USENIX Association

Proceedings of the 5th Smart Card Research and Advanced Application Conference

San Jose, California, USA
November 21–22, 2002

Secure Method Invocation in JASON *

Richard Brinkman
Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, the Netherlands
brinkman@cs.utwente.nl

Jaap-Henk Hoepman
Department of Computer Science, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
jhh@cs.kun.nl

Abstract

In this paper we describe the Secure Method Invocation (SMI) framework implemented for JASON, our Javacard As Secure Objects Networks platform. JASON realises the *secure object store* paradigm, that reconciles the card-as-storage-element and card-as-processing-element views. In this paradigm, smart cards are viewed as secure containers for objects, whose methods can be called straightforwardly and securely using SMI. JASON is currently being developed as a middleware layer that securely interconnects an arbitrary number of smart cards, terminals and back-office systems over the Internet.

1 Introduction

JavaCard[1] [Che00] technology makes it possible to develop software for a smart card using a high level language: JAVA. This technology is platform independent, it can handle multiple applications (each running securely within its own sandbox) on one smart card, post-issuance applications can be added to it and it is compatible with international standards like ISO7816 [ISO7816].

In fact, the JavaCard platform brought high level, Object Oriented Programming (OOP) to the smart card developer. Unfortunately, the OOP paradigm is only applied to the software within the smart card itself: invoking methods implemented by objects on the smart card still requires the developer to send commands to the smart card using Application Protocol Data Units (APDU's) [ISO7816], which have to be processed and transformed into method calls 'by hand'.

It would be much more natural to view an object stored on a JavaCard as a remote object, accessible through a remote method invocation mechanism. In fact, if we look at a smart card application at a higher level of abstraction, we basically see a large collection of interconnected objects. Some of these objects are stored in back offices, others in terminals or PC's and many more stored securely on millions of smart cards. This network is highly dynamic: smart cards are usually offline, and only connect to the network when they are inserted into a terminal (or when they connect to a terminal over a wireless interface in the case of contactless cards). Much more importantly, this network needs to be highly secure. Access to certain objects should be restricted, and the confidentiality and authenticity of the communication between the objects has to be guaranteed.

Hartel *et al.* [HJF95] pose that a smart card should be seen as a processing element

*Id: javacard-smi.tex,v 1.10 2002/09/23 06:04:03 hoepman Exp

[1] `http://java.sun.com/products/javacard`

rather than a storage element (as is traditionally done). In our opinion these views are not contradictory at all, but rather supplement each other nicely in the *secure object store* paradigm. In this paradigm, smart cards are viewed as secure containers for objects, whose methods can be called straightforwardly and securely using Secure Method Invocation (SMI). We are currently developing the Javacards As Secure Objects Network (JASON) platform as a middleware layer (on these smart cards, terminals, PC's and back office systems) to support this paradigm. By simplifying the communication with a smart card, and by providing extensive support to secure this communication, JASON aims to greatly simplify the development of smart card applications.

In this paper we will describe the JASON Secure Method Invocation (SMI) scheme. In this scheme, a JASON definition file (JDF) (resembling a JAVA interface with some additional keywords) is used to specify the access conditions on methods of an object. It also specifies how the parameters of a method call and the result should be protected when transmitted between caller and callee. The JDF is compiled into a *stub* (used by the caller to set up a connection with the object and to call its methods) and a *skeleton* (used by the callee to accept incoming method invocation requests and to handle the security requirements). The big advantage is that the smart card application developer only needs to specify the security requirements, but does not have to implement the security protocols himself. This is done automatically, given the requirements.

The remainder of this paper is organised as follows. We first present related research in the next section. Then, the main requirements for the JASON platform are presented in Sect. 2. The design (in terms of the application programmers view on JASON) is given in Sect. 3. Section 4 discusses the architecture and the way the JASON SMI is actually implemented, while Sect. 5 presents a small example of using JASON to implement a basic electronic purse. Finally, conclusions and issues for further research appear in Sect. 6

1.1 State of the art

Itoi *et al.* [IFH00] add security to the Internet infrastructure for smart cards developed by Guthery *et al.* [Gut00, GBPR00] and Rees *et al.* [RH00], adding the Simple Password Exponential Key Exchange (SPEKE) protocol and using the DNS as a location independent naming scheme for the smart cards involved. These aspects will be taken into account in the networking and naming part of the JASON platform.

Hagimont and Vandewalle [DH00] apply a different approach to enforcing access control on (remote) objects. Their JCCAP system uses capabilities to specify which methods of an object can be accessed by the owner of that capability. Capabilities are implemented through Java interfaces, and provide a limited view on the full interface of an associated object. This makes their system dynamic (in the sense that capabilities can be added and removed from the system independent of the actual implementation of the object, and that capabilities can be delegated between objects. On the other hand, they do not consider the general case of caller and callee residing on different systems separated by a network (as well as the terminal/card line interface). Moreover, the very important matter of protecting the data transfered with an actual method call is not considered in their work.

The latest JavaCard specification (2.2) includes a lightweight version of Sun's Remote Method Invocation (RMI) [Sun99]. It provides a mechanism for a client application running on the terminal to invoke a method on a remote object stored on the card just like an invocation within the same virtual machine. The parameters of a remote method should be primitive (byte, boolean, short, int) or a single-dimension array of a primitive type (byte[], boolean[], short[], int[]). Unlike standard Java RMI, object parameters (whether remote or not) are not allowed. The method result is of primitive type, a single-dimension array of primitive type, a remote interface object or void. All parameters and return values are transmitted by value, except for the remote object. The remote object is transmitted by reference. We have investigated several approaches to implementing our JASON Secure Method Invocation (SMI) system using RMI, but none are quite satisfactory. We discuss this in Sect. 4.4.

Keht *et al.* [KRV00] describe the JiniCard architecture, which allows seamless integration of smart card services in a spontaneous network environment. The approach taken is to keep all functionality required to interact with a certain smart card remotely on the network, and to download this functionality into the card reader based on the ATR (Answer To Reset) of the particular card inserted into it. They also discuss the service-as-object metaphor, but as far as security is concerned, they consider SSL sessions between card and terminal *objects* over which RMI calls are being sent. We, on the other hand, introduce a much finer security granularity at the method level.

There are also a number of related industry initiatives that deserve to be mentioned here.

The Global Platform Specification[2] (formerly VISA's Open Platform specification) is concerned with the secure and platform independent installing and deletion of applications on multi-application smart cards.

The Open Card Framework[3] (and the similarly motivated PC/SC Workgroup[4]) aims to allow software developers to build smart card-aware products without having to worry about platform, card terminal, or smart card-specific interfaces. It supplies an API for handling the communication between a PC application and a smart card reader. Since OCF is developed by the major smart card companies, it supports all kinds of smart cards and card readers. The application does not even have to know which smart card reader is being used during a communication session with a card. OCF does not specify the card side. The choice of a particular type of smart card is free and may change without changing the PC application.

2 Platform requirements

With the JASON SMI system we want to achieve:

- Separation of concerns: specifying security requirements (in the interface definition of a JAVA applet using our keyword approach), and their actual implementation (provided *once* through the JASON SMI system).
- Generic secured access to objects and their methods, independent of their location and whether they are on a compute server or a smart card.
- Providing generic, interoperable, tools to secure method invocations, which can be shared among objects (decreasing the code size) and which can be verified once (increasing robustness and avoiding repeated verification of similar per-applet security measures).
- Decreasing the complexity of writing secure (smart card) applications.

3 Design

The JASON platform implements the secure object store paradigm using the following layers.

Network layer Implements the direct connection between clients, servers, terminals and smart cards, using the Internet Protocol. Between terminal and smart card IP packets are transferred as APDU's. In particular, a smart card (when inserted in a terminal) has an IP address, and the terminal acts as a gateway relaying all incoming IP packets to the appropriate smart card (it may contain more than one smart card) [GBPR00].

remote method invocation layer Serialises method parameters into bytestreams and vice versa, and executes the call on the remote method

[2] http://www.globalplatform.org
[3] http://www.opencard.org
[4] http://www.pcscworkgroup.com/

secure method invocation layer Provides access control and data confidentiality and authenticity.

In this paper we will focus on the design of the secure method invocation layer, and describe it as seen from the application programmer's point of view. We will discuss the close interdependencies with the RMI layer. The SMI layer only requires of the underlying layers that it delivers messages at least to the intended recipient.

3.1 Main components

The Secure Method Invocation (SMI) layer allows a *caller* object to securely call a method implemented by a *callee* object. Both caller and callee are assumed to be stored and run in a protected environment (a *sandbox*) that disables access to all objects and data within the sandbox except through published interfaces.

The JASON SMI layer provides the following services:

- identification and authentication of caller and callee,
- role based access control at the method level, and
- confidentiality and authenticity of method parameters and results.

In future versions other services will be added like:

- logging
- transaction support
- non-repudiation

To call a method of an object, the caller first has to connect to the callee in a particular role. This establishes a *security context* between caller and callee, that (among others) contains the session keys used to protect the communication. Once connected, the caller can call all methods declared by the object accessible to this role. For JASON, roles are equivalent to keys. In other words, ownership of a particular key associated to a role, proves that an object can connect in that role.

To establish a connection, the caller needs a *stub* corresponding to the object to connect to. Similarly, the callee needs a *skeleton* that receives incoming connections, performs access control decisions and protects the method parameters and results. The role keys used to authenticate the caller to the callee are stored in a separate *keystore* object belonging to the same sandbox. This design is sketched in Fig. 1.

The stub and skeleton necessary to securely call the methods of an object are generated automatically from a so called JASON definition file. This file specifies the security requirements for the callee object. The contents and structure of this file are described next. Note that the issue of key management falls beyond the scope of this paper. We are currently investigating the proper tools to support key management within the JASON framework. As far as the JASON SMI platform is concerned, the keystore contains valid and proper keys.

3.2 The JASON definition file

The JASON SMI system has a strict separation between the card application and its security. An application developer has two tasks.

- Write a card object without bothering about security or APDU exchange, instead focusing on the information processing logic of the application.
- Write a JASON definition file describing the security requirements.

Therefore, the security requirements for an object are written in a separate JASON definition file that resembles the syntax of a JAVA interface description.

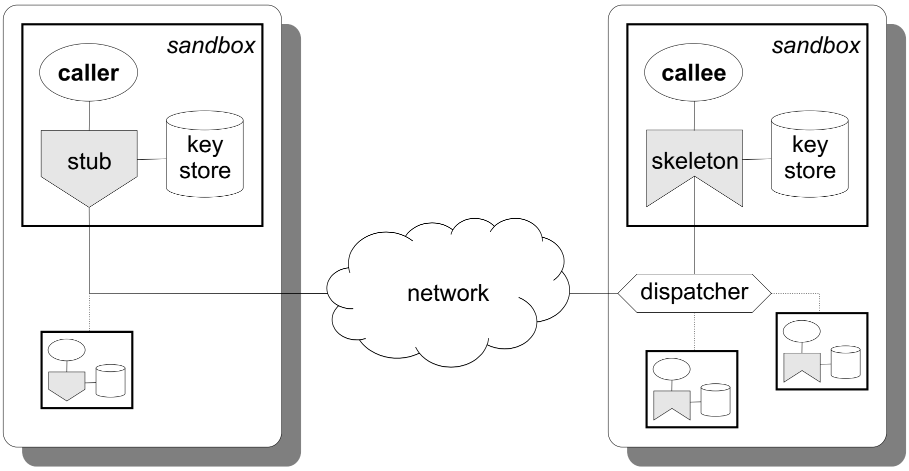

Figure 1: Caller and callee components.

```
package com.ebank;

public interface Purse
{
  roles BANK, MERCHANT, OWNER;

  accessible to ALL
  authentic short getBalance();

  accessible to BANK
  authentic short increaseBalance( confidential authentic short amount);

  accessible to MERCHANT
  authentic short decreaseBalance( authentic short amount);
}
```

Figure 2: JASON definition file for a simple purse.

A sample JASON definition file appears in Fig. 2 (describing the interface of a simple electronic purse application, that will be studied further in Sect. 5). The JASON precompiler will process the definition file and generates three files.

- A plain JAVA interface file. All keywords not known in JAVA are removed. This is the interface implemented by both the implementation of the callee object and the client stub.
- A client/caller stub, whose methods are called to execute the corresponding remote methods, and that performs authentication and marshalling (including protection) of data.
- A callee skeleton performing access control decisions, unmarshalling of parameters (verifying signatures and decrypting parameters where necessary) for incoming invocation requests, and executing the actual method.

In JAVA the keywords `private`, `protected` and `public` are used to limit access to methods and fields to certain classes. An object can only access it's own private members, protected members of it's superclasses or classes in the same package and all public members. These keywords work fine if used inside a single virtual machine. However, when using a distributed system a more fine grained solution is necessary.

In the JASON SMI system, access control is role based. Moreover, the communication between caller and callee has to be protected as well. To specify these requirements, the JAVA interface description is extended with the following keywords.

- `roles` ⟨*role-list*⟩, listing the different roles in which a caller can connect to this object. The roles in this list correspond to keys stored in the keystore.
- `accessible to` ⟨*role-list*⟩, specifying which roles can call the indicated method.
- `confidential` and/or `authentic`, specifying that a parameter or a method result should be confidential and/or authenticated.

Here a ⟨*role*⟩ is an identifier (usually in all caps because it is a constant), and a ⟨*role-list*⟩ is a comma-separated list of roles. Let us discuss the last three keywords in a little more detail.

`accessible to` ⟨***role-list***⟩ Access to a method can be limited by using the `accessible` keyword. Access is only to be granted if the caller can be identified (using the corresponding keys in the keystore) as a role in ⟨*role-list*⟩. The predefined role `ALL` indicates that access is allowed for all roles defined for this object (through the `roles` keyword). The predefined role `ANYBODY` specifies a role that can be assumed by anybody (i.e., a role whose identity is not verified). For security reasons only methods are accessible from off-the-card applications. Variables should be accessed through corresponding set and get methods.

`confidential` Parameters and return values can be specified as `confidential`, meaning that the data involved should be sent encrypted between caller and callee. This guarantees that nobody else can eavesdrop the value. In the negotiation phase (see below) a (symmetric) session key is exchanged and an encryption algorithm chosen.

`authentic` Parameters and return values can also be specified as `authentic`. This gives the following guarantees.

authenticity Only the caller can construct valid parameters[5], and only the callee can construct valid responses. The parameter received by the callee was sent by the caller, and the result received by the caller was sent by the callee. In particular, this gives the caller the guarantee that the intended side effects of the method call did in fact occur at the callee (like decreasing the balance of a purse).

integrity The parameter (or the result) received was not altered while in transit.

freshness The parameter received was passed by the caller for the *current* call of the method (and not for any previous call). The result received was sent by the callee for the current call of the method (giving the guarantee that the method was actually executed at this time, see above).

In practice this means that the data involved should be signed, and that a form of replay protection is added as well.

3.3 Using SMI

To call a method using the SMI framework, the caller has to perform the following two steps (see also Fig. 3 for an example connecting to the purse object whose interface was given previously).

- The first step is to connect to the callee and to establish a security context. The

[5] Strictly speaking, because a symmetric session key is used to protect the data, also the callee can construct valid parameters. Therefore non-repudiation cannot be guaranteed.

```
try {
  Purse purse = (Purse) SMINaming.connect("smi://smartcard/Purse",
      Purse.MERCHANT, purseKeyStore) ;
  try {
    purse.decreaseBalance(10);
    System.out.println("You have paid");
  }
  catch (UserException ue) {
      System.out.println("Transaction failed. You have not paid.");
  }
}
catch (RemoteException re) {
  System.out.println("Failed to connect to service.");
}
```

Figure 3: Caller connecting to a callee

caller passes the name and location of the desired service, the desired role in which to connect, and a reference to the key store to SMINaming.connect(). When successful, this returns a reference to the required stub.

- Subsequently, the methods of the remote object can be called securely as if they were local methods of the stub returned by the previous step.

If a connection is established, the stub also contains the current security context for that connection. Among other things, this security context contains a session key used to secure subsequent method invocations. Also, it contains further identification information on the callee object. This identity can be retrieved by the stub's `getSessionIdentifier()` method.

Note that even for a single call to a method, a connection has to be set up. This may be wasteful for certain applications where transaction speed is very important (e.g., public transport). We are investigating the possibility of calling a single method without connecting to the object first (in fact merging the connection and the calling into one step).

4 Architecture

In this section we describe how the JASON SMI platform is actually implemented, and how the security requirements are actually met using several cryptographic protocols. In particular we show how a secure connection is setup, how the ownership of roles is verified, and how the security context is established. Secondly, we show how a method is called securely using the information and session keys in the current security context. But first we will discuss the keys stored in the keystore in a little more detail.

4.1 On keys

The keys in the keystore correspond one-to-one to the roles declared in the JASON definition file. The keystore also contains keys for key-management. This is discussed in a forthcoming paper.

JASON supports the use of different types of keys in the keystore, depending on the security requirements of the application (or indeed individual objects on particular smart cards). Currently, the following types of keys are supported.

- RSA, with 512, 1024 and 2048 bit keys.
- DES and 3DES.

- AES, with 128, 192 and 256 bit KEYS.

Moreover, JASON supports *diversified* keys [AB96] where the key k_i stored by callee i (used by the callee to authenticate the caller or vice versa) is derived from the master key k_M stored by the caller. The key is derived using the formula

$$k_i = \{i\}_{k_M} ,$$

where $\{m\}_k$ denotes encryption of message m using key k (where the encryption method is defined by the type of the key). Note that in this case k_i performs the role of a public key (from which the corresponding private key cannot be derived), but with additional property that it proves to the *caller* the identity i of the callee.

Depending on the type of key stored in the keystore, the appropriate authentication protocol is run. Note that the caller keystore contains the keys necessary to prove its role (e.g., private keys), while the callee keystore contains the keys necessary to verify a role (e.g., public keys). If an entry in the caller keystore is `null` or invalid, the caller cannot assume the corresponding role. If an entry in the callee keystore is `null` or invalid, the role cannot be verified and all connections for that role will be refused.

Finally, the keystore contains, for each role key, information about the type of cipher that should be used to protect the session once the caller has been authenticated and accepted.

4.2 Connecting to an object

Connecting to an object exchanges and verifies the identity and role of the caller and the callee. Furthermore, a security context is established (containing a shared secret key) that is used to protect all calls to methods of the object. To connect to an object and establish a session the following steps are taken (assuming RSA style authentication).

- The caller sends a message containing
  - the role (as an index in the keystore) as which it wants to connect,
  - the type of key it will use to authenticate the role (RSA in this example),
  - a list of all ciphers it will accept to protect the session, and
  - a nonce.
- The callee looks up the role and the type of keys it can accept. If it can accept the suggested authentication method, it will select one of the ciphers to protect the session from the list it received (provided it supports it). It then sends the following message
  - the selected cipher to protect the session,
  - a random master secret encrypted with the public RSA key found for the role in the keystore, and
  - a nonce,
- The caller validates the proposed cipher, decrypts the master secret with its private key in the keystore.
- Both caller and callee generate the session key (using hashes) from the master secret and both the caller and the callee nonces.
- Caller and callee exchange further identifying information encrypted and MAC-ed using the session key, and record that in the security context.

Both caller and callee record the session key in the security context for this connection. Note that if a connection is established as `ANYBODY`, no verification of that role can be performed. In that case, the master secret must be exchanged using a Diffie-Helman type key exchange. Future method invocations are will be secured using this session key.

The session context also contains two counters, one to count the number of messages sent in this session, and one to count the number of messages received. Both are reset to 0 at the start of a session, and incremented for each message sent or received.

These numbers are used to protect against replay, as explained below.

4.3 Method invocation

Informally speaking, after session setup the stub and the skeleton are connected by a (secure) byte stream. The byte stream is routed by the communications layer to the correct skeleton. In fact, when a stub's method is invoked, it does the following:

- reconnect to the remote JVM containing the remote object,
- marshal(write and transmit) the parameters to the remote JVM,
- wait for the result of the method invocation,
- unmarshal (read) the return value or exception returned, and
- return the value to the caller.

The stub hides the serialisation of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote JVM, each remote object has a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshal (read) the parameters for the remote method,
- invoke the method on the actual remote object implementation, and
- marshal (write and transmit) the result (return value) to the caller.

The byte stream sent from stub to skeleton contains the following elements.

- The name (or rather the index) of the method to call, together with a MAC computed using the session key and the current value of the sent messages counter. Even if RMI is used as the transport mechanism, this information is necessary to prevent remote method invocations being redirected to the wrong method.
- Each confidential parameter is encrypted.
- For each authentic parameter, a MAC computed using the session key and the current value of the sent messages counter is appended to the parameter.

For efficiency reasons parameters are shuffled so that the `confidential` and `authentic` parameters are placed in contiguous blocks within the byte stream (see Fig. 4). All confidential parameters are encrypted as a single block. Similarly, the MAC for all authentic parameters is computed in a single block, appending the sent messages counter only once.

The return stream from skeleton to stub to communicate results has the following structure.

- If the return type is confidential, the return value is encrypted with the session key.
- If the return type is authentic, the sent messages count is appended to the byte stream, and both the count and the value are used to compute a MAC with the session key. The result is appended to the byte stream.

4.4 Inter object communication

Because the caller and callee are physically separated by a network, the call to a remote method must be transferred to the remote object over the network before it can be executed there. The most natural approach would be to use Java's Remote Method Invocation mechanism to achieve this. At the caller side, the SMI stub first converts the

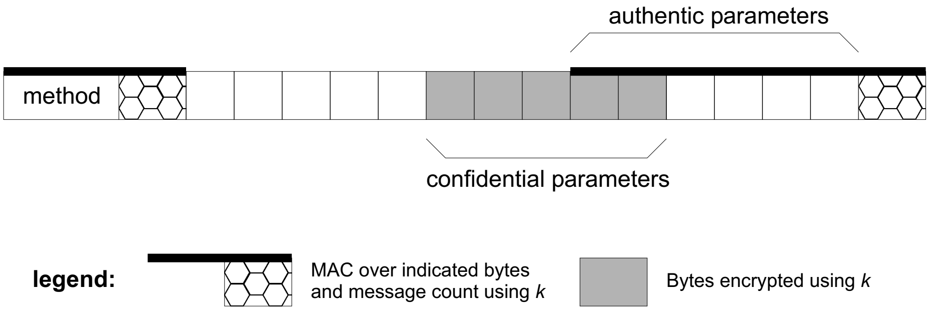
authentic parameters
method
confidential parameters
legend:
MAC over indicated bytes and message count using *k*
Bytes encrypted using *k*

Figure 4: Byte stream structure from caller to callee.

parameters to a protected bytestream, as explained in Sect. 4.3. The RMI layer than transmits this bytestream to the callee, and invokes the corresponding method of the callee SMI skeleton. There, the access permissions are checked and the bytestream is unpacked before the original callee method is invoked.

However, this scenario is complicated by the fact that JavaCard (as of version 2.2) uses a different RMI system, if only because a JavaCard is not connected to a network directly, but instead communicates with the outside world through a terminal using an APDU stream. This would imply that the terminal has to convert an incoming RMI request to a JavaCard specific JC-RMI request (and similarly for the responses). This does not appear to be straightforward, because the RMI wire protocols are different. The only option is to create – for each skeleton on the callee smart card – a separate skeleton (and stub) for the terminal, that receives the incoming RMI request and simply calls the remote method on the smart card using JC-RMI. This means the terminal potentially needs access to a huge number of skeletons and stubs, simply to pass bytestreams verbatim!

Moreover, we note that RMI's support for marshalling and unmarshalling of method parameters and results becomes totally superfluous in this approach, because the SMI layer already converts the parameters to a bytestream in the first place.

To solve the first problem RMI and JC-RMI need to be brought more in line, such that their wire protocols become sufficiently compatible to allow translations between them using a generic translation mechanism running in the terminal. To solve the second problem, the RMI system should provide versatile hooks to allow the outgoing bytestream to be protected in the fine grained manner required by JASON. Or, SMI should be incorporated into the RMI layer.

5 Example

Fig. 2 in section 3.2 shows the security requirements of a simple purse application. It corresponds to the actual implementation given in Fig 5. Clearly the implementation is quite straightforward. Also, the strictness of the separation between implementation and its security is apparent. The implementation does not contain a single line of code concerning security. All the security is contained in the generated stub and skeleton. The skeleton calls the implementation and adds security to it. Note that each method is defined with the default JAVA visibility, to allow the skeleton to access them, but not giving access to subclasses outside the package.

6 Conclusions & Further Research

We are currently implementing the JASON SMI framework on a JavaCard 2.2 platform. The final implementation will be available under the GNU

```
package com.ebank;

class PurseImpl implements Purse {
  public static final byte OVERFLOW = (byte) 1;
  public static final byte UNDERFLOW = (byte) 2;
  private short balance = 0;
  private static final short MAX = (short) 500;

  short getBalance() {
    return balance;
  }

  short increaseBalance(short amount) throws UserException {
    if (balance + amount < MAX) {
      balance += amount;
      return amount;
    } else
      UserException.throwIt(OVERFLOW);
  }

  short decreaseBalance(short amount) {
    if (balance - amount > 0) {
      balance -= amount;
      return amount;
    } else
      UserException.throwIt(UNDERFLOW);
  }
}
```

Figure 5: Implementation of a simple purse.

General Public License (GPL) through http://www.cs.kun.nl/~jhh/jason.html within a few months.

We intend to extend JASON's SMI functionality with logging and auditing functions, as well as transaction (and rollback) support. Related to the logging and auditing issue, is the fact that the current implementation does not provide non-repudiation. The ramifications for implementing non-repudiation are the subject of further investigations. Also, one could argue that the authentic keyword is overloaded (in the sense that it gives too many guarantees, especially freshness, at the cost of a more complex and resource consuming protection mechanism). Using JASON to develop several real-world smart card applications will tell whether a more fine grained set of security specification keywords is required.

Finally, to make the JASON vision of a smart card application consisting of millions of distributed objects a reality, object broker functionality has to be added that is consistent with the high security requirements of typical smart card applications, and the highly dynamical nature of the smart card network.

7 Acknowledgements

We thank the anonymous referees for the valuable comments and suggestions, and especially for bringing to our attention that the JavaCard 2.2 standard contains support for RMI.

References

[AB96] Anderson, R. J., and Bezuidenhoudt, S. J. On the reliability of electronic payment systems. *IEEE Trans. on Softw. Eng.* **22**, 5 (1996), 294–301.

[Che00] Chen, C. *Java Card (tm) for Smart Cards: Architecture and Programmer's Guide.* The Java Series. Addison-Wesley, 2000.

[DH00] D. Hagimont, J.-J. V. Jccap: capability-based access control for java card. In *4th CARDIS* (Bristol, UK, 2000), pp. 365–388.

[Gut00] Guthery, S. How to turn a GSM SIM into a web server. In *4th CARDIS* (Bristol, UK, 2000), pp. 209–224.

[GBPR00] Guthery, S., Baudoin, Y., Possega, J., and Rees, J. Ip and arp over iso 7816-3. Internet Draft guthery-ip7816-00, 2000.

[HJF95] Hartel, P. H., and Jong Frz, E. K. de. Smart cards and card operating systems. Tech. rep., Dept. of EE and CS, University of Southampton, UK, 1995.

[ISO7816] International Organisation for Standardisation (ISO), JTC 1/SC 17. ISO/IEC 7816 Identification cards – Integrated circuit(s) cards with contacts.

[IFH00] Itoi, N., Fukuzawa, T., and Honeyman, P. Secure internet smartcards. In *1st JAVACARD* (Cannes, France, 2000), I. Attali and T. Jensen (Eds.), LNCS 2041, Springer-Verlag, pp. 73–89.

[KRV00] Kehr, R., Rohs, M., and Vogt, H. Issues in smartcard middleware. In *1st JAVACARD* (Cannes, France, 2000), I. Attali and T. Jensen (Eds.), LNCS 2041, Springer-Verlag, pp. 90–97.

[RH00] Rees, J., and Honeyman, P. Webcard: A java card web server. In *4th CARDIS* (Bristol, UK, 2000), pp. 197–208.

[Sun99] Sun. Java remote method invocation specification. Tech. rep., Sun Microsystems, Inc., 1999. Revision 1.7.