JavaCards As Secure Objects Network

R. Brinkman

Master thesis
Distributed and Embedded Systems
Computer Science
University of Twente
November 2002

Evaluation committee:
dr. J.H. Hoepman
prof. dr. P.H. Hartel
prof. ir. E.F. Michiels
dr. S. Etalle

Abstract

Jason is an acronym for JavaCards As Secure Objects Network. The Jason framework simplifies the development of a network of cooperating smart card applets. The security of both the applets and the communication between them is handled completely by the framework. The application developer does not have to be concerned with it. He only specifies the security requirements in the form of a Jason definition file. Data which will be transmitted over the network can be marked as confidential or can be signed. The rights to invoke a method can be specified as well. A special pre-compiler will transform this file into a skeleton object running on the smart card and a stub file running on the PC. The generated objects handle the security and the marshalling of individual parameters into a single byte array.

Samenvatting

Het Jason raamwerk, hetgeen staat voor JavaCards As Secure Objects Network vergemakkelijkt het om een netwerk op te zetten van samenwerkende smart card applets. De beveiliging van zowel de applets zelf als de communicatie ertussen wordt door het raamwerk verzorgd. De programmeur van de applicatie hoeft zich niet druk te maken over de beveiliging. Hij specificeert door middel van een Jason definitie bestand welke eisen hij aan de beveiliging stelt. Data die over het netwerk verstuurd moet worden, kan worden aangemerkt als vertrouwelijk of kan worden ondertekend met een handtekening. Ook de rechten om een functie aan te roepen kunnen worden gespecificeerd. Een speciale pre-compiler leest dit bestand en genereert aan de hand hiervan een skeleton object die op de smart card draait en een stub object die op de PC draait. Deze gegenereerde objecten dragen zorg voor de beveiliging en de conversie van losse parameters naar één groot data blok.

Contents

1 Introduction

The JavaCard platform made it possible to develop smart card application using a high level language: Java. Java is an Object Oriented Programming (OOP) language. 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), 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. These objects are stored on secure smart cards. Therefore, this network is highly dynamic. Smart cards are usually offline and only connect to the network when they are inserted into a card accepting device. This network needs to be highly secure. Not only the objects should be stored securely, also the communication has to be secure. Access to certain objects should be restricted, and the confidentiality and authenticity of the communication between the objects have to be guaranteed. Communication uses a Secure Method Invocation (SMI) scheme. The Javacards As Secure Objects Network (Jason) platform acts as a middleware layer 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 section 2 the current technology is discussed. The ISO 7816 [4] standard and the JavaCard standard are explained in sections 2.1 and 2.2. Two communication mechanisms are discussed in section 2.3. The similar approach of Hagimont and Vandewalle [10] is discussed in section 2.4.2.

After looking at some possible scenarios in section 3.1 the requirements specification is distilled in section 3.3. This requirements specification leads to the design given in section 4. In this section 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 (section 4.3). The protocol data units that the network (section 4.2) uses, are dealt with in section 4.3. At last the security (section 4.5) and the naming paradigm (section 4.6) are described.

In section 5 a simple example is discussed. All steps from writing the Jason Definition File and the implementation to the generation of the stub and skeleton are explained. The big advantage for the smart card application developer is that he only needs to specify the security requirements, but does not have to implement the security protocols himself. They are present in the generated stub and skeleton.

After having seen how the secure method invocation works, section 6 tells us why it works. The chosen architecture of section 6.2 is a natural conclusion of the layering possibilities that are given in section 6.1. A short summary of the Jason pre-compiler is given in section 6.3. For a more elaborate discussion on the pre-compiler's implementation you may take a look at the API documentation. The way key management and naming is handled is given in sections 6.4 and 6.5.

Before the conclusion that the framework works correctly, some problems that arose during the test phase are discussed in section 7 as well as the tools that can be used for testing.

2 State of the art

In the area of smart card technology a lot of research has been done and is being done. Particularly the JavaCard standard is a promising evolvement. The JavaCard standard fully complies with earlier smart cards. It is compatible with the ISO 7816 standard that will be discussed next. Section 2.2 deals with the JavaCard standard in more detail. The discussion about the security of smart cards is postponed to section 2.4.

2.1 ISO 7816

There are many types of smart cards and card accepting devices (CAD). Without a standard each type of smart card should have its own type of CAD. Fortunately there is such a standard. ISO 7816 [4] specifies the protocol that is being used for the communication between smart card and CAD. Like IP packets, the Application Protocol Data Units (APDU's) consist of a header field and a data part. Two types of APDU's exist: command APDU's and response APDU's. Command APDU's are the data packets from the CAD to the card and response APDU's are the data packets from the card to the CAD. Figure 1 shows the fields of a command APDU. The Class byte (CLA) and the Instruction byte (INS) together determine which function should be invoked. CLA values in the range 0x0X, 0x8X, 0x9X and 0xAX are reserved by the ISO standard. The two parameter bytes are not specified and can be used freely. L_c is the length of the adjacent data field. L_e is the expected length of the response data. When the response data is larger than L_e the response is divided into multiple APDU's that are smaller or equal to L_e.

CLA	INS	P_1	P_2	L_c	Data	L_e

Figure 1 ISO 7816 Command APDU

Data	SW_1	SW_2

Figure 2 ISO 7816 Response APDU

Figure 2 shows the format of the response APDU. The Data size is always less than or equal to the L_e value of the corresponding command APDU. The Status Word bytes SW_1 and SW_2 hold the error status. A value of 0x9000 indicates that the command is performed without error and that the Data part contains correct data. The ISO standard predefines status words in the range 0x6XXX. Other status words can be used within the applet.

2.2 JavaCard

JavaCard technology [3] makes it possible to develop software for a smart card using a high level language: Java. This technology is platform independent, since it is running on a virtual machine instead of directly on the target machine. It is possible to download multiple applications to the card. In the JavaCard terminology these applications are named *card applet*, *caplet* or simply *applet*. Each applet runs securely within its own sandbox, without interference with other applets that are present on the card. It is therefore possible for post-issuance applications to be added to an existing card. The JavaCard standard is fully compatible with the international standard ISO 7816 [4].

Smart cards have very limited processing power and even less memory space. A typical JavaCard is equipped with a processor of about 5 MHz and 16 kB of memory. Often, three types of memory are present on a single card. Read Only

Memory (ROM) is used for immutable code like the operating system and the native code of the Java virtual machine. Electronical Erasable Read Only Memory (EEPROM) is used for persistent data like the card applets and long lived objects. For temporary data (i.e. session keys) Random Access Memory (RAM) can be used. RAM will loose all data when the card is removed from the card accepting device. Objects are stored in EEPROM. Their state is preserved even when the card is being ejected from the card accepting device. No file I/O is needed, because all data is stored automatically. Because of the limited amount of memory only a subset of Java is implemented on the card. For example, no String class is available and the garbage will not be collected. Of the primitive types only bytes, booleans, shorts and single dimension arrays of these types are allowed. Newer JavaCards also allow the int type.

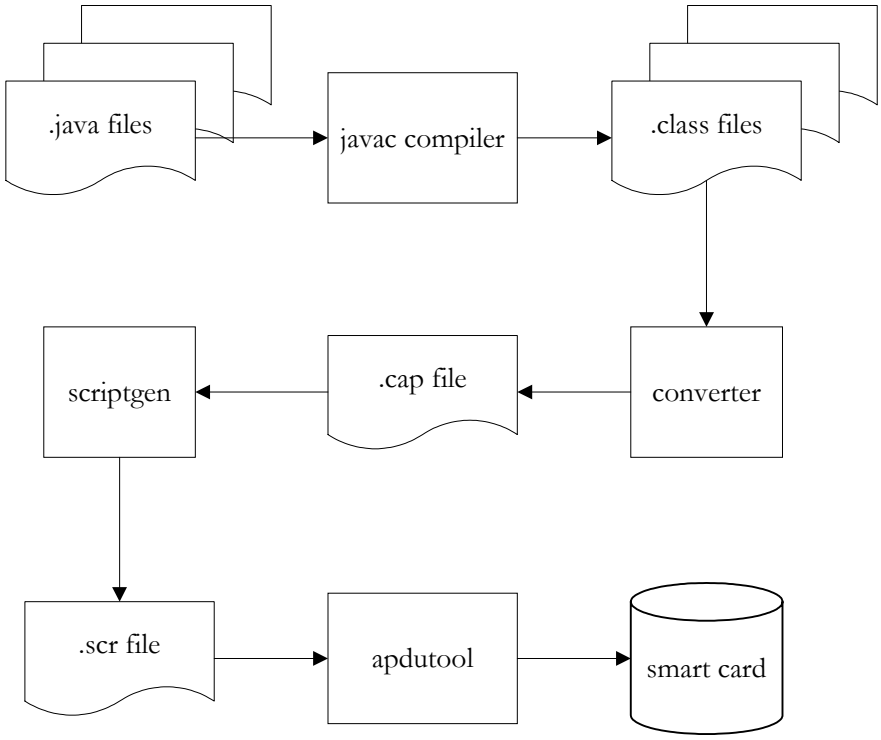

Figure 3 Conversion from source file to smart card

The process of building an applet and downloading it to a smart card is sketched in figure 3. The process starts like any other Java application, enabling the use of of-the-shelf Java Developer Environments to create the class files. The converter behaves much like the standard jar tool except that it also optimizes the code. Before the cap file can be sent to the card, a script file must be generated. Each line of this text file contains an Application Protocol Data Unit (APDU) in hexadecimal form. At last the apdutool reads the script file and sends the APDU's to the card accepting device, which will store it on the card.
Of all present applets, at most one is selected. Only the selected applet can perform tasks. All other applets are suspended. APDU's sent to the card will first pass the internal card dispatcher. The dispatcher keeps track of the currently selected applet and redirects all APDU's to it. When a select APDU is received, the currently selected applet will be deselected. The part of the RAM marked with CLEAR_ON_DESELECT will be cleared, but the EEPROM will keep its state.

2.3 Connecting to smart cards through terminals

In section 2.1 we saw that the ISO 7816 standard defines the protocol between smart card and terminal. Unfortunately no such standard does exist for the com-

munication channel between terminal and client application. A client application specialized for communicating with one type of terminal, will not work in combination with other terminals. Therefore an abstraction layer is necessary between application and terminal, providing a static API for the application. It should be simple to plug in a device driver for a terminal. The OpenCard Framework (OCF) provides such an API. The new JavaCard Remote Method Invocation (JCRMI) implementation makes use of it.

2.3.1 OpenCard Framework

In the smart card business many parties are involved: card terminal vendors, card operating system providers, card issuers and card users. The OpenCard Framework (OCF) [9] provides a mechanism to make them independent of each other. A card user (a bank, for example) is not bound to a card issuer or a terminal vendor. OCF 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. An 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.
To accomplish the separation between the client application and the terminal the OCF framework itself is separated into two layers (see figure 4). The CardService layer consists of a static API for the client application providing a gateway to the services the card implements. The CardService layer abstracts from the type of smart card (operating) system. The client application using the CardService layer may not know whether the card has a file oriented structure or an object oriented one. The CardServices give interfaces to some standard smart card operating functions. These interfaces abstract from the various smart card implementations. The CardTerminal layer abstracts from the type of terminal hardware. It consists of various interfaces and abstract classes that a terminal manufacturer should implement in order to plug their terminal type into the OpenCard framework.

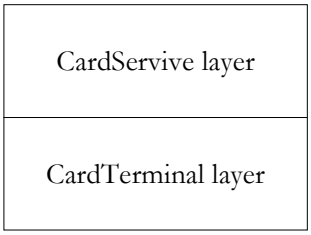

Figure 4 OCF layers

2.3.2 JCRMI

The latest JavaCard specification (2.2) includes a lightweight version of Sun's Remote Method Invocation (RMI) [5]. It provides a mechanism for a client application running on the terminal to invoke a method of 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.
An object can be invoked remotely only when it implements a subinterface of java.rmi.Remote. The subinterface specifies the methods that will be exported. Each method should add java.rmi.RemoteException to its throws clause. A RemoteException will be thrown by the RMI system when a transmit error occurs. Only

exceptions that are specified in the JavaCard specification are catchable by the client application. Use UserException for your own exceptions.
A remote object implementation can be translated to a stub file by the rmic compiler. This stub file also implements the remote interface extension and acts as the local placeholder for the actual implementation. The parameters will be marshalled into a byte array. See appendix B for the format of the serialized byte array. The serialized data is encapsulated into an APDU which is sent to the card. The receiving applet will dispatch the APDU to an instance of the RMIService class. The RMIService is instantiated with a remote object and will handle all the marshalling for that object.

2.4 *Security of smart cards*

A JavaCard, like any other smart card, is mostly used for security reasons. Therefore a JavaCard has build in support for encryption/decryption as well as signing and verifying signatures. Furthermore a firewall is protecting the applets from interference with other applets on the card. Section 2.4.1 will deal with the security aspects that standard JavaCards are equipped with. Section 2.4.2 will handle a security framework similar to Jason.

2.4.1 JavaCard

JavaCards are equipped with a javacard.security and a javacardx.crypto package which are lightweight versions of the java.security and javax.crypto packages. The packages only specify the overall framework. They do not contain the actual implementations. Typically the implementation consists of native methods which are stored in the ROM part of the smart card.

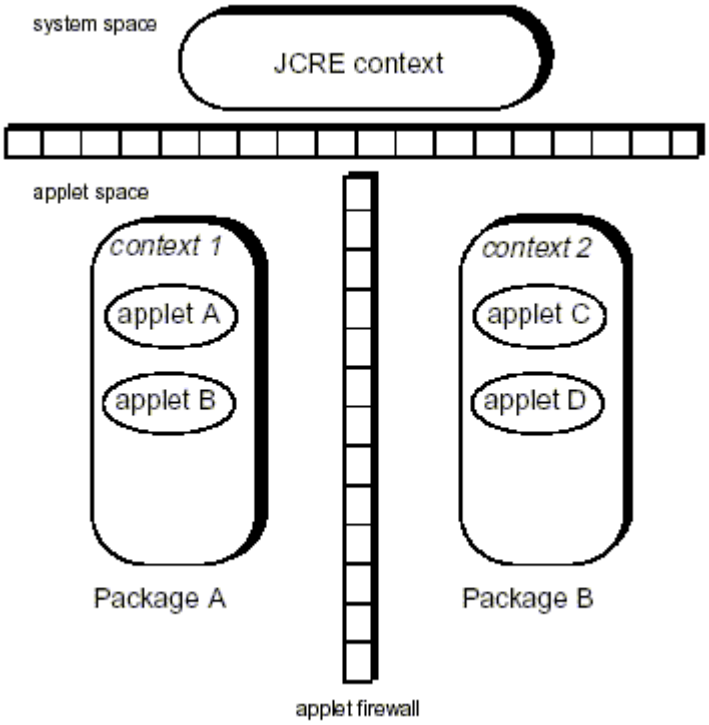

Figure 5 Applet Firewall mechanism [7]

To increase security, a firewall mechanism is added to JavaCards. Each package containing one or more applets is assigned to a security context. Security contexts are separated by a firewall as is shown in figure 5. Apart from the applet contexts one context for the JavaCard Runtime Environment (JCRE) [7] exists. It is merely a

security context with extra system privileges. Only one security context can be active at a time. The virtual machine checks all byte codes that access objects in order to verify if access is allowed. If the object does not belong to the active security context, a SecurityException is thrown.
To allow an applet access to an object of another security context there are four mechanisms for switching contexts in a secure and well-defined way: using JCRE entry point objects, global arrays, JCRE privileges and shareable interfaces. JCRE entry point objects are objects belonging to the JCRE context that are marked as entry point to system routines. Global arrays may be accessed by all contexts, but can only be created by the JCRE itself. Typically, there is only one global array, i.e. the APDU buffer. The JCRE may access all objects of all contexts. This is one of the privileges the JCRE context has. Shareable interfaces are the only mechanism for normal applet contexts to access objects from another context. A server object providing shareable methods should implement a subinterface of Shareable. All methods defined in the subinterface can eventually be accessed from other security contexts. A client object that wants to invoke the shareable method must first invoke JCSystem.getShareableInterfaceObject. This static method redirects the request to the applet owning the server object. This applet may decide whether or not the client applet is granted access.

2.4.2 JCCap

Hagimont and Vandewalle propose in their paper [10] a security framework with capabilities based on the earlier developed Direct Method Invocation (DMI) system [11]. Their requirements specification resembles a subset of our own requirements (see section 3.3). In short, they wanted to separate security issues from the implementation. An application developer should not be bothered with writing security algorithms himself. He only has to specify the security requirements without implementing them. Only access control is handled by their framework. Confidentiality and authenticity of parameters and method returns are not assured.
Capabilities are used to protect the methods of an object. An object possessing the appropriate capability has access to the method. Capabilities can be delegated to other objects. For instance, object A possessing a capability for object B may give this capability to object C.
Hagimont and Vandewalle distinguish two kinds of method invocation. The simplest kind is between objects on the same smart card. If the objects do not belong to the same security context (see 2.4.1) shareable interfaces are used. The more difficult kind is between two objects within different virtual machines. DMI is used for the method invocation of a smart card object by an object residing in a terminal virtual machine.
A capability can be seen as a gateway object containing a reference to the actual object and the access rights for it (see figure 6). For a single object multiple capabilities can exist. For example one view with only read access and another with full access rights. A capability object is automatically generated by a stub compiler. Input for the stub compiler is the user defined view. This view is written in a special Interface Definition Language (IDL) resembling a Java interface.

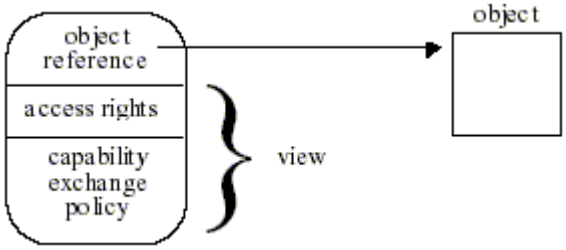

Figure 6 Structure of a capability [10]

As an example: consider two applets, a bank applet and a client applet. The bank applet manages account objects and has full rights for them. The client applet is granted limited access to his own account. The account object implements the Account interface specified in listing 1.

```
public interface Account {
  int readBalance();
  void writeBalance(int balance);
}
```

Listing 1 Account interface

The application developer can specify two views. One for the bank object and one for the client object. These views are shown in listings 2 and 3. The single difference between the two views is the keyword **not** before the *writeBalance* method.

```
view BankView implements Account {
  int readBalance();
  void writeBalance(int balance);
}
```

Listing 2 Bank view

```
view ClientView implements Account {
  int readBalance();
  void not writeBalance(int balance);
}
```

Listing 3 Client view

These two views serve as input for the Stub compiler. It generates the BankAccount and the ClientAccount shown in listings 4 and 5.

```
public class BankAccount implements Account {
  Account account;

  public BankAccount(Account account) {
    this.account = account;
  }

  int readBalance() {
    return account.readBalance();
  }

  void writeBalance(int balance) {
    account.writeBalance(balance);
  }
}
```

Listing 4 Bank account capability

```
public class ClientAccount implements Account {
  Account account;

  public ClientAccount(Account account) {
    this.account = account;
  }

  int readBalance() {
    return account.readBalance();
  }

  void writeBalance(int balance) {
    throw new Exception("Illegal access");
  }
}
```

Listing 5 Client account capability

Instead of giving the actual Account implementation, the bank object which is shown in listing 6 supplies the ClientAccount capability to its clients and BankAccount to itself. The Java virtual machine exhibits access to objects not explicitly given as a parameter of a method or as a result of a method. Therefore a client object cannot invoke the actual writeBalance method, because it does not have a reference to the actual Account implementation.

```
public class Bank {
  public Account getAccount(String accountNumber) {
    ...
  }
}
```

Listing 6 Bank object

When the invocation involves two virtual machines, the problem becomes more difficult. Inside a single virtual machine the byte codes are verified before they are executed. It is impossible to forge a reference to a capability which is not obtained by either a parameter or a method result. When using a remote method invocation scheme the references are encoded into an APDU. The caller may forge the reference. To prevent this a password protection scheme is added. From the paper [10] it does not become clear how the passwords are exchanged between card and terminal.

3 Requirements specifications

Before writing down the requirements specification of a system it is a good practice to define some possible scenarios in which the system will be used. Section 3.1 describes some of these scenarios. Before distilling the requirements some security threats are discussed in section 3.2.

3.1 Possible scenarios

3.1.1 Access card for a building

A smart card can be used as a substitute for a normal key. A normal key typically fits into a single lock. If a person wants to open doors with different locks, he has to carry all the keys.
If the building doors are equipped with card readers, a person has to carry only one card. There are two approaches how a door can decide if a card (and its owner) is allowed to pass through the door.
It is possible to give each door a different symmetrical key (for instance a DES-key). Each user who is allowed to open the door can upgrade his card by some authentication authority with the door's symmetrical key. The door can check if the card has the same encryption key, by sending a random challenge and checking the response (see figure 7).
This scheme works fine, unless the person's card gets lost or stolen. In order to protect the building against entering of the finder or the thief, the door's key has to be changed. All the cards containing the old key have to be updated too. A possible solution to this is to supply different keys to each user and different keys for each door. In practice this means that every ⟨smart card, door⟩ pair uses its own key. Granting a smart card entrance through the door involves updating both the smart card and the door while in the first case only the card was updated.

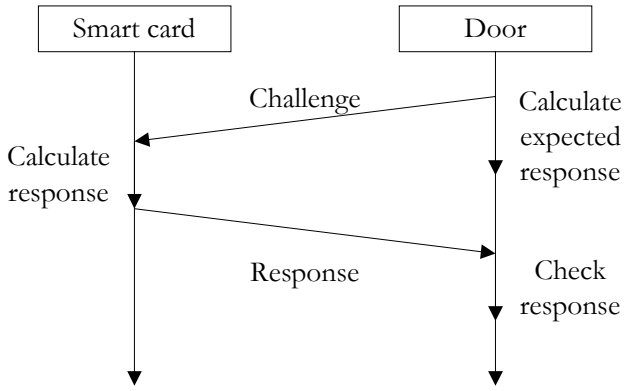

Figure 7 Access to a building

Another approach is the use of asymmetric encryption. Each card has its own private key. A certification authority can grant a person access to the building by signing a certificate stating that the card owning the private key which corresponds to the public key written on the certificate is granted access. The certificate can be stored on the card itself or at another place accessible by the door.
Now, the card sends the signature of the challenge to the door. The door checks the signature by validating it with the public key written on the certificate. It also checks the signature of the certificate itself.
In case of theft the certificate can be added to a revocation list being checked by the door.

3.1.2 Substitute for passport

The same scheme described in paragraph 3.1.1 about the access card for a building can be applied to more cases. At all places where a person has to prove his identity such a smart card can be used. It can serve as a substitute for a passport, because a passport has the same purpose: identification of the cardholder.

3.1.3 Software license management

A great frustration of many software developers is the making of illegal copies of their products. Nowadays software is "personalised" by entering a serial code or a license key when installing the software. But the serial code is as simple to copy as the software itself. Smart cards can be used to securely store the license key. The license key can be checked only at installation or each time the application starts. As a customer buys more software protected with a smart card, he should not be frustrated by switching smart cards when switching to another application. So it should be possible to move a license key from one card to another. In [1] there is an description a method to do so, without copying the license key. After moving the license key from card A to B, card A has become useless and can be thrown away.

3.1.4 Electronic purse

A smart card equipped with an electronic purse is a substitute for the more conventional wallet. However, although it is possible to pay with your electronic purse in a shop, it is impossible to pay your friends with it. As a complete substitute for the conventional wallet, it should be possible to move some "money" from one purse to another. It looks like the movement of software licenses between cards (see 3.1.3).

3.1.5 Health care card

In some cases it is preferable to store a part of someone's medical data on a smart card instead of storing it in the local database of a hospital. In case of an emergency the person carries its own medical data. The information stored has to be protected against unauthorized reading and writing. Only a limited number of people (for instance medical specialists) should be granted access to that part of the medical data which is necessary to perform their job. Before the card grants access to the data, it should authenticate the user.

3.1.6 Electronic Toll System

In some countries it is commonplace to demand toll for using certain roads. Nowadays, the toll fees have to be paid at tollbooths. This means that each car has to stop, with reduction of traffic flow as a consequence. Smart cards can be used to pay the toll while driving at normal speed.
The toll can be paid before or after driving on the toll road. In case of postpaid cards (like credit cards), the car driver will receive a bill at the end of the month. It is almost impossible to preserve anonymity of the vehicle. A recent study in Germany [2] shows that many people want to be anonymous when driving on a toll road.
In case of prepaid cards, however, the anonymity of the car driver can be assured.

3.1.7 Digital signature

Digital signatures are used in many cases. The main purpose of sending a message with a digital signature is to ensure that the message is written by the person who signed it and has not been tampered with. The main problem of using digital signatures is the storage of the keys. If a private key were stored on a computer, everybody who has access to that computer can copy it. If the private key were

stored on a smart card, nobody can copy it. However, the card can be lost or stolen. An online revocation list can be used to check if a key is still valid.

3.2 Threats

In section 3.1 several systems were discussed that deal with security. Each scenario is different and carries its own security threats. You can ask three questions for each: "Who is cheating?", "Why is he cheating?" and most important "How is he cheating?"
In the example of the access card three parties are involved: the legal card holder, the service provider and a third party (the burglar). The legal card holder possesses a card which gives him access to the door. For that door he has no reason to cheat; he has got access and that is all he wants. However, he may want to cheat with other doors to which he is not allowed access. It is practically impossible to add a key for another door. He has to know the door key. But even if the door key is known the legal card will undoubtedly refuse it to be added without proper authentication of the uploading terminal.
Most of time the service provider is considered trustworthy. He provides the door access system and the smart cards. He can build back doors in the system. For instance, a master key which can open all doors. But the service provider has no reason for cheating unless he is a burglar himself.
A burglar wants to gain illegal access to the building. He may have many reasons to do so, but we are not going to discuss the motives for burglary. Question remains, how he can cheat the system. He does not have a legal entrance card. It is pretty useless to make a fake one, since he does not know which keys are being used. Probably the only way to find the key is to try them all. A smart card is pretty slow compared to computers. He therefore can better use a laptop computer in combination with a hardware device that will fit into the door's slot.

As far as the software license card is concerned, the following parties are involved: the software developer, the card manufacturer, the legal software user and the illegal software user. Gaining rights to use the software is not an issue for the software developer himself. The card manufacturer has to be trusted. He sells "security" and wants to keep his reputation of being trustworthy. The legal software user has bought the software product and should not have to cheat. That leaves the illegal software user. His aim is to use the software without payment or to copy the software many times while buying it once. The first aim can be achieved by removing the smart card check from the software. Because this has nothing to do with smart card security it will not be dealt here. The second aim can be reached by copying the license key to a fraudulent card. The original smart card will be destroyed, but the new one can be copied freely, since it is written by the hacker himself.

In the case of an electronic purse many parties are involved. The bank usually is also the card issuer and the overall security system developer. Gaining trustworthiness is not easy because security is threatened by malicious persons in all the parties that are involved: merchants, card holders and thieves. The aim of all of them is making money.
A merchant decreases the card balance with the permission of the card holder. The card holder presses the "Yes" button (or sometimes its PIN code) to confirm the transaction. The merchant can than decrease the card balance and gain the money. Normally the transaction is carried out only once, but a malicious merchant will try to carry it out several times, each time gaining money. Another way to gain more money is to increase the amount after the customer's confirmation. The customer confirms the amount that is shown on the display, but a larger amount is being subtracted from the card.

The card holder has two possibilities for cheating: add money by faking a bank terminal or decrease the amount of a transaction.
A thief can act as a man-in-the-middle redirecting the money to its own purse.

Medical data is considered confidential. Different parties have access to different parts of the data. The card holder should be able to see all the data. He therefore has no reason to cheat. Third parties (insurance companies, for instance) may be interested in data to which they are not granted access. They will examine the firewall around the applet to find holes in it or back doors. Another approach can be to unseal the chip and read the EEPROM directly.

The parties involved in the electronic toll system are the government and the car driver. The government is responsible for installing the tool booths and distributing the smart cards. A frequent car driver may want to install a fake card into his car to avoid payment. The faked card simulates the official smart card.

3.3 Requirements

From the scenario's described above the following requirements can be distilled:

- Authentication should be possible in each direction: it should be possible for the card to authenticate the third party (the card should verify if the merchant is indeed a merchant) and for the third party to authenticate the card (the door checks if the card is a valid building pass). In the software license case the original license card should authenticate the other card before transferring its license key.
- Some information should be stored securely (the door key) while other information may be read by anybody (purse balance).
- A single object can be accessed by different parties. Access rights differ depending on the role in which the user is logged in.
- Communication between the client application and the smart card sometimes contains confidential information (health care data). Such a communication session should therefore be encrypted.
- Authenticity of data should be guaranteed when needed.
- Some methods should only be carried out once (purse transaction).
- Key management and key distribution are important issues. It is important where to store the various keys.
- It should be possible to specify all these security requirements without each time reinventing the wheel. The implementation of the application should be separated from the security. This results in cleaner code and the security has only to be verified once and for all. Of course it should be possible to specify which security requirements are necessary for each method. The best way to store this specification is in a separate file. This could be a text file written in some Interface Definition Language (IDL).

4 Design

The prerequisite of the design is to build a framework in which security issues are separated from the actual implementation. The framework is set up to simplify the task of building secure smart card systems. The only task an applet developer has to perform is writing the software like any other Java application and specifying the security requirements. The general design [12] is sketched in section 4.1. The security requirements specification is written in a separate Jason definition file (see section 4.3). The Jason framework will handle network communication (section 4.2) and data marshalling (section 4.4) as well as security (section 4.5) and naming (section 4.6).

4.1 General design

Secure Method Invocation (SMI) allows a caller object to securely call a method of a callee object. Both caller and callee are assumed to be stored and run in a protected environment that is called a sandbox. A surrounding firewall disables access to all objects within the sandbox except through published interfaces.
The Jason SMI layer provides the following services:

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

To call an object's method, the caller first has to connect to the callee in a particular 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. In fact, the skeleton is the card applet selectable by its Application IDentifier (AID). The session starts with selecting the applet and logging in. During login the identity of the role is checked and a session key is agreed upon. This session key will be used for encrypting the confidential data. The role keys are used when authenticity has to be guaranteed.
Once connected, the caller can call all methods declared by the object as accessible to this role. For Jason, roles are equivalent to keys. In other words, ownership of a particular key proves that an object can connect in that role.
The role keys used to authenticate the caller to the callee are stored in a separate key store object belonging to the same sandbox. The key stores on the card side and the client side contain the same roles, but do not have to be the same. When asymmetric encryption is used for authenticating a client, the public role key is stored on the card and the private role key on the client.
The stub and skeleton needed to securely call the methods of an object are generated automatically from a so called Jason Definition File (see section 4.3). This file specifies the security requirements for the callee object.

4.2 Network

Figure 8 shows the network topology that the Jason framework uses. Several smart cards and applications are connected by a network. The smart card as well as the application environment are supposed to run within a firewall. The application environment may physically be connected to the smart card reader, but they also may be separated by a network.
The application does not invoke the applet's method directly. Secure Method Invocation (SMI), like Remote Method Invocation (RMI), uses stubs and skeletons. RMI stubs and skeletons handle marshalling of parameters and return values, while

SMI also handles encryption. The SMI stubs and skeletons are connected with related key stores.

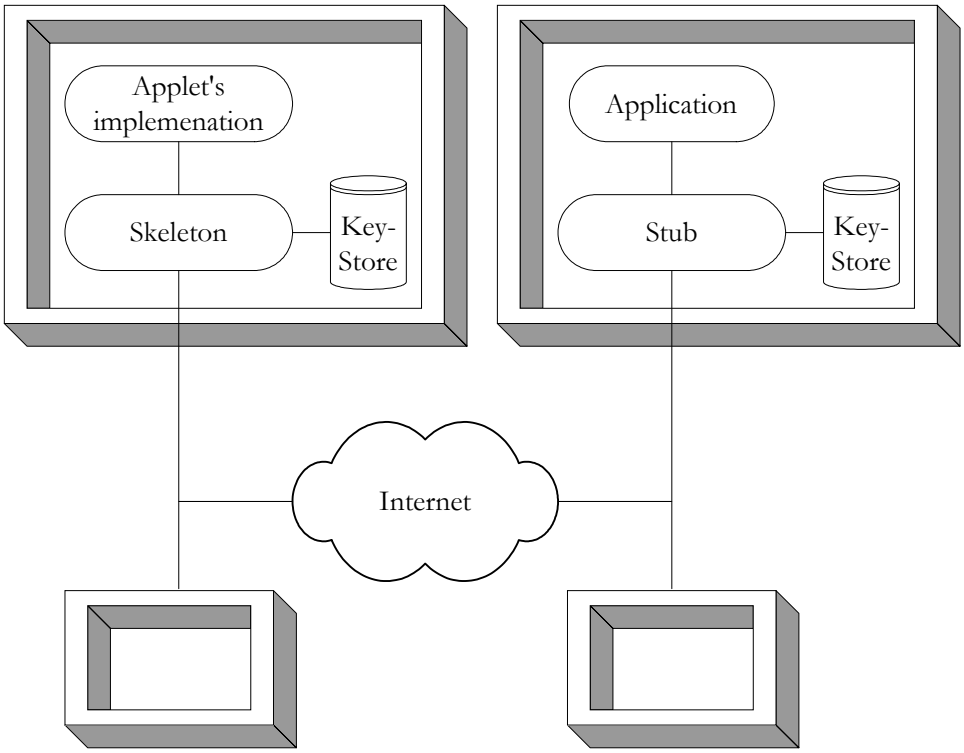

Figure 8 Topology

4.3 Jason definition file

In order to make the security protection transparent to the applet programmer, the Java language is extended by some keywords. The new keywords are **accessible to, authentic** and **confidential**. You can compare these with the standard keywords **private, protected** and **public**. These keywords only occur in the Jason interface definition file. The grammar of the Jason interface definition language is given in appendix A.1.

Authentic attributes are signed with a key corresponding to the given role name. The signature for all authentic attributes is appended to the message. Confidential attributes are encrypted using a symmetric session key.

The authenticity of the off-card application will be checked if the method is marked with the **accessible to** keyword. The card identity will then be checked too. The authenticity of the on-card methods will only be checked if the **authentic** keyword is present in the Jason modifier. The check occurs before the attributes are sent to the card. The method's result, if present, is signed by the applet.

4.3.1 accessible to keyword

The **accessible to** keyword is followed by a role name. A role name can be seen as an alias for an signature key. The keyword can be placed before the method definition (see listing 7). Only the third party that can be authenticated as Bank have rights to invoke the method. The authenticity of the card will also be checked.

```
class Purse {
  private short balance;

  accessible to Bank
  public void increase (short amount) {
    balance += amount;
  }
}
```

Listing 7 **accessible to** keyword

4.3.2 authentic keyword

The **authentic** keyword can be placed before the return value of the method or before one or more parameters. The data values marked as **authentic** are cryptographically signed ensuring that the data originates from the role. Freshness of an authentic method is ensured by signing a freshness counter. The counter is increased each time the signature key is used. This prevents re-invocation (increasing the purse balance several times, for example).

4.3.3 confidential keyword

Parameters and return values marked as confidential are encrypted using a symmetrical cipher. The session key is established during logging in.

4.4 Protocol Data Units

The application calls the stub's methods like any other Java object. The data between them is exchanged with normal parameters and return values. Since both the application and the stub reside in the same sandbox the communication has not to be secure.
In section 4.3 we have seen that four types of parameters exist: plain text, confidential, authentic and both confidential and authentic. These parameters are shown schematically in figure 9 as PP, CP, AP and ACP.

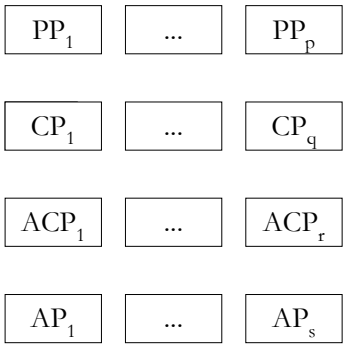

Figure 9 Different parameter types

The stub concatenates them to a large data block (see figure 10). See section 4.5 to see how the Confidential and Signature block are calculated.

L_p	PP_1	...	PP_p
L_c	Confidential		
L_a	AP_1	...	AP_s
Count	L_s	Signature	

Figure 10 All parameters concatenated

Each type of parameter is preceded by a length byte. The L bytes contains the lengths of all the parameter blocks.
This whole parameter block is embedded in the command APDU shown in figure 11. The APDU header is set to the default RMI invocation header: CLA=0x80, INS=0x38, P_1=0x02 (major version number), P_2=0x02 (minor version number). The JC-RMI header contains the object and method identifier. This command APDU is sent through the terminal to the card.

CLA	INS	P_1	P_2	L_c
JC-RMI header				
Parameters				

Figure 11 Command APDU

The command APDU shown in figure 11 is transformed by several card services that are called in sequence. First a decrypt service will transform it to the APDU shown in figure 12. Although not clear from the figure, the parameters are shuffled back to their original sequence. The decrypted command APDU conforms to the JC-RMI specification [7]. Therefore the standard RMIService object can then be used for the unmarshalling of the parameter block to the individual parameters. It also does the actual method invocation.

CLA	INS	P_1	P_2	L_c
JC-RMI header				
PP_1	...			PP_p
CP_1	...			CP_q
ACP_1	...			ACP_r
AP_1	...			AP_s

Figure 12 Decrypted command APDU

When the method invocation does not throw an exception, a normal response APDU is created. Figure 13 shows the layout of the APDU. The value of Tag byte is 0x81. The return value is the value returned from the invoked method. SW_1 and SW_2 form the status word. When no errors occur SW has a value of 0x9000. Other values indicate an error caused by the JavaCard virtual machine. When the invoked method causes an exception to be thrown, an exception response APDU (see figure 14) is created. A tag value of 0x82 indicates that the response APDU contains an exception. The type field identifies the exception. Only the standard exceptions have a number assigned. See for these numbers appendix B.5. Therefore no other exceptions should be thrown by the invoked method. The reason field further

specifies the exception. The status word SW will have a value of 0x9000 when no other errors have occurred besides the exception thrown by the invoked method.

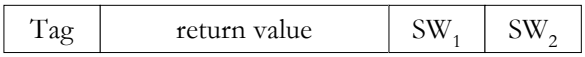

Figure 13 Normal response APDU

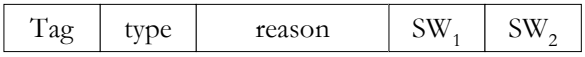

Figure 14 Exception response APDU

An exception response APDU is returned to the terminal without modification. A normal response APDU is eventually transformed by an encrypt service to an encrypted and/or signed form. Figure 15 shows all possible transformations: plain, authentic, confidential or both. The count field is used to guarantee freshness.

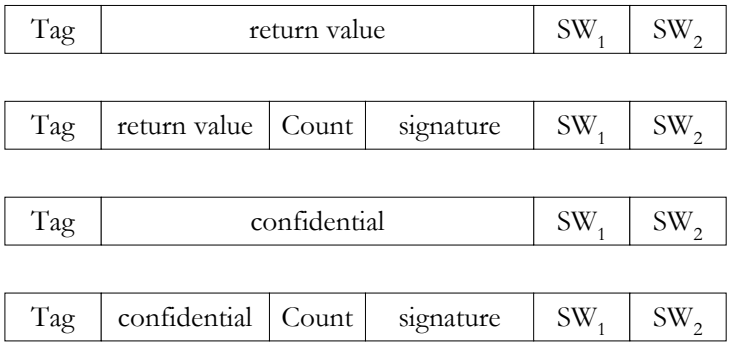

Figure 15 Transformed response APDU

4.5 Security

It is highly inefficient to encrypt each confidential parameter on its own. Many encryption algorithms use fixed size input blocks. DES for example uses input blocks of 8 bytes. If the input block is not a multiple of the block size, padding will be used. The last incomplete block will be appended by a padding block. But this means that if you have 5 confidential byte parameters a total of 35 (= 5 * 7) bytes are wasted. It is much more efficient to concatenate the 5 bytes together and append a padding of only 3 bytes. In order to concatenate the confidential parameters they should be shuffled so that they lay adjacent to each other. The single parameters of figure 9 are shuffled to the sequence shown in figure 16.

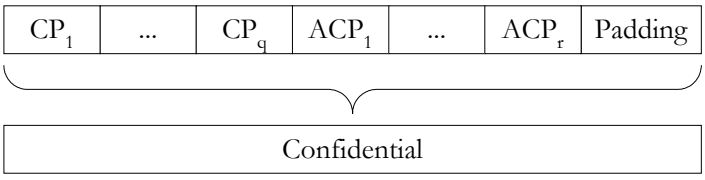

Figure 16 Confidential calculation

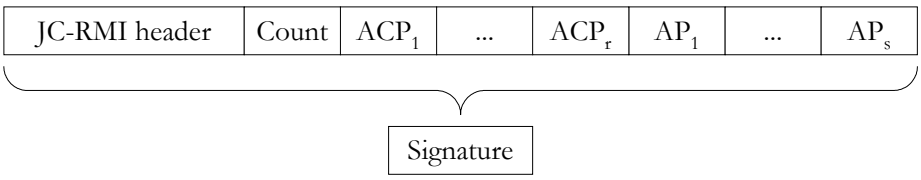

Figure 17 Signature calculation

The same reason for grouping the confidential parameters holds for the authentic parameters. A single signature is sufficient for all authentic parameters. This is sketched in figure 17. When the method is marked as accessible to a role the JC-RMI header and the freshness counter are signed too in order to prevent malicious modification of the object and method identification.
For return values the grouping is not necessary to shuffle because a return value only consists of a single value. For a confidential result this result will be encrypted with the session key. An authentic result is concatenated with the freshness counter and the signature.

4.6 Naming

In order to select an applet on the card, you have to know the Applet IDentifier (AID). The AID consists of 5 to 16 bytes. The first 5 bytes are called the Resource IDentifier (RID). ISO controls the assignment of RIDs to companies, with each company obtaining its own unique RID from the ISO [6]. The remaining bytes can be assigned by the company itself.
For the sake of simplicity an Applet Name Service will be used to map applet names to AIDs much like the more famous Domain Name Service (DNS) which maps host names to IP numbers.

5 Using SMI

It is pretty simple to write an applet using Jason. Normally an applet developer has to write a subclass of javacard.framework.Applet and handle the APDU exchange. When using Jason, the implementation can be written as a normal Java object. This implementation is free of APDU exchange and security issues. These issues will be handled by the Skeleton generated by the Jason pre-compiler. Before the pre-compiler can do its job, it has to know the security requirements. Therefore, the applet developer should supply a Jason interface definition file. The interface definition language that is being used in this file has been explained in section 4.3. The Jason file is the only input for the pre-compiler. It produces a Java interface, a Skeleton and a Stub (not displayed in figure 18). The Java interface is nothing else than the Jason definition file stripped from all keywords not known in Java. The implementation should implement this interface. The skeleton is the actual javacard.framework.Applet subclass.
The skeleton, the implementation and the interface will be compiled and uploaded as described in section 2.2.

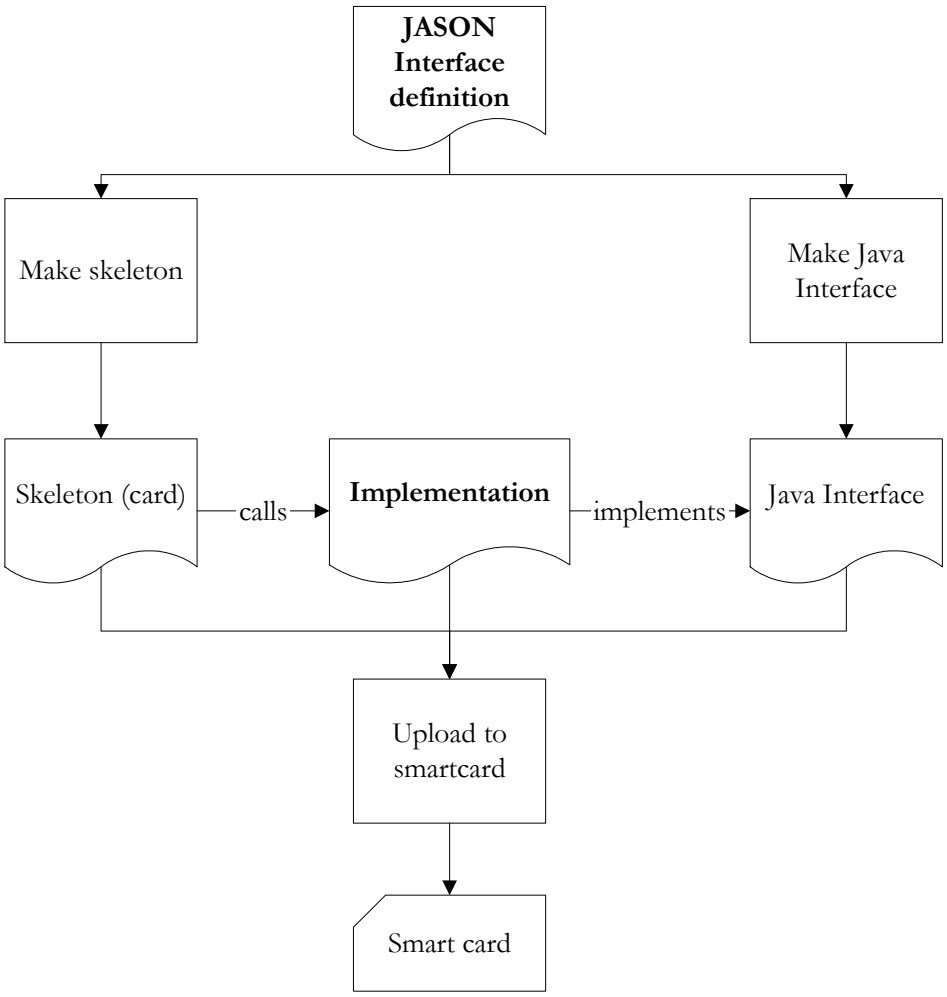

Figure 18 Skeleton generation

The generated stub is the counterpart of the skeleton. It too implements the generated interface (see figure 19). In the following sections we will give an example of the whole trajectory.

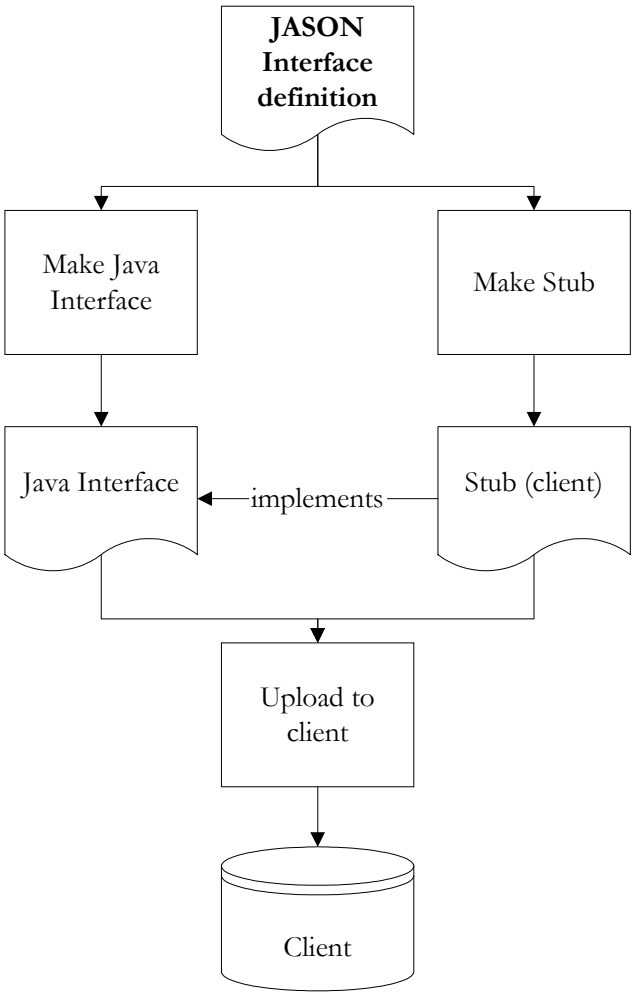

Figure 19 Stub generation

5.1 Writing JASON definition file

Let's take the electronic purse as an example. An electronic purse has three main functions. It should be possible to ask for the balance, increase it and decrease it. Let's name these functions *getBalance, increaseBalance* and *decreaseBalance*. Each function has a different security policy. Only the bank and the owner of the electronic purse should be able to get the balance. The increase function can only be accessed by the bank and the decrease function by a merchant. Listing 8 shows a possible Jason definition file. We saw that there are three parties that can access the purse: the card owner, the issuing bank and all merchants. In Jason these parties are called roles. The access to the *getBalance* function is restricted to the bank and the card owner. The result value, the actual balance, is considered non-confidential. The *increaseBalance* function can only be accessed by the bank. It carries a single parameter. The parameter should be signed because the value of the amount should not be tampered with. Changing the value of the amount will be detected, because then the signature is not valid any more. Furthermore, the bank considers the amount confidential. Nobody can tap the line between the bank and the card to read the amount of money that is deposited on the card. The *decreaseBalance* function is only accessible by a merchant. The amount parameter is authentic because the card owner should not be able to substitute the amount for a smaller amount.

```
package com.mybank;

import javacard.framework.UserException;

public interface Purse {
  roles MERCHANT, BANK, OWNER;

  accessible to OWNER, BANK
  public short getBalance();

  accessible to BANK
  public void increaseBalance(
    confidential authentic short amount)
  throws UserException;

  accessible to MERCHANT
  public void decreaseBalance(
    authentic short amount)
  throws UserException;
}
```

Listing 8 Jason definition file

5.2 Generating interface

The Jason pre-compiler gets the input file given in listing 8 and produces the Java interface given in listing 9.

```
package com.mybank;

import javacard.framework.UserException;

public interface Purse {
  public short getBalance();
  public void increaseBalance(short amount)
    throws UserException;
  public void decreaseBalance(short amount)
    throws UserException;
}
```

Listing 9 Generated interface

5.3 Writing implementation

The implementation of the generated interface is written by hand and shown in listing 10. Notice the way exceptions are thrown. Normally you throw an exception as **throw new UserException("Negative amount detected");**. On a JavaCard however, strings are not implemented. That's why UserException uses a reason parameter of type short. Furthermore, a JavaCard has no automatic garbage collection. It is perfectly legal to throw each time a new exception, but each exception will occupy a little amount of memory that will not be disposed. Therefore all JavaCard exceptions have a static throwIt function. It throws a static instance of the exception.

```
package com.mybank;

import javacard.framework.UserException;

public class PurseImpl implements Purse {
  public static final short
    NEGATIVE_AMOUNT = 1,
    NEGATIVE_BALANCE = 2;

  private short balance;

  public PurseImpl() { balance = 0; }

  public short getBalance() { return balance; }

  public void decreaseBalance(short amount)
    throws UserException {
    if (amount < 0)
      UserException.throwIt(NEGATIVE_AMOUNT);
    if (balance-amount < 0)
      UserException.throwIt(NEGATIVE_BALANCE);
    balance -= amount;
  }

  public void increaseBalance(short amount)
    throws UserException {
    if (amount < 0)
      UserException.throwIt(NEGATIVE_AMOUNT);
    balance += amount;
  }
}
```

Listing 10 Purse implementation

5.4 Generating Skeleton

Listing 11 shows the generated skeleton. The skeleton extends javacard.framework.Applet and is therefore selectable by sending a select APDU command. The security requirements that are lost in the transformation from the Jason Definition File (JDF) to the interface file are encoded in the JDF array. See appendix A.2 for the encoding being used. In the constructor a dispatcher is given a sequence of services. Before and after the purse service is called the Session service, that is initialised with an empty key store, will transform the APDU from secure data to plain text and vice versa.

```
package com.mybank;

import javacard.framework.*;
import javacard.framework.service.Dispatcher;
import javacard.framework.service.RMIService;

public class Purse_Skel extends Applet {
  private Dispatcher dispatcher;
  private static final byte[] JDF = {
    (byte) 0x03, (byte) 0xec, (byte) 0xa8,
    (byte) 0x01, (byte) 0x03, (byte) 0x03,
    (byte) 0x00, (byte) 0xe5, (byte) 0x8b,
    (byte) 0x01, (byte) 0x02, (byte) 0x00,
    (byte) 0x01, (byte) 0x33, (byte) 0x33,
    (byte) 0x7e, (byte) 0x01, (byte) 0x01,
    (byte) 0x00, (byte) 0x01, (byte) 0x23};

  public Purse_Skel() {
    dispatcher = new Dispatcher((short) 4);
    Purse purse = new PurseImpl();
    Session session = new Session(
      new KeyStore((short) 4), JDF);
    dispatcher.addService(session,
      Dispatcher.PROCESS_INPUT_DATA);
    dispatcher.addService(session,
      Dispatcher.PROCESS_COMMAND);
    RMIService rmiService = new RMIService(purse);
    dispatcher.addService(rmiService,
      Dispatcher.PROCESS_COMMAND);
    dispatcher.addService(session,
      Dispatcher.PROCESS_OUTPUT_DATA);
    register();
  }

  public static void install(
    byte[] buffer, short offset, byte length) {
    new Purse_Skel();
  }

  public boolean select() {
    return true;
  }

  public void process(APDU apdu)
    throws ISOException {
    dispatcher.process(apdu);
  }
}
```

Listing 11 Generated skeleton

5.5 Generating Stub

The generated stub file is almost equal to a stub file that is generated by the standard rmic compiler. The only difference is the presence of a JDF array similar

to the one used in the skeleton and a getJDF function. This function is specified by the Stub interface.

```
package com.mybank;

import java.rmi.Remote;
import java.rmi.server.RemoteStub;

public final class PurseImpl_Stub
  extends RemoteStub
  implements Purse, Remote, Stub {

  /*
  Code similar to code generated by the rmic
  compiler
  ...
  */

  private static final byte[] JDF = {
    (byte) 0x03, (byte) 0xec, (byte) 0xa8,
    (byte) 0x01, (byte) 0x03, (byte) 0x03,
    (byte) 0x00, (byte) 0xe5, (byte) 0x8b,
    (byte) 0x01, (byte) 0x02, (byte) 0x00,
    (byte) 0x01, (byte) 0x33, (byte) 0x33,
    (byte) 0x7e, (byte) 0x01, (byte) 0x01,
    (byte) 0x00, (byte) 0x01, (byte) 0x23};

  /* Specified by Stub interface */
  public byte[] getJDF() {
    return JDF;
  }
}
```

Listing 12 Generated stub

5.6 Personalization

The KeyStore that is instantiated in the skeleton (see listing 11) is initially empty. The internal array of keys is filled with null pointers. During the personalization phase keys can be downloaded to the card. The phase is irreversible. A role key can be downloaded once. Subsequent downloads result in a security exception.

5.7 Client application

The client application may look like listing 13. First an instance of the Applet Name Server is created. In the example given here the default values are used. The default values are stored in a properties file located somewhere in the classpath. The property file used here is given in listing 14. The host and port values specify the location of the terminal. The keystore value gives the location of the local keystore.

```
package com.mybank;

public class Client {
  public static void main(String[] args) {
    KeyStore keyStore = ...
    Ans ans = new Ans(keyStore);
    Purse purse = (Purse)
      ans.getApplet("jcrmi.server.Purse",
      Purse.ROLE_BANK);
    try {
      System.out.println("Balance: " +
        purse.getBalance());
      purse.increaseBalance((short) 25);
      System.out.println("Balance after increase: "
        + purse.getBalance());
    }
    catch (UserException ue) {
      switch (ue.getReason()) {
        case PurseImpl.NEGATE_AMOUNT:
          System.out.println(
            "You tried to increase or decrease " +
            "with a negative amount");
        case PurseImpl.NEGATIVE_BALANCE:
          System.out.println(
            "Negative balance not allowed");
      }
    }
  }
}
```

Listing 13 Client application

```
host = localhost
port = 8080
com.mybank.Purse = 0x33:0x04:0x00:0x00:0x00:0x00
```

Listing 14 Applet Name Server properties file

5.8 *Running*

Before being able to run the applet it should be compiled and uploaded to the smart card. This has been explained in section 2.2.

6 Implementation

In the previous chapters we have seen how the Jason system is designed and how it can be used. In this chapter we take a look at how the system is implemented. We start with a general overview before we dig into the details.

6.1 Layering possibilities

Naturally the Secure Method Invocation system (RMI) uses the JavaCard Remote Method Invocation (JCRMI). The question is: should we implement SMI on top of RMI or vice versa or SMI besides RMI (see figure 20)? The first option seems most natural, but both options have some disadvantages. By "SMI on top of RMI" I mean the situation in which the application uses the SMI layer while not using the RMI layer directly.

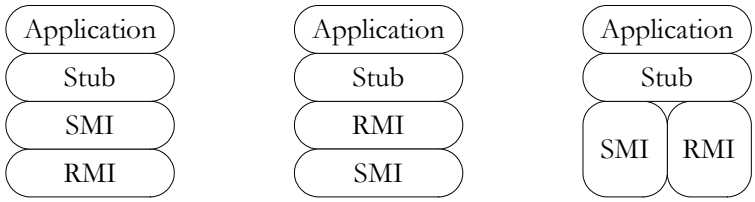

Figure 20 Layering possibilities

6.1.1 SMI on top of RMI

Let's start with the "SMI on top of RMI" situation. Because the Stub is generated by the Jason pre-compiler it can fully cooperate with the SMI layer. In theory they could be merged into a single layer, but that is not efficient. Consider multiple applets on a smart card. It is far better to have some small Stubs and a single (larger) SMI layer than multiple big Stubs. Therefore the stubs should be as small as possible. Because the SMI layer is static (that is, not generated), it does not have the knowledge of the security requirements of the applet. The stub has to give this information to the SMI layer. For example, it can send the method parameters along with the requirements in sequence to the SMI layer. The SMI layer then shuffles these parameters to create contiguous data blocks with the same security requirements. These blocks can be encrypted and/or signed and concatenated to a single byte array. This byte array is supplied to the RMI layer as a parameter of a remote method.
The big disadvantage of this layering possibility is the limited use of the RMI layer. The RMI layer is only used to transport a byte array from the client to the terminal. A lightweight socket connection could have solved the same problem. In fact the main part of the RMI layer, the marshalling and unmarshalling of parameters is done twice. The parameters are first marshalled to get a (secured) byte array and then this byte array is marshalled by the RMI layer into another byte array.

6.1.2 RMI on top of SMI

In paragraph 6.1.1 we encountered the problem of double marshalling. When the RMI layer is placed on top of SMI this problem can be solved. The parameters are first marshalled to an unsecured byte array in the RMI layer. The SMI layer transforms this to a secure byte array.
The RMI layer is used in the true sense: marshalling of parameters. However the SMI layer is supplied only with an unsecured byte array. It has no knowledge of the security requirements. It does not know which parameters should be encrypted and

which ones should be signed. Furthermore, where to look for the encryption keys? They cannot be supplied through the RMI layer.

6.1.3 SMI besides RMI

In paragraph 6.1.2 the problem of the double marshalling of paragraph 6.1.1 is solved creating a new one. This problem can be solved by placing the SMI and RMI layers besides each other. Before invoking remote methods, the SMI layer is initialised with a key store containing all the keys that the applet may use. The SMI layer consists of many classes that are shared with multiple clients. The object instances, however, are run within the client sandbox. At a remote method invocation request the SMI layer asks the stub for the security requirements. The generated stub implements the Stub interface. This interface specifies the *getJDF* method. An implementation of the Stub interface will return a byte array containing the security requirements of all methods of the stub. The grammar that is being used for the JDF array is given in appendix A.2. The SMI layer gets the object ID and the method ID from the marshalled data and searches the JDF array to find the security requirements.

6.2 Architecture

In section 6.1 we discussed the layering possibilities. The best way is placing the SMI and the RMI layers at the same level next to each other. Figure 21 shows the total architecture of both the client and the card. At the client side the application accesses two objects. Firstly it instantiates the Applet Name Service (ANS) and provides it a with key store. Secondly it asks the ANS for a stub object given a remote interface name and a role for logging in. After the obtainment of the stub object the application can use it like any other Java object.
The stub object calls JCRemoteRefImpl inside the JavaCard RMI layer in order to marshal the parameters to a byte array. Normally this byte array is given to a CardAccessor. A CardAccessor has an *exchangeAPDU(byte[] data)* method which transmits the data to the card accepting device in which the card is present. The generated card applet receives the data. Along with the JDF array it sends the data to the Session service which decrypts the data and checks the signature. The plain data conforms to the JavaCard RMI specification and therefore can be used directly by the RMIService. The RMIService unmarshals the data and calls the implementation. The response follows the opposite direction.

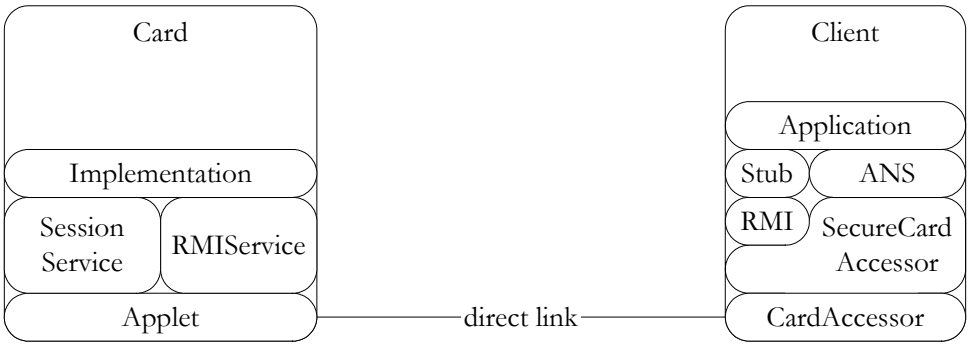

Figure 21 Direct connection

When the client is physically separated from the card accepting device by a network a slightly different architecture is being used (see figure 22). The CardAccessor itself has become a remote object. All objects are static accept the application and implementation which are written by the programmer and the Stub and the Applet which are generated by the Jason pre-compiler.

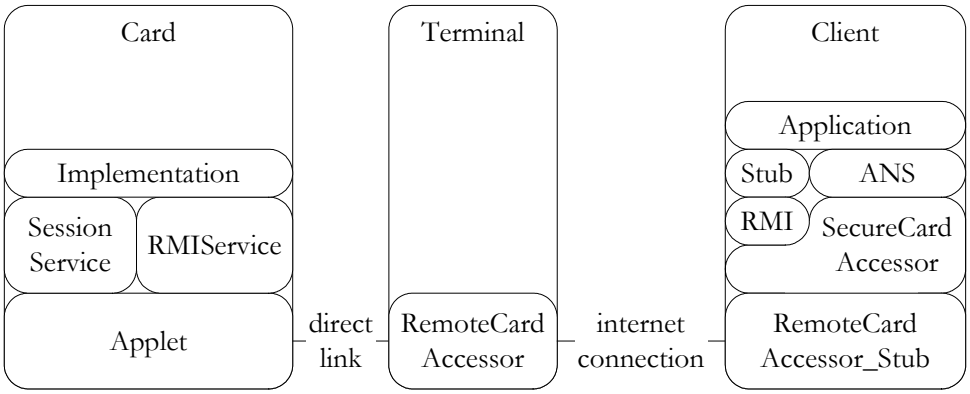

Figure 22 Internet connection

6.3 *Jason pre-compiler*

As discussed in section 6.2 the client stub and card applet are generated by the Jason pre-compiler. Not displayed in figure 22 is the interface that the client stub and the card implementation implement. This interface is generated too. All three files are generated by jason.compiler.Main. This object contains three methods, one for each file. They share the same JDF parser. The parser parses the .jason files into an intelligible format. Because most of the functionality resides in the static objects of the Jason framework, the tree generated files could be kept very small.

6.3.1 Java interface generator

The interface is very simple. It removes all the non-native Java keywords. It also adds constants for each role in the file. Unfortunately it is impossible to add the JDF array to the interface. For the client side there is no problem, but the interface file is removed by the CAP file converter. The CAP file converter only accepts constants if they are of primitive type (byte, boolean, short or int). That is the reason why the JDF array is located in both the card applet skeleton and the client stub.

6.3.2 Skeleton generator

The skeleton extends javacard.framework.Applet. Almost all code is static. Only the JDF array and the instantiating of the implementation object are variable. The JDF array follows the grammar of appendix A.2.

6.3.3 Stub generator

Of the three generated files the client stub is the most difficult one. It is also the biggest one, but that does not matter since it is run on a client computer with much more system resources than a smart card has. The stub file very much resembles a stub file that is generated with the standard rmic tool. There is only one difference: the stub implements the Stub interface. The *getJDF* method that is specified in the Stub interface returns the JDF array.

6.4 *Key management*

Keys are stored at the smart card as well as on the client computer. The way they are stored differs. At the smart card the keys are stored in a KeyStore object permanently, since JavaCard objects are persistent. At the client side the KeyStore object must be serialized to disk. It is the application's task to store the file at a secure place.

For each role in the Jason Definition File (see appendix A.1) a key is inserted in the key stores. When asymmetric encryption is used the private keys are stored at the

client and the public keys on the card. There is one role for which the opposite is true: the card role. The card role is not specified in the Jason Definition File but is used to authenticate the card. When symmetric encryption is used the same symmetric key is stored on the card as well as on the client.
To limit the code size of the card applet the keys are generated at a secure client station. The personalization phase consists of generating key(pair)s and downloading them to the card. The personalization phase is irreversible since keys can only be downloaded once.
The keys that are stored serve only for authentication purposes and for signing data. The encryption of confidential data is done with a symmetric encryption algorithm. The secret session key is generated on-the-fly by the card and given to the client after authentication has completed. Session keys are never stored at the client and placed in the RAM memory on the card, so that it will be removed when the card is ejected.

6.5 Naming

In section 4.6 the necessity of an Applet Name Service was discussed. In the current implementation of the Jason framework the Applet Name Service is implemented in the Ans object. This object reads a property file like listing 14 of section 5.6. In future versions of the framework the property file will be replaced by a network of Applet Name Servers like the international network of Domain Name Servers for solving Internet addresses.

7 Testing

Testing smart card application is a delicate task. An applet is not easily debugged. After downloading an applet to the card, the code is not visible any more. Therefore standard debugging environments cannot be used. The only way of communication with the applet is through APDU exchange. When an unexpected error occurs an ISO exception is thrown resulting in a status word of 0x6F00 which means "something went wrong". Furthermore most smart cards will destroy themselves when encountering unexpected behaviour. So after loosing 5 smart cards (of $ 20,-- each) I decided to do the rest of the testing with a simulator. Sun provides two separate simulators. They are discussed in section 7.1. Unfortunately, forced by the American export regulations, the simulators do not implement any security algorithms. In section 7.2 I will discuss my solution to this problem. In section 7.3 some tools are discussed. Unfortunately the JavaCard Development Kit is not free from bugs (section 7.4).

7.1 Two simulators

Sun provides two simulators [8]: JCWDE and C-JCRE. JCWDE is written in Java and C-JCRE in C.

7.1.1 JCWDE

The Java Card Workstation Development Environment (JCWDE) simulates the card environment. The JCWDE is not an implementation of the Java Card virtual machine, but it uses the Java virtual machine to emulate the JavaCard Runtime Environment. Debugging is straightforward, since standard debugging environments can be used. Furthermore it is possible to let applet do things that are normally not allowed, printing a String to the standard output for instance. But be careful to remove all this when actually downloading the applet to the card. It is not necessary to follow the whole trajectory sketched in figure 3 in section 2.2. You only need the class files.
The following aspects are not implemented in the JCWDE:

- package installation
- applet instance creation
- persistent card state
- firewall
- transactions
- transient array clearing
- remote method invocation
- applet deletion
- package deletion

Unfortunately remote method invocation is not implemented. Therefore this tool is useless for debugging the Jason framework which heavily depends on it.

7.1.2 C-JCRE

The C-language Java Card Runtime Environment (C-JCRE) implements the JavaCard virtual machine using the C language. Simulation with C-JCRE much more resembles the real situation than the JCWDE simulator. The C-JCRE acts as a real smart card. All communication to and from the C-JCRE is performed by APDU's. In contrary to JCWDE it is necessary to run through the whole trajectory of figure 3 in section 2.2. The scriptgen and apdutool are explained in more detail in section 7.3. C-JCRE simulates the EEPROM with a disk file. The state of the applet will be remembered between two sessions.

The latest JavaCard specification (2.2) contains a CrefCardTerminal. This class extends the OCF CardTerminal class. It makes it possible to plug the cref tool (implementing the C-JCRE) into the OCF framework [9]. This framework is used inside the JC-RMI implementation.

7.2 Crypto implementations

Due to export restrictions of the United States of America, the JavaCard Development Kit [8] is shipped without crypto implementations. Only the framework specified by the JavaCard specification is present. For example, there is a javacard.security.RandomData class, but the *getInstance* method will always generate a security exception, stating that no implementation is present. An actual smart card will be shipped with a different javacard.security.RandomData class that will implement the *getInstance* method correctly.
Not only the RandomData implementation is missing. Also the implementation of the Signature and the Cipher class is not available. While implementing the Jason security framework, the missing of the crypto implementation is not very practical. Therefore I had two options: buy lots of smart cards with crypto implementations[1] or write my own crypto implementation. On financial grounds I chose the latter. Since the crypto implementation is only necessary for testing in the simulator I did not spend much time to write real secure algorithms. Instead I use a simple XOR mechanism for both the Signature and the Cipher. It is totally insecure but at least it gives the possibility to test the security framework.

7.3 Scriptgen and Apdutool

All communication with a smart card is done with APDU's. Downloading a CAP file to the card is no exception. The scriptgen tool that is shipped with the JavaCard Development Kit [8] translates a CAP file into a sequence of command APDU's and stored in a script file. This script file is read by the apdutool which sends it to the C-JCRE. The scriptgen tool and apdutool only work in conjunction with C-JCRE. A real smart card may have a different applet installer that will use different command APDU's.

7.4 Bugs in the JavaCard Development Kit

The cref tool has an option to show the resource consumption. Before and after a card session the amount of memory that is allocated as well as the amount that is still free are shown. It is split into the different types of memory: stack, EEPROM, transaction buffer, RAM which is cleared on reset and RAM that is cleared on deselect. The JavaCard Development Kit 2.2 simulates cards with 384 bytes of stack, 16 kB of EEPROM (of which 2718 bytes are used for the virtual machine), 2 kB of transaction space, 256 bytes of clear-on-reset-RAM and 128 bytes of clear-on-deselect-RAM. The RAM part of the memory should be deallocated when the smart card is ejected (simulated by sending a power down command to cref that will exit then). However, the RAM memory is not freed when cref starts up. Only when the applet explicitly asks to free the memory with a call to JCSystem.requestObject-Deletion() the memory will be cleared the next time cref starts up. However, after uploading an applet cref allocates 163 bytes of clear-on-reset-RAM and 25 bytes clear-on-deselect-RAM. That leaves only 93 bytes of clear-on-reset-RAM and 103 bytes of clear-on-deselect-RAM. This memory is never cleared. Even when object deletion is requested. The memory leak is not due to the uploaded applet, since the same memory is lost when only the commands "Power Up" and "Power Down" are sent to cref. This bug has already been submitted to Sun.

[1] A large number of smart cards is necessary, since most errors lead to destroying the card.

Another deficiency in the JavaCard Development Toolkit is the failure of the native code. The javacard.framework, javacard.security and javacardx.crypto packages all use native code. The native code is entirely encapsulated in the cref tool. When using JC-RMI some parts of the packages are used also at the client side. However on the client side the native codes are not present. There should have been a dynamic link library (Windows) or a .so package (Linux / Solaris) containing the native code. You can proof the failure of the native code by throwing an exception on the card. The JC-RMI specification [7] tells us that the exception is transported by value to the client where it is thrown again. But throwing a UserException for instance results in an UnsatisfiedLinkError.
I have solved this deficiency by writing parts of the missing native code myself. The native code is written in C++ and compiled to NativeMethods.dll. This DLL is statically linked to the Applet Name Server by loading it with System.load-Library("NativeMethods"). I hope that Sun will ship the native codes in their next JavaCard Development Kit.

8 Conclusions and future work

The prerequisite of the Jason framework was to make the development of a network of cooperating JavaCard applets as simple as possible. In section 5 we followed a real world example through all the steps. Most steps are automated by using the Jason pre-compiler. A solid design has resulted in a lightweight framework in which the generated skeletons and stubs are kept as small as possible. The main part of the framework consists of a static collection of objects that does not have to be rebuild for each applet. This increases the efficiency when uploading multiple applets to a single card.
The framework has been revised entirely to make use of the new JavaCard remote method invocation specification. In June 2002 Sun introduced with the new JavaCard 2.2 specification [6,7,8]. Before that time the Jason framework used its own kind of remote method invocation which was not compatible with the new standard. Also new in version 2.2 was the existence of card services and dispatchers. In the old specification there was only one logical channel between a card and its terminal. In the new specification up to four logical channels may exist. The Jason framework has not yet been revised to take advantage of this. In future versions of the Jason framework multi-tasking may be added.
Some other modifications to the Jason framework are possible:

- Modification of role keys after personalising the card can be made possible. The *putKey* method of the KeyStore object can be protected much like other methods by adding a *accessible by* keyword.
- The Applet Name Server can be extended. The local properties file containing the applet names and the AID's can be substituted to a system like the Domain Name Service. AID's are managed by ISO and are globally unique. There is an apparent similarity between AID and IP numbers. This Applet Name Server can also provide the stubs. A remote class loader may obtain the stubs from the server instead of searching for them in the local classpath.
- To make the applets as small as possible, I chose to store only one key per role. When memory of smart cards increases, multiple keys can be stored for each role. Each key can be used with its own encryption algorithm. Before connecting to a card applet a handshake can take place in which encryption algorithm are agreed upon.

The secure method invocation scheme of the Jason framework [12] will be presented at the CARDIS 2002 conference. The conference will be held in San José at November 21 and 22.

9 References

[1] T. Aura and D. Gollman, *Software License Management with Smart Cards*, in USENIX Workshop on Smartcard Technology, 1999

[2] W. Rankl and W. Effing, *Smart Card Handbook*, second edition, 2000

[3] http://java.sun.com/products/javacard

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

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

[6] Sun Microsystems Inc, *JavaCard™ 2.2 Virtual Machine Specification*, June 2002

[7] Sun Microsystems Inc., *Java Card™ 2.2 Runtime Environment (JCRE) Specification*, June 2002

[8] Sun Microsystems Inc., *Java Card™ 2.2 Development Kit User's Guide*, June 2002

[9] OpenCard Framework, http://www.opencard.org/

[10] D. Hagimont, J.-J. Vandewalle, *JCCap: capability-based access control for Java Card,* CARDIS 2000

[11] J.-J. Vandewalle, E. Vétillard, *Developing Smart Card -Based Application using Java Card*, 3rd Smart Card Research and Advanced Applications Conference, September 1998

[12] R. Brinkman, J.H. Hoepman, *Secure Method Invocation in Jason*, University of Twente and Nijmegen, November 2002, to be presented in CARDIS 2002 conference proceedings

A Jason Definition File Grammar

The Jason Definition File can be specified in a text file with the jason extension. The grammar is given in A.1. The text file is parsed by the Jason pre-compiler. Internal the Java Definition File is represented in a byte array. The grammar for this array is given in A.2.

A.1 .jason file format

```
interface      -> java_modifier "interface"
                  <interface_name> "{" member* "}"
java_modifier  -> scope ["final"]
scope          -> ["public" | "protected" |
                   "private"]
member         -> variable | method | roles
variable       -> scope "static final" type
                  <variable_name>
method         -> "public" jason_modifier type
                  <method_name> "(" attributelist
                  ");"
jason_modifier -> ["accessible to" rolelist]
                  security
roles          -> "roles" <rolelist> ";"
rolelist       -> <role_name> ["," rolelist]
security       -> ("confidential" | "authentic")*
attributelist  -> attribute ["," attributelist]
attribute      -> security type <attribute_name>
type           -> ("byte" | "boolean" | "short" |
                   "int") ["[]"]
```

Listing 15 Jason interface definition grammar

A.2 JDF array

```
jdf {
  u1 number_of_methods
  method[number_of_methods] methods
}

method {
  u2 method_id
  u1 number_of_roles
  u1[number_of_roles] role_id
  u1 return_modifier
  u1 number_of_parameters
  u1[number_of_parameters] parameter_modifiers
}
```

Listing 16 JDF array

The format of all modifiers follows the same bit structure. The least significant nibble specifies the type:
-----***: 000=void, 001=byte, 010=boolean, 011=short, 100=int, 101=object
----*--- : 1=array
The most significant nibble specifies the security requirements:
--**----: 00=plain, 01=confidential, 10=authentic, 11=confidential+authentic

B JCRMI data formats

JCRMI uses two types of command APDU's for selecting an applet and for invoking a method. The structure is specified in sections B.1 and B.2. The rest of appendix B specifies the grammar for all types of parameter encoding and response messages.

B.1 Select APDU command format

CLA 000000xx - The least significant two bits are used for logical channels
INS 0xA4 - SELECT FILE
P1 0x04 - Select by AID.
P2 000x00xx - Return FCI information. The bits b2 and b1 are used for partial selection if supported. If bit b5 is 1, the remote reference descriptor uses the remote_ref_with_interfaces format, otherwise it uses the alternate remote_ref_with_class format.
Lc xx - Length of the AID
Data - AID of the applet to be selected (between 5 and 16 bytes)

```
select_response {
  u1 fci_tag = 0x6F
  u1 fci_length
    u1 application_data_tag = 0x6E
    u1 application_data_length
      u1 jc_rmi_data_tag = 0x5E
      u1 jc_rmi_data_length
      u2 version = 0x0202
      u1 invoke_ins
      union {
        normal_ref_response normal_initial_ref
        normal_null_response null_initial_ref
        error_response initial_ref_error
      } initial_ref
}
```

B.2 Invoke APDU command format

CLA 1000xxxx - b4 and b3 for secure messaging (ISO 7816-4) and b2 and b1 for logical channels
INS - invoke_ins returned in the previous select_response
P1 02 - RMI major version #
P2 02 - RMI minor version #
Data - As described below
The data part of the request command is structured as:

```
invoke_data {
  u2 object_id
  u2 method_id
  param parameters[]
}
```

B.3 Remote Object Reference Descriptor

```
remote_ref_descriptor {
  union {
    ref_null remote_ref_null
    remote_ref_with_class remote_ref_c
```

```
    remote_ref_with_interfaces remote_ref_i
  }
}
ref_null {
  u2 remote_ref_id = 0xFFFF
}
remote_ref_with_class {
  u2 remote_ref_id <> 0xFFFF
  u1 hash_modifier_length
  u1 hash_modifier[ hash_modifier_length ]
  u1 pkg_name_length
  u1 package_name[ pkg_name_length ]
  u1 class_name_length
  u1 class_name[ class_name_length ]
}
remote_ref_with_interfaces {
  u2 remote_ref_id <> 0xFFFF
  u1 hash_modifier_length
  u1 hash_modifier[ hash_modifier_length ]
  u1 remote_interface_count
  rem_interface_def
    remote_interfaces[remote_interface_count]
}
rem_interface_def {
  u1 pkg_name_length
  u1 package_name[ pkg_name_length ]
  u1 interface_name_length
  u1 interface_name[ interface_name_length ]
}
```

B.4 Parameter encoding

```
param {
  u1 value[]
}
boolean_param {
  u1 boolean_value
}
byte_param {
  s1 byte_value
}
short_param {
  s2 short_value
}
int_param {
  s4 int_value
}
null_array_param {
  u1 length = 0xFF
}
boolean_array_param {
  u1 length <> 0xFF
  u1 boolean_value[length]
}
byte_array_param {
  u1 length <> 0xFF
```

```
  s1 byte_value[length]
}
short_array_param {
  u1 length <> 0xFF
  s2 short_value[length]
}
int_array_param {
  u1 length <> 0xFF
  s4 int_value[length]
}
```

B.5 Return value encoding

```
return_response {
  u1 tag
  u1[] value
}
normal_param_response {
  u1 normal_tag (= 0x81)
  param normalValue
}
normal_null_response {
  u1 normal_tag (= 0x81)
  ref_null null_array_or_ref
}
normal_ref_response {
  u1 normal_tag (= 0x81)
  remote_ref_descriptor remote_ref
}
exception_response {
  u1 exception_tag = 0x82
  u1 exception_type
  s2 reason
}
java.lang.Throwable = 0x00
java.lang.ArithmeticException = 0x01
java.lang.ArrayIndexOutOfBoundsException = 0x02
java.lang.ArrayStoreException = 0x03
java.lang.ClassCastException = 0x04
java.lang.Exception = 0x05
java.lang.IndexOutOfBoundsException = 0x06
java.lang.NegativeArraySizeException = 0x07
java.lang.NullPointerException = 0x08
java.lang.RuntimeException = 0x09
java.lang.SecurityException = 0x0A
java.io.IOException = 0x0B
java.rmi.RemoteException = 0x0C
javacard.framework.APDUException = 0x20
javacard.framework.CardException = 0x21
javacard.framework.CardRuntimeException = 0x22
javacard.framework.ISOException = 0x23
javacard.framework.PINException = 0x24
javacard.framework.SystemException = 0x25
javacard.framework.TransactionException = 0x26
javacard.framework.UserException = 0x27
javacard.security.CryptoException = 0x30
```

```
javacard.framework.service.ServiceException = 0x40
exception_subclass_response {
  u1 exception_subclass_tag = 0x83
  u1 exception_type
  s2 reason
}
error_response {
  u1 error_tag = 0x99
  s2 error_detail
}
```