JCCap: capability-based access control for Java Card

D. Hagimont
SIRAC Project
INRIA, 655 av. de l'Europe,
38330 Montbonnot Saint-Martin, France
Daniel.Hagimont@inrialpes.fr

J.-J. Vandewalle
Gemplus
BP 100
13881 Gémenos cedex, France
jeanjac@research.gemplus.com

Abstract: This paper describes JCCap, a protection facility for cooperating applications in the context of Java Card. It enables the control of access rights between mutually suspicious applications, either between one terminal application and one Java Card applet or between two applets hosted inside the same Java Card.

Using JCCap, access to objects is controlled by means of software capabilities that can be exchanged between mutually suspicious applications.

An important advantage of JCCap is that the definition of the protection policy of an application (*i.e.*, how access rights are granted to other applications) is completely separated from the application code. The protection policy is described in an extended Interface Definition Language (IDL) at the interface level, thus enhancing modularity, separation of concerns, and ease of expression in the design of the overall security architecture. Each application can define its own protection policy independently from the other applications, thus enabling the expression of mutual suspicion without any prior knowledge about the policies of other applications. Every protection policy is then applied when applications interact with each other.

This paper describes the implementation of a prototype of JCCap and experiments with simple Java Card-based applications. Reported results show the feasibility and applicability of this technique in today's Java Card and outline its advantages.

1 Introduction

With the advent of open smart cards, it becomes possible to embed multiple applications within one smart cards[Digiorgio99]. These applications may need to cooperate with other applications, either co-located within the same smart card (we call them *local applications*), or located in the terminal[1] in which the card is inserted (we call them *remote applications*).

More precisely, in the context of Java Card [Siddalingaiah97], two local applications can cooperate through Java method invocation on "shareable" objects, and two remote applications can cooperate through a remote invocation mechanism such as DMI [Vandewalle98], an equivalent of the Java standard RMI mechanism [Sun98b] dedicated to Java Card. Thanks to the Java programming language and to the DMI facility, the development of smart card-based applications is greatly simplified.

However, enabling cooperation between applications requires mechanisms to control the access rights that one application grants to its peers. Naturally, with Java Card, an access control policy should be expressed in terms of checks on which methods one application may or may not invoke on objects of another application. The current release of Java Card specifications only defines a mechanism for sharing objects between local applications but does not provide any integrated facility for managing the access control policy associated with those shared objects. That implies that the application programmer must explicitly manage such a policy. Such a burden makes applications difficult to develop and maintain as it implies to mix the functional application code with security-related code, such as authentication and access right checks, and explicit access right transfers. In this paper, we propose a protection facility called JCCap, which addresses the above issue. It is based on software capabilities [Levy84] and therefore enables mutually suspicious applications to dynamically exchange access rights according to their protection policy and their execution context.

JCCap has the following advantages:

- Mutual suspicion: JCCap supports mutual suspicion in two ways. First, it allows an application to *dynamically* grant access rights to another application. Therefore, the access rights that the application must grant at the bootstrap of the execution can be restricted to a minimum; access rights are granted on demand, following the well-known "need-to-know" principle. Second, each application is responsible for the definition of its own protection policy, which is *transparently* taken into account at run-time. Therefore, there is no need for an *a-priori* definition of a global protection policy agreed by all the applications. Applications that are mutually suspicious can each define their own protection policy independently of the others. At run-time, applications can discover themselves and cooperate by *dynamically* granting access rights while *transparently* checking accesses according to their protection policy.
- Modularity: JCCap brings modularity in the application design since the definition of protection policies is totally separated from the application code.
- Transparency: the expression of a protection policy is not impacted by the location of the involved applications. The involved applications can be local (within the same card) or remote (one in the terminal and one within the card).

JCCap has been implemented on top of the Java Card 2.1 environment. It consists in a stub generator which generates *filter objects*, which are responsible for implementing the access controls

[1] From now on, we call *terminal* the host in which the card is inserted.

associated with the application protection policy. This paper presents the protection model of JCCap and its implementation in the Java Card environment.

The rest of the paper is structured as follows. In section 2, we motivate the overall design choices for JCCap. We present JCCap's access control model in section 3. Section 4 describes the implementation of JCCap. Section 5 exhibits our experiments using JCCap and we conclude the paper in section 6.

2 Overall design choices

In order to motivate our work, we first present a simple application example that is used to identify the requirements for the design of the JCCap facility. We then introduce the capability-based protection model, on which JCCap relies. Finally, we discuss the rationale of our choice for a capability-based protection mechanism in regards with the existing literature and the problem of its implementation costs.

2.1 Application example

Let's consider the example of a *Bank* application that manages clie nt accounts (Figure 1).

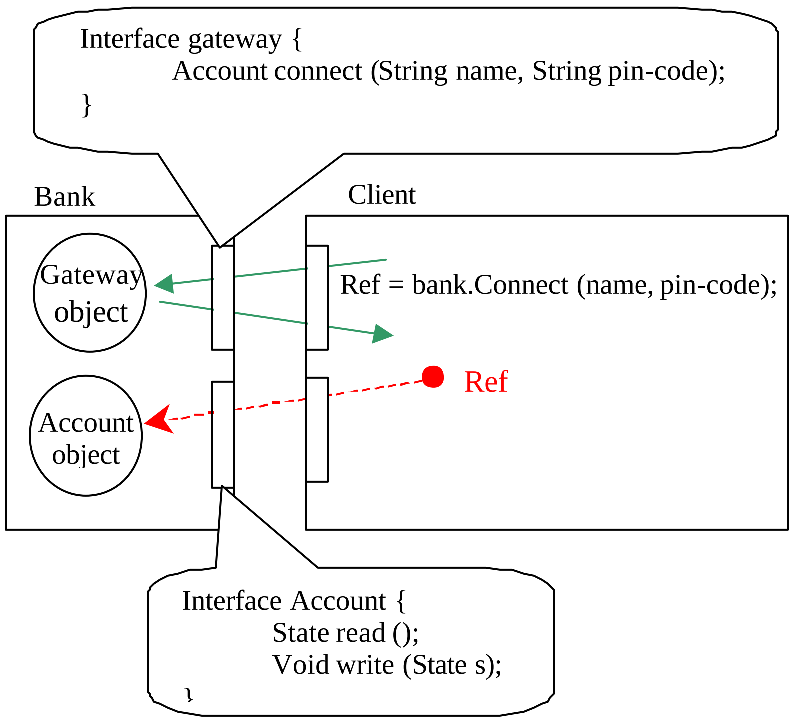

Figure 1. The Bank example

This application has to cooperate with *Client* applications for which the accounts are managed. Whenever a Client[2] connects to the Bank through a *Gateway* object, the Bank returns to the Client a reference to its *Account* object, allowing the client to read the state of the Account. Each application (Bank and Clients) knows the cooperation interfaces, *i.e.* the interfaces of the Gateway and Account objects. The interface of the Gateway object allows the Client to connect to the Bank, providing the client name and the pin-code associated with that client name. The *connect* method returns a reference to the client Account object. The interface of the Account object defines two methods, which respectively allows reading and writing the state of the account (we use the syntax of the Java programming language).

Let's study the requirements in terms of access control in this application example. The Bank has full access right on its own objects, but clients should not. A client should be allowed to read its bank account (and only its), but not to arbitrarily write it (only a bank transfer initiated by the bank should be granted write access on the account).

Therefore, when the bank returns a reference to the account object to the client, this reference should only allow the client to read the account. This reference should only include restricted access rights on the account object. Moreover, a client should not be granted access to the account object as long as it doesn't obtain such a (protected) reference to the account object through the *connect* method of the Gateway object. The *connect* method requires the client to provide the correct password (*pin-code*) associated with the account.

The above application example shows that access control between cooperative applications should allow dynamic evolution of access rights. It is not possible to define statically (prior to applications execution) the access rights required by applications to execute properly. In the Bank example, a client application may acquire an access right on the account object, provided it passed the correct *pin-code* parameter to the *connect* method. This implies that the access control system must allow access rights to evolve dynamically during execution. Transferring an access right from one application to another generally occurs when applications interact and more precisely when one application provides a reference on one of its objects to another application. The reference passing has to be accompanied by a right transfer.

These requirements led us to the definition of a capability-based access control model. The next subsection introduces capability-based access control, before the description of the JCCap model in section 3.

2.2 Capability-based access control

The JCCap model is based on software capabilities [Levy84]. The advantage of capabilities is that they allow access rights to evolve dynamically, which is one of our objectives.

A capability is a token that identifies an object and contains access rights, i.e. the subset of the object's methods whose invocation is allowed. In order to access an object, an application must own a capability to that object with the required access rights. When an object is created, a capability is

[2] From now on, the term *client* refers to the *client application* (not a person).

returned to the creator, that usually contains all rights on the object. The capability can thus be used to access the object, but can also be copied and passed to another application, providing it with access rights on that object. When a capability is copied, the rights associated with the copy can be restricted, in order to limit the rights given to the receiving application.

Therefore, each application executes in a protection environment in which it is granted access to the objects it owns. This application can obtain additional access rights upon method invocation. When an object reference is passed as parameter of an invocation, a capability on that object can be passed with the parameter in order to provide the receiving application enough access rights to use the reference.

In a Java environment, a capability may be viewed as a Java reference with restricted access rights. A system operation should allow the restriction of the access rights associated with such a reference, which can then be passed as a parameter when an object from an untrusted application is invoked.

In order to illustrate capability-based protection, let us consider the Bank example described previously. A capability on the *Gateway* object is given to the client applications providing them with the right to *connect* to the bank. When a client wants to read its account, the client connects to the bank using this capability. In return from the *connect* invocation, the client receives a capability on the account object, which only allows reading the account. This capability allows the client to read its account.

However, even if managing capabilities simplifies protected applications development, the access control policy of an application still has to be programmed in the code of the application, thus leading to complex programs. Our goal was to separate the access control aspect from the implementation aspect of the application. This simplifies the expression of an access control policy, keeps application code simple and enforces modularity. The JCCap capability-based model is presented in the next section.

2.3 Rationale

Capability-based protection mechanisms have been defined and implemented in a variety of systems [Levy84, Tanenbaum86, Richardson92] including the Java environment [Goldstein97]. However, in all the proposed approaches, capabilities are made available at the programming language level through capability variables that are used explicitly for accessing objects, changing protection domains and transferring access rights between protection domain. Therefore, it is demanded a programming effort to implement them in a particular application. Moreover, this binding of the protection mechanism with the implementation code does not help a clear cut separation between the security policy and its control at the runtime.

Our JCCap capability-based model rely on the *Hidden Software Capabilities* defined and developed by one of the author [Hagimont96]. The hidden software capabilities technique overcome the difficulties of capability-based systems as presented in the section 3. We have extended the application of this technique to the Java Card environment in two directions:

- Protecting the access to objects in different card applet contexts (cf. section 4.1) with an implementation of hidden software capabilities with Java Card 2.1 [Sun99] shared objects (objects that implements a *Shareable* interface).
- Protecting the access to object of a card applet from the terminal application (cf. section 4.2) with an implementation of hidden software capabilities on top of a Java Card RPC-like communication scheme developed by the other author [Vandewalle98].

These two applications of the hidden software capabilities have been studied in order facilitate the definition and the implementation of complex protection schemes with Java Card that will surely face with this problem in a near future. Today's, implementing a protection scheme with off-the-shelf Java Card 2.1 platforms is difficult and requires a lot of code to be hand-coded and added to the applets (shareable interfaces, implementation of shared objects, authentication tool, *etc.*). Our solution does not eliminate such addition of code to applets but automates their production and therefore enables more reliable code and fastens its development cycle.

3 The JCCap capability-based access control model

In this section, we present our protection model based on software capabilities.

3.1 The model

As explained in the previous section, software capabilities provide a model in which access right can be dynamically exchanged between applications. The issue is then to provide applications programmers with a means for controlling rights exchanges with other applications.

One strong motivation for the JCCap model is modularity. Indeed, we don't want to provide extensions to the programming language that allow an application to express capability parameter passing when an object from another application is invoked. This would overload programs and make them much more difficult to maintain.

To achieve this goal, our idea is to define capability exchanges between interacting applications using an interface definition language (IDL) [Hagimont96]. Since an interface can be described independently from any implementation, describing capability exchanges at the level of the interface allows the protection definition to be clearly separated from the code of the application, thus enhancing modularity.

Therefore, an IDL has been defined that allows the application programmer to express the capabilities that should be transferred along with parameters in a method invocation. This IDL allows the definition of *views*. A view is an interface that includes the definition of an access control policy. A view is associated with a capability and describes:

- the methods that are authorized by the access rights associated with the capability,
- the capabilities that must be transferred between the caller and the callee along with the parameters of the methods authorized by the view. These transferred capabilities are expressed in terms of views.

Therefore, a capability is structured as shown in figure 2. It includes the identifier of the object, the access rights that the capability provides to its owner and the capability exchange policy which defines what capabilities must be passed along with parameters when the object is invoked. The access rights and the capability exchange policy are defined with a view.

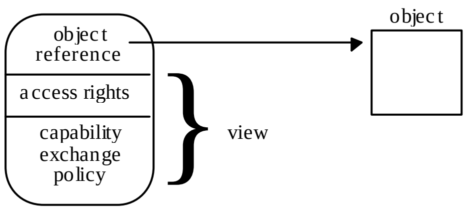

Figure 2. Structure of a capability

The definition of views is naturally recursive since it specifies the capabilities that should be transferred with parameters, this specification being in terms of view. For that reason, each protection view is given a name at definition time.

In the Bank example described above, two views may be associated with an *Account*: a view *reader_account* that only grants access to the *read* method and a view *writer_account* that grants access to both methods *read* and *write*. For the *Gateway* class, we define the view *client_gateway* which authorizes invocation of the *connect* method, which signature in the view expresses that a capability with the *reader_account* view must be returned to the caller application along with the reference to the account object returned as result of *connect*. These views are described below:

```
view reader_account implements Account {
        State read();
        void not write (State s);

}
```

```
view writer_account implements Account {
        State read();
        void write (State s);

}
```

```
view client_gateway implements Gateway {
        reader_account connect ( String name, String pin -code);
}
```

Such a protection policy, defined only on the callee side, would be sufficient if we were considering a client/server architecture where protection is only there to protect the server against its clients. Instead, we are considering an architecture where applications are mutually suspicious. Each application must have full control over the capabilities it exports to other applications (each application may be a caller or a callee). Moreover, we want to ensure applications independence. More precisely, it is not possible for an application programmer to verify the protection policy defined by an application that exports a service since at programming time, the programmer may not yet know which applications it is going to interact with.

For these reasons, each application can define its own view of the protection policy to apply when interacting with other applications. Therefore, two views are associated with a capability: the view of the caller application and the view of the callee application.

The view defined by the callee *Z* describes:

- The methods that are authorized.
- For each input parameter of a method (reference *R* received by *Z*), the view describes the capabilities that are given by *Z* when the reference *R* is used for method invocation. This view describes, from the callee point of view, the capabilities that it accepts to export.
- For each output parameter of a method (reference *R* given by *Z*), the view describes the capability returned with the reference *R*.

and similarly the view defined by the caller *A* describes:

- For each input parameter of a method (reference *R* given by *A*), the view describes the capability given with the reference *R*.
- For each output parameter of a method (reference *R* received by *A*), the view describes the capabilities that are given by *A* when the reference *R* is used for method invocation. This view describes, from the caller point of view, the capabilities that it accepts to export.

This symmetric scheme is the answer to mutual suspicion and applications independence. Both the caller and the callee specify their protection views for their objects. They are taken into account as follows.

In order to share objects, applications must exchange object references (Java object references in the context of Java Card). Thus, the runtime must provide a name server that allows objects references to be exchanged, *i.e.* to associate symbolic names with object references. We assume that this name server is used by applications in order to start cooperating[3]. When an application obtains a reference to an object from the name server, it can invoke the object by using the Java interface that this object is supposed to implement (the two applications must agree on an interface in order to cooperate). Then, applications can exchange references as parameters (onwards or backwards) without using the name server.

When an application exports an (Java) object reference through the name server, it defines the view associated with the reference, i.e. the capability that is exported for this exported reference. This way, the applicatio n also defines the capabilities that may be exported subsequently to an invocation of that object.

When an application fetches the reference from the name server, it also defines the view associated (on its side) with the reference it obtained. This way, the application defines the capabilities that may be exported subsequently to an invocation of the object.

Any invocation that derives from an invocation on that object will take into account the view definitions from both interacting applications.

3 This name server must also include an authentication mechanism in order to select to which application a capability can be delivered.

3.2 Example

In order to illustrate the expression scheme of JCCap, let us consider the example[4] of a *Printer* object, exported by a print server, that allows a client to print a file (Figure 3).

A capability on the *Printer* object is given to the client application providing them with the right to print files. When a client wants to print a file (*File* object), the *Printer* object needs to get read rights for this file; therefore the client will pass, at invocation time (1), a read-only capability on the file (*file_capa*) to the callee application. This capability allows the *Printer* object to read the contents of the file (2).

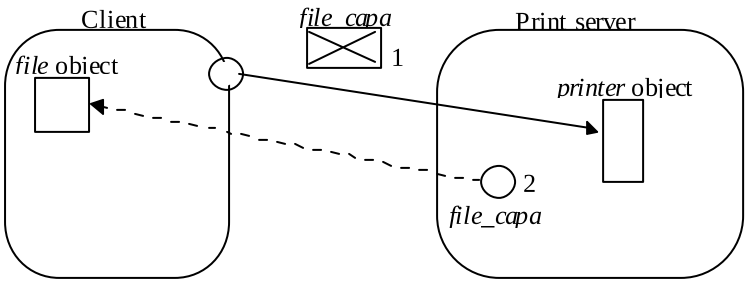

Figure 3. Print server example

Here are the Java interfaces of the *Printer* application.

```
interface Printer_itf {
  void init ();                          // initialize the printer
  Job_itf run (Text_itf text);           // send a text to the printer
}
interface Text_itf {
  String read();                         // read the text
  void write (String s);                 // write the text
}
interface Job_itf {
  void stop ();                          // kill the current job
}
```

These interfaces are shared between the caller and the callee. In order to make the print service available to the clients, the *Printer* application exports an instance of class *Printer* through the name server. The *Printer* class is an implementation of the *Printer_itf* interface. On its side, the client

[4] This example only aims at explaining the protection model. We don't use it as an example of JavaCard-based application. This example has the advantage to include capability parameter passing, onward and backward, and therefore to illustrate the power of JCCap.

application fetches this instance from the name server and can invoke a method (*init* or *run*) on this instance, using the *Printer_itf* interface. When the client wants to print a file, it invokes the method *run* and passes a reference to an instance of class *Text* which implements interface *Text_itf*. The *run* method returns a reference to an instance of class *Job* that implements interface *Job_itf*. The client application can invoke this instance in order to stop the *job*.

In the example, the definition of protection aims at avoiding the following protection problems:

- the printer doesn't want the client to invoke the *init* method on its printer objet (and to initialize the printer),
- the client doesn't want the printer to invoke the *write* method on its text object (and to modify the text of the client).

In our protection scheme, the client and the server will define the following views:

client

```
view client_printer implements Printer_itf {
        void init ();
        Job_itf run (reader_text text);
}
view reader_text implements Text_itf {
        String read();
        void not write (String s);
}
```

print server

```
view server_printer implements Printer_itf {
        void not init ();
        Job_itf run (Text_itf text);
}
```

Each application defines a set of views that define its protection policy. Each view « implements » the Java interface that corresponds to the type of the objects it protects. A **not** before a method name means that the method is not permitted. When an object reference is passed as parameter in a view, the programmer can specify the view to be passed with the reference, by using the view instead of the type of the parameter. If no view is specified, this means that no restriction is applied to this reference.

In this example, the print server defines the view *server* which prevents clients from invoking method *init*. No restriction is applied to the parameters of method *run*. The client defines the view *client* which says that, when a reference to a text is passed as a parameter of method *run*, the view *reader* must be passed, which prevents the print server from invoking method *write*. Notice that the client doesn't have any reason to prevent itself from invoking method *init*; this is a decision to be taken by the print server.

When the print server registers an instance of class *Printer* in the name server, it associates view *server* with it. When the client obtains this reference from the name server, it associates the view *client* with it. These two views and the nested ones (*reader*) define the access control policy of the two applications.

To sum up, each application defines its own protection policy independently from any other application or server and this policy specification is defined separately from the application implementation using views, thus enhancing modularity.

4 Implementation of JCCap

The first subsection describes the implementation of JCCap for local applications, while the second deals with remote applications.

4.1 Local applications

In this section, we present the implementation of JCCap within the JavaCard, i.e. we assume that the interacting applications are both located within the card.

For the implementation of JCCap within the JavaCard, we used the fact that Java object references are almost capabilities[5]. Indeed, since Java is a safe language [Gosling95, Sun96], it does not allow object references to be forged. This implies that if an object *O1* creates an object *O2*, object *O2* will not be accessible from other objects of the Java runtime, as long as *O1* does not explicitly export a reference to object *O2* towards other objects. The reference to *O2* can be exported (as a parameter) when an object invokes *O1* or when *O1* invokes another object. Therefore, as long as an application within the JavaCard does not export a reference to one of its objects, these objects are protected against other applications that are loaded into the card. This protection relies on the JavaCard bytecode verifier [Rose98] which verifies that the bytecode which is loaded into the card conforms to the strong typing of the Java language.

Thus, Java object references can be seen as capabilities. However, they are all-or-nothing capabilities since it is not possible to restrict the set of methods that can be invoked using such a reference. In order to implement our capabilities, we implemented a mechanism inspired from the notion of Proxy [Shapiro86], which allows access rights associated with a reference to be restricted.

Our implementation relies on the management of *filters* that are inserted between the caller and the callee. For each view defined by an application, a filter class is generated (by a pre-processor) and an instance of that class is inserted to protect the application.

When a reference to an object is passed as input parameter of a method call, instead of the real object, we pass a reference to an instance of the filter class generated from the view defined by the application providing the reference.

This filter class implements all the methods declared in the interface of the view. It defines an instance variable that points to the actual object and which is used to forward the authorized method calls. If a forbidden method is invoked on an instance of a filter class, then the method raises an exception.

The reference to the filter instance, which is passed instead of the reference parameter, is inserted by the caller application. In fact, this filter instance is inserted by the filter used for the current invocation. In figure 4a, the invocation of *O2* performed by *App1* passes a reference to *O1* as parameter. The filter *F1(O2)*, which corresponds to the protection policy of *App1* for invocations of *02*, inserts filter *F1(O1)* before the parameter *O1*. Therefore, filters that are associated with reference parameters are installed by filters that are used upon method invocations.

[5] This is the case for any object-oriented strongly typed language (safe language).

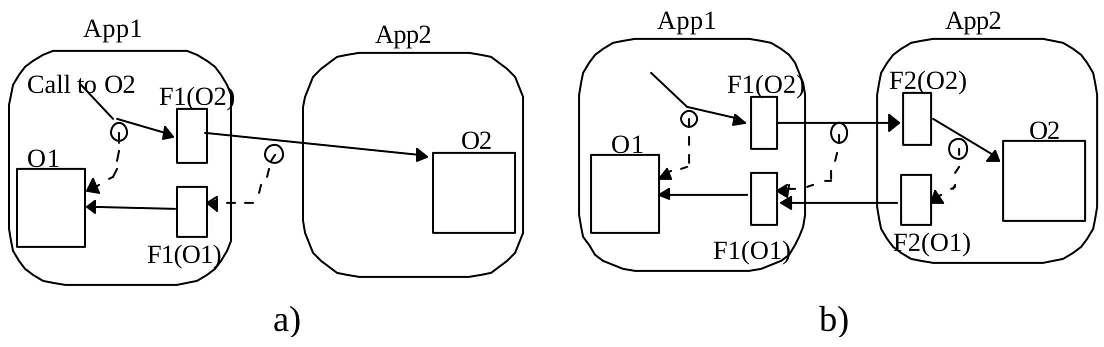

Figure 4. Management of filters.

Conversely, when a reference is received by an application, a reference to a filter instance is passed instead of the received parameter, which class is generated from the view specified by the application that receives the parameter. In figure 4b, the filter *F2(O2)*, which corresponds to the protection policy of *App2* for invocations of *02*, inserts filter *F2(O1)* before the received parameter.

Therefore, two filter objects (*F1(O1)* et *F2(O1)*) are inserted between the caller and the callee for the parameter *O1* passed from *App1* to *App2*. These two filters behave as follows:

- *F1(O1)*: it enforces that only authorized methods can be invoked by *App2* and it inserts filters on the account of *App1* for the parameters of invocations on *O1* performed by *App2*.
- *F2(O1)*: it inserts filters on the account of *App2* for the parameters of invocations on *O1* performed by *App2*.

Below is the code of the filter classes for the print server example.

client

```
public class reader_text implements Text_itf {
    Text_itf obj;
    public reader_text(Text_itf o) {
        obj = o;
    }
    public String read() {
        return obj.read();
    }
    public void write(String s) {
        Exception !!!
    }

public class client_printer implements Printer_itf {
    Printer_itf obj;
    public client_printer (Printer_itf o) {
        obj = o;
    }
    public void init() {
        obj.init();
    }
```

print server

```
public class server_printer implements Printer_itf {
        Printer_itf obj;
        public server_printer (Printer_itf o) {
            obj = o;
        }
        public void init() {
            Exception !!!
        }
        public Job_itf run(Text_itf text) {
            return obj.run(text);
        }
}
```

```
    public Job_itf run(Text_itf text) {
        reader stub = new reader_text (text);
        return obj.run(stub);
    }
}
```

The filter class *reader* of the client forwards *read* invocations to the actual instance, but it does not forward *write* invocations. Similarly, the filter class *server* of the print server only forwards invocations of the *run* method. In the filter class *client* of the client, method *run* takes as parameter a reference *text* for which a capability with the *reader* view must be passed. For this parameter, the *run* method of the filter class creates an instance of the filter class *reader* and initializes it with the actual parameter, and forwards the invocation, passing as parameter the created filter instance instead of the actual parameter.

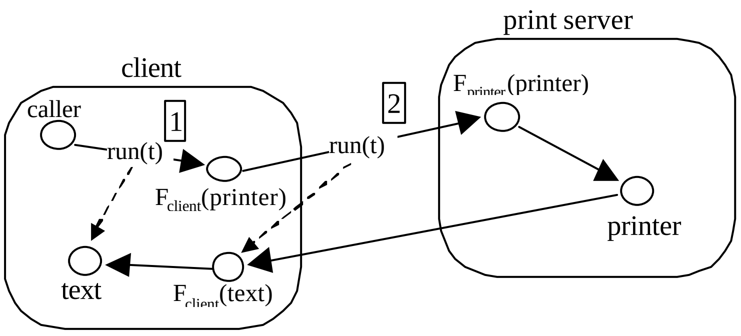

Figure 5. Filter objects management in the print server.

Figure 5 illustrates the management of filter objects in the print server example. At step 1, the invocation of the *run* method passes the reference of the actual *text* object. This invocation is performed on the filter object of the client for the reference to the *printer* object. This filter object creates a filter object for the text parameter and forwards the invocation to the filter object of the print server (step 2), passing the filter object of the text parameter. The invocation is then forwarded to the actual *printer* object. Later, the invocation of the *read* method on the *text* object goes through the filter object that was inserted by the client.

Notice that if the client had defined a view *job_view* for the reference to the instance of class *Job* returned by method *run*, the filter class *client* would have the following *run* method:

```
public Job_itf run(Text_itf text) {
  reader stub = new reader_text(text);
  return new job_view(obj.run(stub));
}
```

This method creates two filter objects, one for the (text) input parameter and one for the (job) returned parameter. In the case of the print server described earlier, view *job_view* is not necessary.

We have presented the implementation of JCCap within the Java Card. The implementation within the card relies on the safety of the Java language (enforced by the bytecode verifier) and on the

Java Card firewall that isolates the different applet contexts. Filters are implemented by the means of Java Card 2.1 shared objects that enable capabilities to be accessible from different contexts. In the next section, we describe how this implementation can be adapted in order to control access rights between remote applications, i.e. one application being within the Java Card and the other being executed on the terminal.

4.2 Remote applications

In this section, we describe the adaptations to the previously described implementation that are required in order to control access rights between remote applications (one in the card and one in the terminal).

The implementation for local applications relies on Java strong typing (implemented by the bytecode verifier when code is loaded into the card). Since each class loaded into the card is verified prior to loading, the environment guarantees that an application which uses a reference to a filter object, did not forge it and obtained it either as a parameter of an invocation or from a name server (which generally authenticates the application); this reference to a filter object acts as a capability.

When considering remote applications, we have to take into account the following characteristics [Vandewalle98]:

- First, communication between the terminal and the card implements a simplex (or master-slave) RPC paradigm. This means that invocations can only take place from the terminal to the card (and not the other way around). Therefore, regarding cooperation between remote applications, we only have to deal with capabilities stored in the terminal which reference objects within the card, and a capability transfer can only occur on return of a method invocation.
- Second, even if an RMI-like facility (such as DMI) can be used for cooperation between remote applications, communication between the terminal and the card is based on message passing, messages being structured as APDU (Application Protocol Data Units). A remote invocation facility, which implements remote references (from the terminal to the card) must relies on APDU. Thus, if a card is inserted in a malicious terminal, an application in that terminal may send the APDU that would have been sent by the remote invocation facility (in other words, the application in the terminal forged a remote reference to an object in the card). This implies that the implementation of JCCap for remote applications cannot rely on the safety of the Java language as for local applications.

We have implemented a DMI facility which integrates the management of JCCap capabilities. The basic different with the implementation described in section 4.1 is that the Java reference between the two filter objects become a remote reference (implemented with APDU message passing).

In order to protect capabilities against forgery, we have to use secrets or passwords, which allow capabilities to be authenticated by the card when used for method invocation. The management of such passwords may rely on cryptographic techniques when transferred over an untrusted communication path; we only address here the issue of authenticating capabilities which are used from the terminal in which the card is inserted.

Whenever a capability is exported from the card to the terminal, a password is generated by the card and stored into the filter object on the card side. The password is transferred to the reader along with the returned reference and stored in the filter object on the terminal side. In the terminal, when the capability is used for object invocation, the password in the filter object is transferred along with the APDU which implements the remote invocation. In the card, the filter object which receives the invocation verifies that the password received in the APDU corresponds with the password stored in the filter.

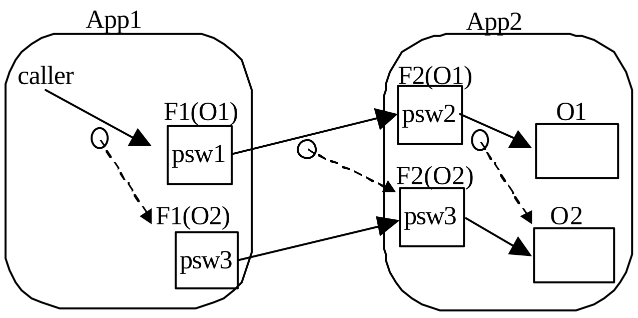

Figure 6. Filter objects management between remote applications.

In Figure 6, *App1* is an application in the reader machine and *App2* is an application in the card. *App1* invokes a method on object *O1* from *App2*. *Psw1* is included in the message (APDU) sent by *F1(O1)* to *F2(O1)*. *F2(O1)* verifies that psw1 is equal to *psw2* and that the invoked method is authorized in the view associated with this capability. If so, the invocation is performed.

On return, we assume that a capability on object *O2* (from *App2* within the card) is returned to *App1*. When *F2(O2)* is created, a password (*psw3*) is generated and stored in its state. An APDU is then returned to the terminal, including all information required to initialize *F1(O2)* (which acts as an RMI stub). *Psw3* is included in this APDU and stored in *F1(O2)* in the terminal.

With this implementation, capabilities on objects stored within the card may be acquired by the terminal, and reused later when the card in reinserted in the same terminal. Moreover, a malicious terminal in which the card is inserted cannot arbitrarily invoke an object in the card, since it is practically not possible to guess sparse passwords associated with capabilities.

To sum up, we have presented the implementation of JCCap. The definition of the protection policy of an application is based on the expression of view that are used to generate object filters that are inserted between the caller and the callee. Each application defines its own protection policy independently from any other application. Capabilities protection relies on Java's strong typing for capabilities within the card and on passwords stored in filters for capabilities in the terminal.

The current prototype is composed of a view processor that generates filter classes from views, and few runtime system classes (such as the name server). This prototype has been validated on JavaCard 2.1 [Sun98a, Sun99].

5 Application Example

The examples we used in this paper (bank, print server) to illustrate our protection model only aimed at explaining the design and implementation of JCCap.

In order to validate our protection model, we implemented a scenario where several mutually suspicious applications cooperate.

This scenario is about a frequent flyer program. We will use the example of FrequencePlus, the frequent flyer program of AirFrance. This program allows clients to cumulate *miles* as they travel flying AirFrance. Later, these miles may then be used to benefit from free tickets on AirFrance flights. However, there are many other partners (than AirFrance) in the FrequencePlus program, notably many other airlines or car rental companies (renting a car earns you miles). In the scenario we implemented, we assume that FrequencePlus only involves two partners: AirFrance (airline) and Hertz (car rental).

A JavaCard is used in order to host applications which represent each company. In our case, there are three applets in the JavaCard, one which represents the FrequencePlus program, one which represents AirFrance and one which represents Hertz.

The Hertz applet exports to the terminal an interface which allows a car rental:

- to be booked,
- to be cancelled,
- to be closed (when the car is returned),
- to be read (to read the characteristics of this rental, such as the category of the rented vehicle).

```
interface IHertz {
        public int book (int category, int duration);
        public void cancel (rentalId);
        public void close (int rentalId, int kilometers);
        public IRental getInfo (int rentalId);
}
```

```
interface IRental {
        public int readDuration ();
        public void writeDuration (int duration);
        public int readKilometers ();
        public void writeKilometers ();
        public int readCategory ();
        public void writeCategory (int category);
}
```

IHertz is the interface exported by the Hertz applet to the terminal. A car rental is here identified by an integer (a *rentalId*) return by the *book* method. When the rental terminates, the *close* method is invoked, which records the amount of kilometers performed with the car. All the information about the car rental are recorded in a record object which implements the *IRental* interface.

Notice that access rights to these methods vary according to the entity which invokes them.

The AirFrance applet exports to the terminal an interface which allows a reservation:

- to be booked,
- to be paid,
- to be cancelled.

```
interface IAirFrance {
        public int book (int class, int flightNumber, int date);
        public void cancel (int reservationNumber);
        public void pay (int numReservation, boolean payWithMiles);
}
```

IAirFrance is the interface exported by the AirFrance applet to the terminal. Notice that it is here possible to pay for a flight ticket using FrequencePlus miles.

The FrequencePlus applet exports to the terminal an interface which allows miles to be:

- to be directly credited. Some miles may be credited as a gift from FrequencePlus to a client.
- to be credited as a consequence of a car rental or a flight on AirFrance.
- to be used for paying an AirFrance flight ticket.

```
interface IFrequencePlus {
        public int read ();
        public void creditMiles (int miles);
        public void creditCarRental (IRental rental);
        public void payWithMiles (int miles);
}
```

IFrequencePlus is the interface exported by the FrequencePlus applet to the terminal. The *creditCarRental* method allows the Hertz applet to pass a reference to a rental record (which implements the *IRental* interface) in order to credit miles as a consequence of a car rental.

Notice that, within the card, Hertz has to cooperate with FrequencePlus for miles to be credited when a car rental is validated (using the *creditCarRental* method). The same cooperation would take place between AirFrance and FrequencePlus when a client flew with AirFrance. Finally, a cooperation involves AirFrance and Frequence when a client buy an AirFrance flight ticket and pay with FrequencePlus miles (using the *payWithMiles* method)

This is illustrated on Figure 7.

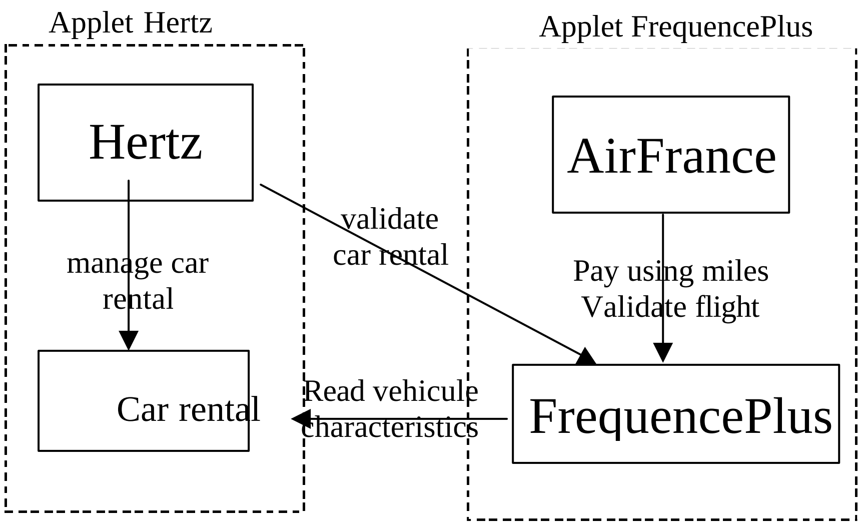

Figure 7. The FrequencePlus applications scenario.

In this example, while the AirFrance and FrequencePlus applets belong to the same company AirFrance (but could not), there is mutual suspicion between Hertz and FrequencePlus. Hertz does not want to grant FrequencePlus full access to its internal data. Hertz want to grant (only) read access to the data required by FrequencePlus to credit the miles for its client.

Therefore, using JCCap, Hertz defines the following views:

```
view VfrequencePlus implements IFrequencePlus {
      public int read ();
      public void creditMiles (int miles);
      public void creditCarRental (VRental rental);
      public void payWithMiles (int miles);
}
```

```
view VRental implements IRental {
    public int readDuration ();
    public void not writeDuration (int duration);
    public int readKilometers ();
    public void not writeKilometers ();
    public int readCategory ();
    public void not writeCategory (int category);
}
```

Conversely, FrequencePlus does not want to grant Hertz access to the “direct credit” operation (neither to the read method which returns the client’s miles total). Hertz must only be granted access to the operation that credits miles as a consequence of a car rental.

Therefore, using JCCap, FrequencePlus defines the following view:

```
view VfrequencePlus  implements IFrequencePlus {
        public int not read ();
        public void not creditMiles (int miles);
        public void creditCarRental (IRental rental);
        public void not payWithMiles (int miles);
}
```

This application has been implemented on top of JavaCard 2.1. Our facility integrates both the DMI facility and the filtering of access rights (capabilities). Therefore, after the (centralized) development of the application code, on the terminal side and on the card side, we just had specify the previous view definitions and to use our stub generator in order to enable (controlled) cooperation between the terminal applications and the applications within the card.

This experiment demonstrates the adequacy of this facility, since the application programmer only has to focus his/her interest on the problem the application aims at solving.

6 Conclusion and Perspectives

In this paper, we presented a protection model, which allows the definition of access control policies for Java Card-based applications.

Access control is defined at the level of the application interface, thus enhancing modularity and making this definition easier and clearer. The model is based on software capabilities and allows access rights to be dynamically exchanged between mutually suspicious applications. In this model, each application defines its protection policy independently from any other application. This policy is enforced dynamically during execution.

Our protection scheme has been prototyped on the Java Card 2.1 environment and experiments with simple applications revealed the advantages of this approach.

This work is the result of a cooperation with the Gemplus company. It resulted in an international patent.

Bibliography

[DiGiorgio99] R. Di Giorgio, M. Montgomery, "Write OpenCard services for downloading Java Card apps", JavaWorld 4; 2, February 1999.
URL : http://www.javasoft.com/javaworld/jw-02-1999/jw-02-javadev.html

[Goldstein97] T. Goldstein, "The gateway security model in the Java electronic commerce framework, Proc. of Financial Cryptography'97, pp. 340-354, Springer, 1997.
URL: http://www.javasoft.com/products/commerce/docs/whitepaper/security/JCC_gateway.html

[Gosling95] J. Gosling and H. McGilton, "The Java Language Environment: a White Paper", Sun Microsystems Inc., 1995.
URL: http://java.sun.com/whitePaper/java-whitepaper-1.html

[Hagimont96] D. Hagimont, J. Mossière, X. Rousset , and Pina and F. Saunier, "Hidden Software Capabilities", *Sixteenth International Conference on Distributed Computing Systems (ICDCS)*, May 1996.

[Levy84] H. M. Levy, "Capability-Based Computer Systems", Digital Press, 1984.

[Richardson92] J. Richardson, P. Schawrz, and L. -F. Cabrera, " CACL: Efficient Fine-Grained Protection for Objects", Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), ACM SIGPLAN Notices 27.10, pp. 263-275, 1992.

[Rose98] E. Rose, "Towards Secure Bytecode Verification on a Java Card", Master's thesis, Univeristy of Copenhagen, September 1998.
URL: http://www.ens-lyon.fr/~evarose/speciale.ps.gz

[Shapiro86] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle", Proc. of the 6th International Conference on Distributed Computing Systems, pp. 198-204, 1986.

[Siddalingaiah97] M. Siddalingaiah, "The Java Card ", The developer.com Journal, October 1997.
URL: http://www.developer.com/journal/techfocus/n_tech_javacard.html

[Sun96] Sun Microsystems, "JDK 1.1 Documentation", Sun Microsystems,
URL: http://www.javasoft.com/products/jdk/1.1/docs/index.html

[Sun98a] Sun Microsystems Inc., "Java Card Applet Developer's Guide", July 1998.
URL: http://java.sun.com/products/javacard/JCADG.html

[Sun98b] Sun Microsystems Inc., “Java Remote Method Invocation – Distributed Computing for Java”, May 1998. URL: http://java.sun.com/marketing/collateral/javarmi.html

[Sun99] Sun Microsystems Inc., “Java Card 2.1 Virtual Machine, Runtime Environment, and Application Programming Interface Specifications”, February 1999.
URL: http://java.sun.com/products/javacard/

[Tanenbaum86] A. S. Tanenbaum, S. J. Mullender, and R. V. Renesse, “Using sparse capabilities in a distributed operating system”, Proc. og the Sixth IEEE International Conference on Distributed Computing Systems, pp. 558-563, 1986.

[Vandewalle98] J.-J. Vandewalle, E. Vétillard, “Developing Smart Card-Based Application using Java Card “, 3rd Smart Card Research and Advanced Applications Conference, September 1998.