Voting using Java Card smart cards: A case study

Cees-Bart Breunesse, Bart Jacobs, and Martijn Oostdijk

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

Abstract. This paper documents a case study in the development of a Java Card application and describes some related security issues. An implementation of a simplified electronic voting system is presented, in which a server residing on the Internet is able to authenticate registered voters based on information encrypted by a Java Card smart card. Although this work is done in the context of a larger project, in which verification is the main goal, the paper focuses on the technical aspects of the implementation. Yet, it also addresses more general issues, such as how to design and analyze secure Java Card applets and host applications.

1 Introduction

As more and more transactions are performed over the Internet, the problem of *authentication* becomes increasingly important. Common authentication protocols rely on public key cryptography, involving keys of typically 1024 bits. Such keys have to be distributed to the different parties over an untrusted channel without compromising the protocol. Moreover, the keys have to be stored on a computer since humans are not particularly good at remembering large prime numbers. Smart cards are the perfect carriers for such large keys. Since the cryptographic algorithms are performed on the cards themselves, the keys never have to leave the card. This paper discusses how smart cards, and more precisely *Java Card* smart cards, can be used to implement a (drastically simplified) electronic voting system.

Generally speaking, Java Card [4] is neither a subset nor a superset of the Java language [9]. On the one hand, Java Card lacks features like multithreading, garbage collection, and multidimensional arrays and primitive types such as `float` and `long`. On the other hand, the Java Card run-time system is different from Java's run-time system, since it has to deal with persistent memory. Furthermore, the standard libraries that come with Java and Java Card differ in the classes and interfaces they offer the programmer. Nevertheless, the two languages are similar in style, and clearly very closely related.

The voting case study discussed in Section 5 consists of a 100% pure Java solution. What makes Java or Java Card so suitable for this problem? Features like object-orientation, exception handling, and a large library of predefined classes (network, cryptography, security, GUI) make it possible to implement high level protocols relatively easily. Moreover, Java's semantics are reasonably well understood. Verification of the correctness of Java classes is possible using the tools developed within our group, the LOOP group in Nijmegen [11], but also with other tools such as Bandera [5], ESC [7], and Jive [15].

The goals of this paper are twofold. On the one hand, it attempts to record the technical details encountered so far in developing a concrete smart card application. Section 2 provides some technical details on the setup and the cards and libraries that were used. Section 4 explains how to implement security protocols based on strong cryptography in Java. Section 5 presents the electronic voting system case study. The case study on electronic voting is documented, so that it is clear what protocol was chosen, which APIs were used, etc. On the other hand, the paper studies the state-of-the-art in design and analysis of safe and correct smart card application. Section 3 discusses some of the issues in designing Java Card applets and host applications. Section 6 presents the conclusions and contains a discussion about future research. An important question that is left for future research is: Is verification of correctness of Java applications enough, or is a high-level formalism needed to verify the security protocol?

2 Enabling Technologies

This section describes the steps necessary to set up a computer suited for developing Java based smart card applications. The description is based on experience with some concrete smart cards and associated hardware, which are discussed in detail. It is not meant to be an exhaustive account of currently available solutions, nor is it intended to be a comparison of the cards that are discussed. The purpose of this section is merely to give a flavor of the kind of preparatory work involved in developing a Java Card application.

A Java Card application consists of a card-resident part (the *applet*) and a card-external part (the *host application*). In fact, a card can contain many applets belonging to different Java Card applications thus making the card "multipurpose". Once the card has been issued, the applets run on the Java virtual machine on the card itself. The host application runs on an ordinary computer, and may therefore be implemented in an arbitrary language. However, for the purposes of this paper, we restrict ourselves to Java based host applications. In this way, both the host application and the applets can be developed on the same computer.

Our setup for developing the application consists of a Linux workstation with card terminal attached to the serial port. It is assumed that Java SDK 1.3 is installed on this machine. Both the applet and the host application use additional libraries. Fortunately, this merely involves copying some `jar` files into the `$JDK/jre/lib/ext` directory. The interfaces to Java libraries are known as *Application Programmer's Interfaces*, or APIs.

The first step is to enable Java to communicate with the serial ports. For this purpose, Sun provides the JavaComm API [20]. However, before JavaComm can be installed, the RXTX package [12], which connects the Linux serial port device to JavaComm, has to be installed, this is because the Sun implementation of JavaComm only supports Windows or Solaris machines. To let JavaComm know about RXTX, the following line is put in the `javax.comm.properties` file (this file should be in `$JDK/jre/lib`).

```
Driver=gnu.io.RXTXCommDriver
```

To allow any user (not just `root`) to use serial ports, this user should be a member of the `uucp` group. Also the permissions of the `/var/lock` directory and the `/dev/ttyS?` devices may cause problems.

Host application development requires installation of additional APIs. If the Java Card application uses cryptography, for example, the JCE API may be very useful; this API is discussed in Section 4. Obviously, the host application needs the ability to send APDUs (Application Protocol Data Units) to the card. The Open Card Framework (OCF) [16], discussed in more detail below, provides this ability. Furthermore, the host application may need to load new applets onto the card, or in some other way manage the applets on the card. The procedure to do this differs per card. One initiative to standardize this is the *Visa Open Platform* specification [8]. All cards considered below implement applet loading through sending appropriate APDU commands to the card.

Applet development requires the Java Card API to be installed on the developer's machine. The classes in the Java Card API can be mere stubs[1], since the applet will use the on-card implementation of the Java Card API, once it is loaded onto the card. If the applet is to use additional APIs present on the card, for example a GSM API or the Visa Open Platform API, installation of (stubs for) those APIs on the development machine is also necessary. All cards considered below comply to the Java Card standard. Note, however, that there are different versions (the current Java Card version is 2.1.2).

Using Java for the implementation of both applet and host application has a number of advantages over a mixed language solution. First of all, switching between different APIs when writing either the applet or host application is difficult enough writing in the same language. Furthermore, the OCF [16], which enables us to access a whole lot of different terminals and cards, is written in Java. Alternatively, we could take the PC/SC architecture, which is available for Windows and Linux machines.

To access the card terminal, the OCF API may be used. Alternatively, some cards come with their own API for accessing the card, for instance the OneWire API below. Note that the standards for both card and host application are relatively new, and these frameworks and APIs are continually being improved. See Figure 1 for an overview of the APIs that we use.

[1] The Java Card reference implementation of Sun may be used to test applets as it contains a full implementation of the API which runs as a smart card simulator on top of a Java VM.

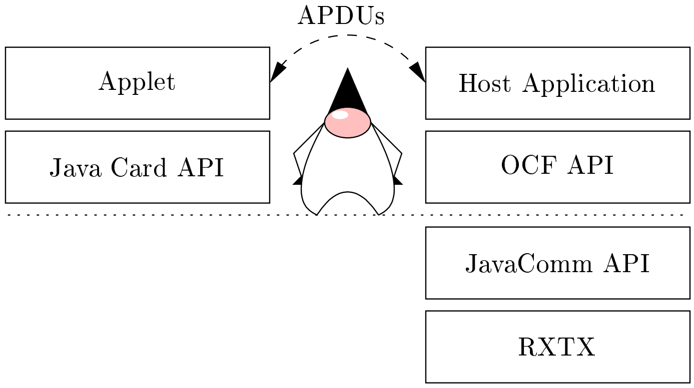

Fig. 1. The applet, the host application, and the different APIs.

Once the necessary APIs are installed on the machine, the OCF API should be configured to enable the host application to access card terminals attached to the system. The OCF plays the role of middleware by abstracting away from concrete card terminals and smart cards. Its configuration is specified in the `opencard.properties` file which should be in `$JDK/jre/lib`.

There are many different terminals on the market. We restrict ourselves to the Gemplus GCR410 model card terminal and iButton adaptors. To tell the framework about these terminals, suitable OCF drivers are installed and the following lines are added to the `opencard.properties` file.

```
OpenCard.terminals=\
   com.gemplus.opencard.terminal.GemplusCardTerminalFactory|\
      gcr410|GCR410|/dev/ttyS0 \
   com.ibutton.oc.terminal.jib.iButtonCardTerminalFactory|\
      iButtonAdapter|iBUTTON_PORT_TYPE_SERIAL|/dev/ttyS1
```

A central notion in OCF is the *card service*. A card service specifies the functionality of a related set of smart cards. The `PassThruCardService` is about the simplest card service available. The only functionality it provides is the sending of APDUs to the card. This means that all Java Card smart cards can be addressed by this card service. Adding the following lines to the `opencard.properties` file makes it possible for a host application to use the `PassThruCardService`:

```
OpenCard.services=\
   opencard.opt.util.PassThruCardServiceFactory
```

The idea behind OCF card services, however, is that the host application does not send APDUs itself, instead the card service should offer methods which take care of low-level APDU communication with the card. For example, many host applications need the ability to load new applets onto the card which requires a card service such as the Visa Open Platform card service which we are developing.

So far, we have a working OCF system and a working card terminal, what about the cards themselves? The following list is a very small subset of all available cards on the market. Note that we do not draw any conclusions on which card is the best, but merely sum up the cards we fiddled with:

- The Dallas Semiconductor Java Card iButton. These smart "cards" have a different appearance than the ones below, because the Java Card chip is contained in a small button-like metal casing. The Java Card iButtons are members of the larger iButton family of devices, they comply to Java Card 2.0 with extensions to access the cryptographic coprocessor. The iButtons are more tamper resistant than conventional smart cards. An iButton contains a battery which is used to completely erase its memory when an attempt is made to open the casing. On the host application side Dallas offers two APIs for managing the applets on a Java Card iButton. The non-OCF-based OneWire API and the OCF-based iButton API. The former has a broad support for all (also the non-Java) iButtons, the latter is specifically meant for Java Card iButtons.
- The Gemplus GemXpresso 211IS. These cards support Java Card 2.1 and Visa Open Platform. On the host application side one can use the Gemplus software kit (OCF based). A drawback to this particular version of the card (the *international* version) is that it does not support the RSA security algorithm due to export regulations.
- The Schlumberger Palmera Protect V2. These cards have similar specifications as the Gemplus cards, they also support Java Card 2.1 and Visa Open Platform. Unfortunately, the Schlumberger software kit, needed to load applets onto the card, only works under Windows. To overcome this we wrote our own Visa Open Platform card service for OCF.

Because of the many different cards and card terminals it is difficult to develop host applications and applets that work on all platforms, despite standardization efforts like OCF, Visa Open Platform, and Java Card. With all the different setups, the communication and speed of the CPU on the card remains a bottleneck. Choosing a card and card terminal will probably be an economic choice.

3 Designing Java Card Applications

In true object-oriented fashion, a Java Card applet extends the `Applet` class in order to add specific functionality to an already existing base functionality. In Java Card the programmer overrides the `process` method which receives data from the host application in the form of a byte array in a specific form, called an APDU (Application Protocol Data Unit). The applet receives Command APDUs telling the applet what to do. The applet is able to respond by sending Response APDUs which have a different form. One could say that all functionality of the applet revolves around the process method, ignoring the fact that different applets can talk to each other using the shareable interface between them. Thus, Java Card applets are small pieces of code. Moreover, since they are restricted to non-threaded behavior, they are well suited for formal verification. Specification of their behavior in terms of bytes is easy.

The following program fragment shows a typical process method in a Java Card 2.0 style applet.

```
public void process(APDU apdu) throws ISOException {

   byte[] buffer = apdu.getBuffer();

   if((buffer[ISO.OFFSET_CLA] == SELECT_CLA) &&
      (buffer[ISO.OFFSET_INS] == SELECT_INS)) {
         return;
   }

   short bytesLeft = (short) (buffer[ISO.OFFSET_LC] & 0x00FF);
   short bytesTotal = bytesLeft;
   short readCount = apdu.setIncomingAndReceive();

   switch( buffer[ISO.OFFSET_INS] ) {

      case INS_SIGN: ....
   }
}
```

A first glance at the program text learns that programming Java Card applets forces the programmer to descent to the low level of bits and bytes. The method first pulls out the bytes from the APDU, it then performs an input check on them, it writes some data at specific offsets in local variables and then does a case distinction on the `buffer[ISO.OFFSET_INS]` byte which represents the instruction or "what-to-do byte". The process method is the only place where APDUs enter the applet[2]. The code shown above is the most common way to handle APDUs. Implementing the same routine for every applet can become somewhat tedious eventually. Future versions of Java Card may support Remote Method Invocation (RMI) which is more friendly to the programmer, but also removes the property of having only one entry point for APDUs and therefore is less clear as to which method may or may not be called by the host application. Furthermore, as the (serial) communication is already slow, RMI makes the communication overhead even worse.

Since the applet has to make use of limited resources, it tends to be simple. This is a good thing because, as noted before, it makes formal verification feasible. Still, simple things also need interaction with real life users. This is where the host application comes in. The user interface is an important part of the

[2] Properties of the APDU mechanism have been formally specified in [19].

host application because it defines the user friendliness of the smart card. User interfaces thrive on threads to prevent them from locking up when communication with the smart card is slow or waiting. Its threaded organization enables the host application to give feedback to the user and, at the same time, maintain communications with the smart card. Host applications might also keep a copy of the state of the smart card because communication is expensive in time. All these factors combined make the host application a complex piece of software, not quite suited for formal verification, and vulnerable to attacks. Since we want to build secure smart card applications, both the host application and the applet should be taken into account.

It is important to distinguish, within the design of the host application, the parts that contribute to the high-level communication protocol. Formal verification of those parts is possible only if they can be isolated from the parts that are not relevant to the protocol. A well-designed host application should also be robust against attacks. Reasoning about security is discussed in the next sections. For an example of an ad hoc protection against virus attacks, see the VSVPP discussed in Section 5.

Extra care should be taken in the design of the host application in order to allow formal verification. As the host application is probably the weakest link of the protocol, it is a likely target for an attack. The example implementation in Section 5 illustrates how to design such a host application.

4 Java and Cryptography

Cryptography can be employed in protocols to achieve security properties such as *authentication*, *confidentiality*, and *integrity*. The problem of where to safely store private keys is solved using smart cards. Section 5 describes a case study involving such a security protocol. This section is meant as an introduction to the use of cryptography in Java Card and Java.

By default, Java SDK 1.3 contains classes dealing with security related concepts: The Security API. For example, the Security API contains the `Signature` and `MessageDigest` classes. Unfortunately, due to export regulations, the Security API does not provide classes to do cryptographic encryption. The *Java Cryptography Extension* (JCE) is an API which should be downloaded separately and which supplements the Security API with strong cryptography. Fortunately, due to recent relaxations in the above mentioned export regulations, the JCE API will be integrated in the upcoming release 1.4 of Sun's Java SDK.

The main class of the JCE API is the `Cipher` class. To create instances of the class, it has a factory method, that is parameterized with a text string specifying the algorithm, for instance `"DES"` or `"RSA"`. A `Cipher` object may be used for encryption and decryption of messages, or for wrapping keys (which is, at least for DES and RSA, the same thing as encryption). In combination with the `MessageDigest` class it can also be used to add signatures to messages, although the Security API provides a more general `Signature` class. The `Cipher` and `Signature` classes need to be initialized with cryptographic keys which are supplied by factory classes in the Security API.

The JCE API is really just an interface that is to be implemented by different *providers*. Sun has its own JCE provider, which will be integrated in Java SDK 1.4. The current version of Sun's JCE provider does not provide RSA ciphers. In the case study in Section 5 the BouncyCastle JCE provider [14] is used.This provider is added to the list of providers by adding the following line to the `java.security` file in `$JDK/jre/lib/security` directory:

```
security.provider.1=\
   org.bouncycastle.jce.provider.BouncyCastleProvider
```

Table 1 lists some cryptographic operations from the Security and JCE APIs. It also introduces some high-level security protocol notation in the last column. A full introduction to such notation is outside the scope of this paper, instead see, for example, [1]. This notation is used in Section 5 to specify the security protocol used in the case study.

Table 1. Notation for Java cryptographic operations.

Operation	Class	Notation
Encrypting	`Cipher`	$\{m\}_k$
Decrypting	`Cipher`	–
Wrapping keys	`Cipher`	$\{k_1\}_{k_2}$
Unwrapping keys	`Cipher`	–
Signing	`Signature`	$\{m\}_{\mathsf{priv}_P}$
Verifying signatures	`Signature`	–
Message Digesting	`MessageDigest`	$\partial(m)$
Concatenating	–	$\langle m_1\rangle_{m_2}$

On the card side, the cryptography API which is part of the Java Card 2.1 API is similar in style to the cryptographic classes provided by the JCE API. Of course, Java Card only provides a fixed set of algorithms. There is also the separation of the API into a security package (classes for signing messages and computing message digests) and a cryptography package (classes for strong cryptography). Some of the cards suffer from the export rules. For instance, as mentioned in the previous section, the international version of the GemXpresso card does not feature RSA encryption because of these rules. The Java Card 2.0 iButtons do not contain a cryptographic API comparable to JCE, however their API allows direct access to the cryptographic coprocessor which features operations such as modular arithmetic and exponentiation. Implementing RSA encryption with these primitives is quite simple. However, during implementation of the case study discussed in Section 5, some very low-level programming was required to ensure that the iButtons and the Java Card 2.1 cards compute similar signatures. For example, both the iButtons and the Palmera cards require that the public exponent of the RSA key pair is one byte, a property not all JCE providers ensure. Another example, it seems that the SHA message digest algorithm on the Java Card iButton differs from the SHA message digest in JCE in that the resulting bytes have to be swapped in groups of four to get the same result.

5 The Case Study: Electronic Voting

The case study presented in this section is an implementation of a drastically simplified electronic voting protocol. In the near future, casting a vote in an election will no longer require a voter to visit a physical voting station. Instead, a vote can be cast from the voter's PC at home. The PC sends the vote information over the Internet to a central server which keeps track of the election result. There are a number of potential problems against which the voting protocol should be protected. Only three problems are discussed here.

First, the vote information should remain a secret, this means that only the server should receive the vote information that is sent by the voter. This property is known as *confidentiality*. Obviously, the vote information can be encoded using public key cryptography before it is sent over the Internet.

Second, only registered voters should be allowed to cast their votes. The result of the voting process should reflect the choices of the registered voters, and the registered voters only. These properties are known as *authentication* and *integrity*. This also can be implemented using public key cryptography. Every registered voter receives a smart card by ordinary mail. The smart card cryptographically signs the vote before it is sent to the server.

Third, not even the voter's PC should be trusted. Or, put differently, the host application is not part of the *trusted computer base*. Think, for example, of a virus which infects the voter's PC prior to the election period. On election day the virus becomes active and may circumvent the cryptographic protections described above. The virus might, for instance, read the vote and send it to some third party, thus violating the confidentiality property. Worse yet, it might send a completely different vote to the smart card for signing. The server would accept the vote since it was signed by a legitimate smart card. This would violate the integrity of the election result. A solution to this problem uses a personal vote list, on which the list of parties is permuted in a way that is unique for the voter's id. Each of the voter's choices is thus encoded, and the voter enters the *vote code* into the GUI, rather than the choice itself. A virus might intercept the vote code, and start a "vote thread" with a changed vote code, but it cannot intensionally change the outcome of the election. Moreover, if the vote codes are chosen from a large enough set of codes, it is extremely unlikely that it choses a vote code which is valid for the particular voter id. This simple idea is called the *Voter-Side Virus-Protection Protocol* (VSVPP). Of course, the server now needs to maintain a database relating voter ids to vote lists.

Note that many potential problems are not addressed in the above analysis. For example, *denial of service attacks* are ignored. Also, the server is completely trusted. The latter problem can perhaps be solved using a protocol which distributes trust over several servers, such as described for example in [6].

The security protocol implemented in the voting case study involves three principals: The applet (A), the host application (H), and the server (S). For a complete security analysis perhaps the voter, the virus, and potential adversaries on the Internet should be considered as well. Figure 2 shows how the protocol works. First, A computes the signature s, based on the vote v and A's private key priv_A. Next, H encrypts the vote v and the signature s using a freshly generated session key, based on the server's public key pub_S, resulting in the message m. The message m is sent over the Internet to the server, which is able to decode it, since it has the private key priv_S and A's public key pub_A. The diagram shows

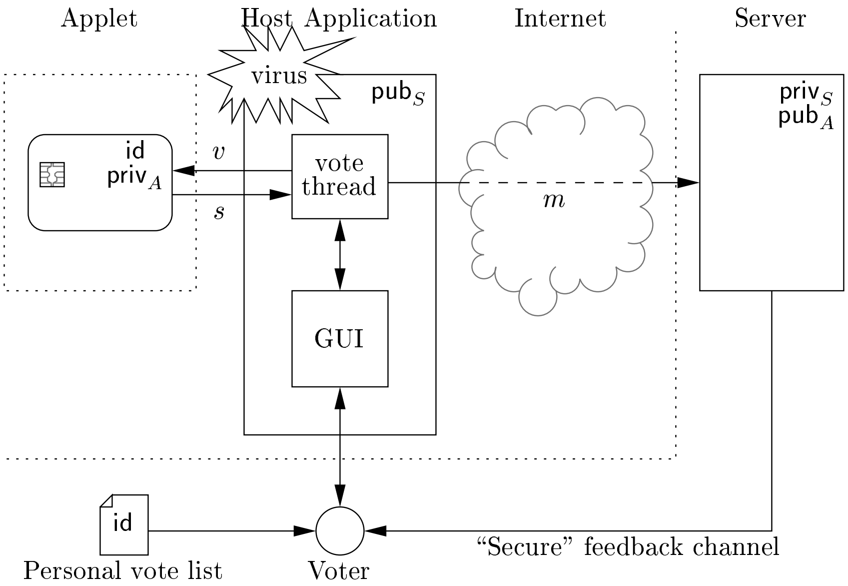

Fig. 2. Overview of the voting system.

where the public keys pub_A, pub_S are needed. It should be assumed, however, (as is common) that all involved principals know all public keys. The trusted computer base includes the applet and the server, but not the host application.

The keys pub_A and priv_A are loaded only once onto the smart card during initialization. Once the keys are present, the applet can be used to sign packets.

The packet v that gets sent to the applet for signing is composed as follows:

$$v = \begin{array}{|l|c|}\hline \mathsf{id} & 10 \\ \hline \text{vote code} & 2 \\ \hline \text{email} & 40 \\ \hline \text{gsm} & 8 \\ \hline \text{padding} & 4 \\ \hline \end{array}$$

In total v contains 64 bytes. The applet first computes a 20 byte SHA message digest, and then encrypts it with its 128 byte (or 1024 bit) RSA private key priv_A to get the 128 byte signature s.

Rather than use pub_S to directly encrypt the vote information, first a session key des is generated. The session key des is *wrapped*, that is encrypted with, pub_S and sent to the server. Such a session key is normally used to help ensure non-repudiation. In this case, however, it is used because the message $\langle v\rangle_s$ gets too large to be encrypted by a 128 byte RSA key. The vote v and signature s are thus encrypted with des to obtain the message m which gets sent to the server.

The server S unwraps the session key des and decodes m to obtain v and s. The signature s is verified by decrypting s using pub_A, computing the SHA message digest of v and comparing the two results. In case they are equal, S updates the election result accordingly and sends an acknowledgment message back to H. Additionally, S sends the information it received over a "secure" feedback channel to the voter. In the implementation this channel is implemented by sending email or an SMS message (through the GSM network) to the voter.

In conventional security protocol notation, as introduced in Section 4, the total protocol reads as follows:

$$A \to H : \mathsf{id} \tag{1}$$
$$H \to A : v \tag{2}$$
$$A \to H : \{\partial(v)\}_{\mathsf{priv}_A} =: s \tag{3}$$
$$H \to S : \{\mathsf{des}\}_{\mathsf{pub}_S} \tag{4}$$
$$H \to S : \{\langle v\rangle_s\}_{\mathsf{des}} =: m \tag{5}$$
$$S \to H : \mathsf{ack/deny} \tag{6}$$

Security protocols like this, once formalized, can be analysed using formal methods such as BAN logic [3]. An interesting issue, not addressed here, is how to bridge the gap between the formal version of the security protocol and the actual implementation.

6 Conclusions and Future Work

The main result of the work described in this paper, apart from a working smart card application, is insight into the issues involved in designing such an application. For instance, one of the unforeseen problems is that, although the applet part of the application remains quite simple, the host application part becomes increasingly complex. Although the host application is usually not part of the trusted computer base, one typically wants it to be correct anyway. The issues involved in designing such an application have become much clearer during the implementation work.

Another lesson we learned is, that even though standards like Java Card, the OpenCard Framework, and Visa Open Platform provide interoperability and bring with them the promise of reusable code, in practice the development of cross platform smart card applications still requires a lot of hard work. This is due to slight differences in the available interfaces. Especially working with different cryptographic libraries (both on-card and off-card) can be a burden.

As for verification, the work described here was done in the LOOP group in Nijmegen. This group focuses on the development and use of the LOOP tool [11], a front end for theorem provers such as PVS [17]. The tool translates Java source files into mathematical PVS models. Goal is formal verification of Java applications whose behavior is specified using the modeling language JML [13] which essentially is a Hoare-like annotation language based on Java expressions with some extras. Some academic, yet concrete, examples have already been verified [10, 2, 19]. Recently, we started looking at Java Card within the context of the European *VerifiCard* project, which is coordinated from Nijmegen. Java Card offers unique opportunities to show that formal verification is feasible. Since smart cards only provide limited resources, the applications on the smart card have to be very small and memory efficient. This puts us back in the 8-bit-processor era.

The voting case study is not formally verified in this way. Since the case study implements a security protocol, it seems that high-level security protocol verification is more appropriate, as opposed to low-level verification of correctness. While high-level security analysis is an important new area of interest, we believe that abstracting away from the low-level details is a bad idea, as attacks occur on all levels. We feel that low-level verification of correctness should serve as a basis for high-level security protocol verification. How the two can be combined, especially in the context of automated theorem proving such as in [18], remains an interesting problem for future research.

Acknowledgments

The work described in this paper was carried out in the context of the European Union project IST-2000-26328-VERIFICARD. The voting demo was originally developed as a demonstration for the ICT kenniscongres held on 6 and 7 September in the Netherlands, which provided us with funds. Schlumberger/CP8 generously donated some smart cards to experiment with.

References

1. Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley and sons, Inc., 2001.

2. J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of Java Card's application identifier class. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of *LNCS*, pages 137–150. Springer-Verlag, 2001.
3. Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. Technical Report SRC-RP-39, DEC Systems Research Center, February 1989.
4. Zhiqun Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley, June 2000.
5. James Corbett et al. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, June 2000.
6. Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. Multi-authority secret-ballot elections with linear work. In *Advances in Cryptology - EUROCRYPT'96*, volume 1070 of *LNCS*, pages 72–83. Springer-Verlag, 1996.
7. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, December 1998.
8. Global Platform. Open platform card specification version 2.1, June 2001. Available at `http://www.globalplatform.org/`.
9. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
10. M. Huisman, B. P. F. Jacobs, and J. A. G. M. van den Berg. A case study in class library verification: Java's vector class. Technical Report CSI-R0007, Computing Science Institute, University of Nijmegen, March 2000.
11. B. 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.
12. Keane Jarvi. RXTX: Serial and parallel I/O libraries supporting Sun's CommAPI. Web site, 2001. Available at `http://www.rxtx.org/`.
13. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specifications for Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
14. Legion of the Bouncy Castle. Legion of the Bouncy Castle. Web site, 2001. Available at `http://www.bouncycastle.org/`.
15. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description, 2000. Available at `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
16. OpenCard Consortium. Welcome to OpenCard! Web site, 2001. Available at `http://www.opencard.org/`.
17. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999. Available at `http://pvs.csl.sri.com/`.
18. Lawrence C. Paulson. Inductive analysis of the Internet protocol TLS. In Bruce Christianson, Bruno Crispo, William S. Harbison, and Michael Roe, editors, *Security Protocols*, volume 1550 of *LNCS*, pages 1–12. Springer-Verlag, 1998.
19. Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.
20. Sun Microsystems, Inc. Java communication API. Web site, 2001. Available at `http://www.javasoft.com/products/javacomm/`.