SENT ==>
  @  (bke_state[0] == \old(bke_state[0]) ||
  @   bke_state[0] == KEY_ESTABLISHED ||
  @   bke_state[0] == INIT));
  @ ensures (\old(bke_state[0]) == KEY_ESTABLISHED ==>
  @  (bke_state[0] == \old(bke_state[0]) || bke_state[0] == INIT ||
  @   bke_state[0] == MSG2_SENT ));
  @ signals (Exception) (\old(bke_state[0]) == PRE_BKE ==>
  @   bke_state[0] == PRE_BKE);
  @ signals (Exception) (\old(bke_state[0]) != PRE_BKE ==>
  @   bke_state[0] == INIT);
```

```
@*/
public void process(APDU apdu) throws ISOException
```

The method specification consists of a precondition (indicated by requires), postconditions for normal termination (indicated by ensures), and postconditions for abnormal termination (indicated by signals). All ensures clauses should be considered together as a logical conjunction. The first ensures clause is specifically for the initialization phase. The signals clauses should be considered as a conjunction as well. There are two lines here because we need to make a distinction between whether a card has been issued or not.

Below we give the specification of the receiveMsg1 method. On the top level we see new keywords also and exceptional_behavior. The also splits the specification into two parts. The distinction is based upon the value of bke_state[0] on entry of the method. If this state is PreBKE the card has not been issued yet and hence an exception *must* be thrown and the resulting state will still be PreBKE. In any other state we allow this message to come in. If the receiving succeeds, the resulting state will be Msg1Received, otherwise an exception is thrown and the applet will go to InitialState. Note that this method has as a postcondition that the state will be Msg1Received. This state does not appear in the specification of process. This is because the process method will always call sendMsg2 and this method will always change the state –either to Msg2Sent or to InitialState– before process terminates.

```
/*@ behavior
  @ requires bke_state[0] != PRE_BKE;
  @ ensures bke_state[0] == MSG1_RECEIVED;
  @ signals (Exception) (bke_state[0] == INIT);
  @ also
  @ exceptional_behavior
  @ requires bke_state[0] == PRE_BKE;
  @ signals (Exception) (bke_state[0] == PRE_BKE);
  @*/
  private void receiveMsg1(APDU apdu) throws ISOException
```

4.2 Runtime checking with the JML tool

The JML runtime assertion checker [**?**] takes as input Java source files annotated with JML specifications. It augments the source files with runtime checks based on the JML specifications so that all invariants, constraints, pre- and postconditions are checked at runtime and any violation result in a special exception being thrown.

We used this tool to check the Java Card code of our applet against our JML specification. To do this we could not execute the code on an actual smart card, but we had to use a smart card simulator instead. (The reason for this is that the runtime assertion checker uses some Java API classes that are not part of the more restricted Java Card API, and are consequently not available on Java Card smart cards.) The smart card simulator we used was Sun's Java Card Workstation Development Environment (JCWDE).

In this setup we were able to find quite a few mistakes in our JML specification. Typically these errors were caused by forgetting to specify some of the implicit transactions from Fig. 6.

4.3 Static checking with ESC/Java

ESC/Java [13], the ‘extended static checker’ for Java is a tool developed at Compaq SRC for automatically checking JML-annotated code[2]. The tool uses a theorem prover to automatically verify that assertions in Java code are correct, without any user interaction. The tool is neither sound nor complete, i.e. it can warn about possible violations of assertions that cannot happen, and fail to warn about possible violations of assertions that can happen. Still, the tool is very useful for debugging Java(Card) code and formal specifications, as it can provide quick feedback pointing out possible mistakes, especially since, unlike for runtime assertion checking, no test scenarios are needed for using ESC/Java. ESC/Java has already been used with great success in debugging Java Card source code, see [4].

The kind of JML specifications we have written are well within the range of what ESC/Java can handle. Running ESC/Java on our annotated applet pointed out several mistakes. For example, our initial JML specifications did not allow for the fact that on any point in the protocol the `process` method may receive a select APDU, in which case the applet reverts to the `INIT`. In particular we did not find this mistake when we used the JML runtime assertion checker, simply because this possibility wasn’t included in our test scenarios. On the other hand runtime assertion checking can deal with the actual contents of APDUs being sent, which is something ESC/Java cannot.

Note that ESC/Java requires ESC/Java specifications of the API classes used by the applet, such as the `javacard.framework.APDU`. Here we used the formal JML specifications for the Java Card API version 2.1.1, discussed in [11,10] and available on-line via `http://www.verificard.org`.

4.4 Card tears and invariants

Card tears cause a special problem for invariants specified in JML. JML allows an invariant to be temporarily broken during the execution of a method. But if a card tear should happen at such a point, this could cause problems later, when the applet continues its operation in a state where some of its invariants are broken.

Such problems will not show up in the runtime checking with the JML tool, as the simulator we use does not simulate card tears, and will also not show up in the static checking with ESC/Java, as ESC/Java has been designed for Java and does not take the peculiarities of Java Card into account.

There are three ways in which problems with a temporarily broken invariant at the moment of a card tear can be avoided:

[2] Actually, the specification language ESC/Java uses is a ‘dialect’ of JML.

1. The invariant could become re-established the next time the smart card powers up again, as a result of the resetting of all transient memory to its default initial value.
2. The invariant could become re-established when the applet is selected again, namely if the applet itself takes care to restore the invariant when it receives its select APDU.
3. Finally, Java Card offers a so-called *transaction mechanism*. By invoking special methods from the Java Card API one can turn any sequence of Java instructions into an atomic action. When the smart card powers up, the smart card operating system will roll-back to the pre state of such a sequence of instructions if it has been interrupted by a card tear.

For every invariant in our JML specification we *manually* checked the following properties:

- Is this invariant ever temporarily broken during a method?
- If so, is the invariant re-established by one of the three mechanisms mentioned above ?

Because the Java Card transaction mechanism is not supported by the tools ESC/Java and JML, our applet does not use this functionality, and hence we never have to rely on the third way to re-establish invariants after a card tear listed above.

5 Conclusions

We started with an abstract description of a security protocol, for bilateral key exchange, for which we had earlier used Casper [8] and FDR2 [14] to prove its correctness. This paper describes how, based on a few explicit design decisions, we refined this protocol in several stages to an actual implementation, where one of the agents is implemented as a Java Card smart card applet. It should be stressed that our interest here is not the outcome of our decisions, but rather the decision making-process itself. The JML language was used for a formal specification of the Java Card code. It turns out that this specification can be systematically derived from the finite automaton in Fig. 6, the final refinement of the abstract protocol, that includes card tears and the handling of all possible exceptions that may arise. We have checked that the implementation meets these specifications, using runtime assertion checking with the JML tool, and doing static checking using ESC/Java.

Implementing a security protocol on a smart card involves some non-straightforward decisions: decisions 2, 3, 4 and 5.

Both static checking and runtime checking turn out to be good methods to check JML specifications of applets. Although they will not notice all problems, they certainly help to improve the code and the specifications. Both methods have their own advantages and disadvantages, therefore it is a good idea to use both ESC/Java and JML, although they are not specific Java Card tools.

In [9] a list of important security properties for Java Card applets is presented. The JML specifications for our applet include several of these properties, e.g. ‘service control’ and ‘error prediction’, but not all of them. Some properties, most notably ‘secure information flow’, have been taken into account while coding, but cannot be specified in JML in any convenient way.

Future Work

The most important open question is how we can prove that the refinements of the protocol ensure the same security properties that the original protocol establishes. We can think of two ways to do this.

We already used the FDR2 model checker to prove that the original abstract protocol establishes authenticity and secrecy. We also tried to use it to check the same properties of our refinements of the protocol. However, FDR2 had problems with the recursion in these refinements, which allow infinite traces. It might be a good idea to spend some more time in defining this recursive CSP model. Maybe we could get FDR2 to prove that our security properties are valid for traces of a certain maximum length, which would give us some more confidence that the refinements preserve the properties of the original protocol.

Alternatively, we could investigate properties of the refinements between the different automata. For example, one obvious property that holds for the refinements is trace inclusion. Another property is that any trace that leads to the state `KeyEstablished` in the final refinement Fig. 6 will have a tail that leads to the state `Msg3Received` in Fig. 2. Intuitively, such properties seem sufficient to guarantee that our refinements preserve the security properties of the original protocol. However, we have not formally proved this yet.

It would be interesting to experiment with model checkers, such as Uppaal [15], to check interesting properties of the automata describing the protocol. (In fact, we already used Uppaal to draw all the diagrams in this paper.) Even if we are not able to check typical security properties, we might for example be able to rule out the possibility of deadlock.

Maybe it is worthwhile to develop new versions of ESC/Java and JML in order to cope with features specific to Java Card, such as the transaction mechanism and the possibility of card tears.

After static checking with ESC/Java and runtime checker for JML, the next step would be to prove the correctness of the JML specification with respect to the Java Card implementation using the LOOP tool [6], in combination with the theorem prover PVS [**?**]. See [2] for a discussion of examples of such correctness proofs. Such a formal verification would provide a much higher level of assurance that our implementation does indeed meet its specification than the checking with ESC/Java. However, it would also require much more effort, as such verifications are very labor-intensive. For the current JML specifications, we do not think such an additional effort would be worthwhile, as ESC/Java seems reliable enough when dealing with these kinds of properties.

References

1. A. Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Dep. of Comp. Science, Iowa State Univ., 2000.
2. C.-B. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In *9th Algebraic Methodology and Software Technology (AMAST)*, LNCS, St. Gilles les Bains, Reunion Island, France, September 2002. Springer-Verlag, Berlin.
3. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proc. Royal Soc.*, Series A, Volume 426:233–271, 1989.
4. N. Cataño and M. Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe (FME)*, volume LNCS 2391, pages 272 – 289, Copenhagen, Denmark, July 2002. Springer-Verlag, Berlin.
5. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0. http://www-users.cs.york.ac.uk/ jac/drareviewps.ps, 1997.
6. Bart Jacobs et al. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
7. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Dep. of Comp. Sci., Iowa State Univ., 2002.
8. G. Lowe. Casper: A Compiler for the Analysis of Security Protocols, http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/.
9. R. Marlet and D. Le Metayer. Security properties and java card specificities to be studied in the secsafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 23 2001. Available from http://www.doc.ic.ac.uk/ siveroni/secsafe/docs.html.
10. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.
11. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.
12. P.Y.A. Ryan, S.A. Schneider, M.H. Goldschmith, G. Lowe, and A.W.Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison Wesley, 2001.
13. Compaq SRC. Extended Static Checker for Java, http://www.research.compaq.com/SRC/esc/Esc.html.
14. Formal Systems. FDR2, Failures Divergence Refinement, http://www.formal.demon.co.uk/FDR2.html.
15. Uppaal. An integrated tool environment for modeling, validation and verification of real-time system modeled as networks of timed automata, extended with data types. Available at http://www.uppaal.com.