Extending the Data Storage Capabilities of a Java-based Smartcard

Clemens H. Cap, Nico Maibaum[1], Lars Heyden
Chair for Information and Communication Services[2]
University of Rostock, Germany
{cap, maibaum, heyden}@informatik.uni-rostock.de

Abstract

Present limitations on data memory for Java based smartcards are a serious restriction for application development. The paper presents a method for enhancing the data memory using virtual memory. Transparency for the off-card application is achieved by the facade design pattern. Security constraints of smartcard applications have implications for the overall architecture and influence the proposed design. The architecture is used in the EU project FASME for storing a larger number of XML-based administrative documents on a JavaCard for electronic government purposes.

1 Introduction

With more than one billion copies, smartcards are an important device of todays information society. The development of the JavaCard standard made this device even more popular: Capable of processing a subset of the platform independent, object oriented, and widely used programming language Java, the JavaCard puts smartcard technology at the disposal of many programmers and significantly shortens the time to market for smartcard applications [5].

Unfortunately, the small memory of a JavaCard is a serious restriction for application development. Today, JavaCards can be expected to have $1 - 4$ kByte of RAM (for the runtime stack, temporary variables and transient objects) $16 - 32$ kByte of ROM (for the card operating system, the Java virtual machine and preinstalled applications) and $8 - 32$ kByte of (flash) EEPROM (for the application code of the cardlets, presistent objects and class variables). It is the EEPROM limitation which most seriously restricts developers.

Recent JavaCard projects clearly demonstrated this limitation: [4] implemented the German national homebanking protocol HBCI (**H**ome**b**anking **C**omputer **I**nterface, http://www.hbci.de). Despite many space saving techniques, the implementation results in a CAP-file of 8.5 kByte and leaves only 2480 Bytes for data. As a result, only two bank access codes can be stored on the card instead of five as required by the HBCI specification. [6] describes problems encountered when moving the SET protocol to a JavaCard. [7] develops techniques for splitting a secure application protocol into an on-card and an off-card part, in order to reduce the memory requirements of the on-card components. In the FASME project (**F**acilitating **A**dministrative **S**ervices for **M**obile **E**uropeans, http://www.fasme.org) [8] personal and administrative data, documents and profile information of a citizen is stored on a JavaCard to facilitate administrative processes, such as registering at a new place of living. In the DISTINCT project (**D**eployment and **I**ntegration of **S**martcard **T**echnology and **I**nformation **N**etworks for **C**ross-Sector **T**elematics, http://www.distinct.org.uk) user preferences on a smartcard adapt the user interface and the collection of services offered. All mentioned projects are troubled by the limited on-card data memory [8], [1].

An often mentioned advantage of the JavaCard is the possibility to download new applications (so called cardlets) dynamically, as they are required by the card owner. Furthermore, JavaCards support multiapplication scenarios. With the above examples of memory troubles even for single applications the practical use of multiapplication settings may be seriously doubted, especially with available memory sizes lagging behind

[1]Supported by the EU Fifth Framework Project FASME, IST-1999-10882, http://www.fasme.org
[2]Supported by the Heinz-Nixdorf Foundation.
Classification (CR 1998): C.3, C.5.3, D.4.4, H.3
Keywords: Smartcard, Virtual Memory, Memory Management, Information Storage and Retrieval, Electronic Government.

manufacturer roadmaps and white papers announcements.

What are – besides waiting for larger cards – the *present possibilities* of dealing with this limitation?

It is always possible to *store the identity* of the card owner on the card and to transmit it to the card terminal, which then uses a database to associate this identity with the required application parameters. This approach has obvious security and data privacy flaws for most scenarios in which the card terminal cannot be fully trusted. In typical smartcard applications the card stores and transmits certain rights, profiles or specific attributes of the owner but should not disclose his full identity. The transmitted information often is of a confidential nature, which the user does not want to hand over to a central database. European mentality favours privacy protection mechanisms (and laws) where personal data is under the tight control of that person. It mandates the possibility of the card owner to view and fully understand every bit on a smartcard she is using.

Space saving coding techniques make a more efficient use of the limited memory, but at the price of a very twisted program structure, which is untypical for Java and object oriented development [4]. *Application splitting techniques* move parts of the code to the (untrusted) card terminal [6]. However, the splitting of a secure application protocol may endanger its security properties. Furthermore, these techniques slightly reduce the card-resident code portion and do not help in case of large applications.

In this paper we present the *smartcard extension (SCE)* as a new concept for enlarging the memory of a JavaCard for application data. The basic idea is well known and uses a networked virtual memory. However, the security and privacy needs of smartcard applications require additional cryptographic techniques. Furthermore, the smartcard extension should be transparent to the card terminal application. JavaCards with small memory using the SCE should be replacable by future JavaCards with larger memory without a need for changes to the application. We discuss a number of architectural models of the SCE, their consequences for application design, migration paths for existing applications and the security and privacy issues involved. Finally, we describe an implementation of the SCE and its use within the FASME project.

2 The Basic Concept

The basic idea of the smartcard extension is to store application data not on the smartcard itself but on an *external (virtual memory) server*. This requires a network connection to a memory server and two software components: A *communication service* to this memory server and a *memory manager*, ie. an instance deciding which data accesses are mapped to the card and which shall be delegated to the server.

The *communication service* resides on the card terminal. In most applications the card terminal consists of a card reader which is connected to a microcontroller (eg. in a point-of-sales device) or to a PC. In both cases a connection to a card service provider by a telephone line or to the Internet or Intranet is available. The communication service can thus provide a secure connection to the memory server, which stores the data in encrypted, signed, or in clear text form, depending on security requirements. If the terminal device running the communication service cannot be trusted, then the JavaCard can be used to encrypt and sign the data, which then is transmitted and stored only in encrypted form so long as it is outside of the (trusted) JavaCard.

In traditional operating system contexts, a *memory manager* translates addresses of a virtual memory space into addresses of a physical memory space. If the virtual address of a certain data object cannot be mapped to a physical address, then this data object must be loaded from a suitable backing store into a physical address (possibly overwriting this physical address and thus destroying any previously established mapping). The virtual memory address then is translated to this new physical address.

By its design principles, Java does not allow a direct manipulation of memory locations, and access to objects is never by addresses but by object references. Implementing a "traditional" memory manager would therefore require the superficial construction of an address and contents based memory model, for example on top of a Java byte array. We chose the more natural approach of the object oriented data access by calling suitable *access methods* on those objects whose instance variables we want to change. A data access thus always consists in calling a method on an object. The object itself is known through an object reference which is valid within the virtual machine of the JavaCard and the code of this object must be resident within the virtual machine. However, from the point of view of those application components using the object, its instance variables could be stored on the card itself as well as on the memory server. In both cases they shall be accessed using access methods only.

Up to now, the concept seems quite straight forward. Difficulties arise, however, due to the specific execution modes of a smartcard and from the aim to make this memory extension transparent for most components of the entire system.

3 Design Options for the Extension

Our concept still allows a wide range of design options. This section describes the interconnection of a basic memory manager with smartcard middleware layers and introduces a tag-length-value storage concept.

3.1 Smartcard Middleware Architecture

Smartcards use a special mode of communication with the outside world [3]: A card "reader" or "terminal" sends specially formatted byte codes, so called *application protocol data units (APDUs)* to the card. Upon reception of an APDU, the card activates a registered method, which upon termination must provide a return value in special APDU format. To allow the application programmer a more service oriented view of the card, formatting of application data into APDU schemata, communication with the card reader and card management usually are delegated to smartcard middleware such as OCF[3] [2] and PC/SC[4] .

OCF is the industry standard for smartcard access in a Java environment. It is an extensible collection of classes for the card terminal or the connected PC, supporting the development of JavaCard applications. The terminal part of the application uses the convenient, object structured interface of a `CardService` object. The smartcard part, independent of the use of OCF, employs the usual APDU-communication. The application programmer can be further relieved from OCF details by packaging OCF-specific startup code into a `SmartCardProxy`. Fig. 1 illustrates this architecture.

3.2 Basic Memory Manager

Suppose, a `CardService` object offers a `void setBalance (float x)` and a `float getBalance ()` service for a cash card, each sending a suitable APDU to the smartcard. Upon reception of the APDU, the card executes an access method which writes or reads the balance variable, and sends back a response APDU. Control then returns to the application.

For the smartcard extension, the `CardService` is modified. The decision to delegate a data access to the external memory server can be made on the terminal or on the card.

If the decision is made on the terminal, the `CardService` has to know which values are stored on-card and which are stored remotely. It asks the card

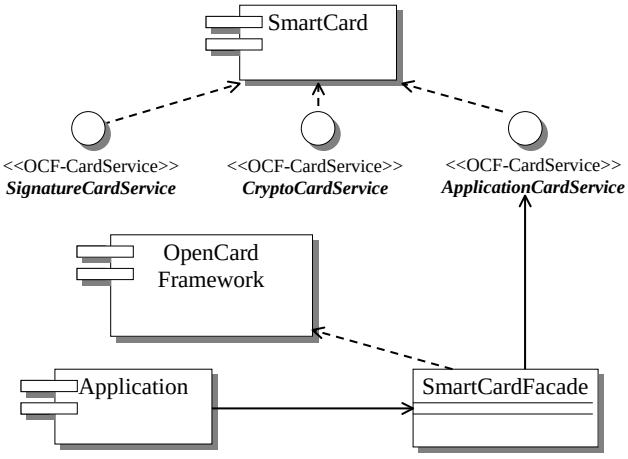

Figure 1. OCF-based Interface.

for an access-id and then performs the requested operation on the value stored under this id on the external memory server. This design, however, requires total trust into the terminal: A bogus application or `CardService` could perform incorrect operations on the value on the server, the operation no longer being under control of the smartcard. On the other hand, this approach allows an extension of the smartcard without modifying the card resident code at all (provided the `CardService` is aware of the access-id of the card, which can be chosen to be the card-id, which the card sends as response to the ATR (answer-to-reset) command executed upon insertion of the card into the reader).

In a more secure design, this decision is made by the smartcard itself. This also allows the cardlet to disallow write and read accesses depending on the application context, and to make security and plausibility checks of the values the application requests to be written. In this design, the `CardService` sends a suitable APDU to the smartcard. The card then makes all internal checks according to the semantics of the application and, if required, generates appropriate error return codes. If all checks are passed, the card however does not read or write the respective instance variables on card but returns a response APDU to the `CardService`, directing it to initiate a specific operation on the memory server. In this design, although the decision is made on the smartcard, the `CardService` could still send a bogus request to the memory server. This can be easily prevented if the card provides the request with a digital signature. If the terminal should not know the values which, possibly after some intermediary processing, is written to the card (or rather: to its extension), the request can also be encrypted

[3] **OCF:** Open Card Framework. http://www.opencard.org
[4] **PC/SC:** Personal Computer / Smartcard. http://www.pcscworkgroup.com

by the JavaCard. In addition to an extension of the `CardService`, this approach also requires a (small) modification of the smartcard access functions.

Fig. 2 illustrates the structure of the extension architecture.

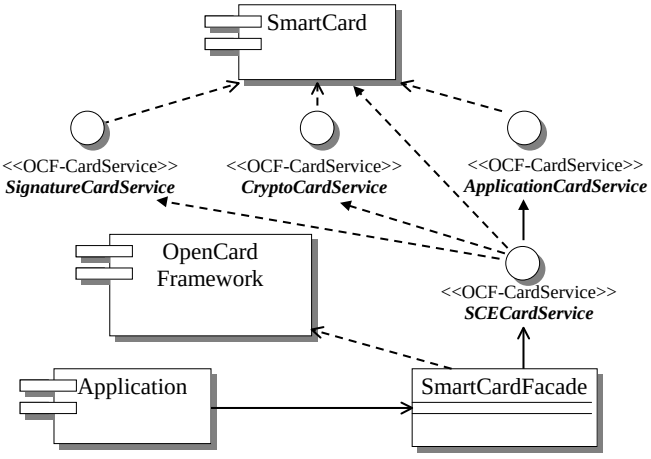

Figure 2. Smartcard Extension.

The enterprise Java beans concept and a remote method invocation (RMI) strategy using the secure socket layer (SSL) for data transport complete the secure interaction of the individual components within our architecture. The decision to use features of the Java enterprise edition provides numerous configuration advantages, but could, however, pose those problems of scalability and robustness which can be associated with new architectural concepts not yet tested under field conditions. Fig. 3 shows the overall smartcard application, consisting of terminal, JavaCard and external memory server, deployed within a Java 2 enterprise edition environment.

In all discussed approaches, the smartcard extension is fully transparent to the off-card application.

3.3 Tag-Length-Value Data Storage

The above approach is inflexible since the respective individual access methods decide statically about on-card or off-card storage of data. We therefore introduce a tag-length-value storage service to our concept [3]. This service, in the form of an on-card filesystem, is common to smartcard technology. It stores values, identified by their tag and characterized by their length. On a JavaCard, a typical implementation uses a linear byte array of a length which is fixed at cardlet compile time. To prevent overlap and to guarantee efficient memory use, the length of the stored data objects must be provided as well.

This storage concept allows the smartcard to dynamically try to store as many data elements as possible on the card itself before accessing the external memory server. This concept adapts to varying memory sizes of different card models and to the specific memory situation in multiapplication JavaCards. Furthermore this approach can accomodate a compactifying garbage collector on the storage file and store data objects of varying length. Since the JavaCard standard does not require a garbage collector most cardlets are extremly conservative on runtime object creation and do not rely on garbage collection. Therefore, and to save on code memory, the tag-length-value store should be restricted to its most basic functionality.

4 Implementation and Evaluation

To study the effects of our architecture on memory consumption, we implemented a personal id smartcard in various versions.

Version 0 and 1 are the traditional JavaCard implementations with all data stored on the card. Version 0 stores the data in instance variables, version 1 uses a tag-length-value (TLV) file. In version 2 data is stored on the card using the set and get methods to access the TLV file, but should the card run out of TLV space, the external memory server is used. In this case the card uses a generic signature and encryption method to sign and encrypt the requests sent to the memory server. The single generic signature and encryption method, although protected against access from outside the card, could be used to fool the card into signing a sensitive data element contrary to its application semantics. For example, an attacker could present the card the text of a contract using a `setName` APDU. To prevent this, we can further design type- or APDU-specific signature methods to include information on the access method with which the request to the server was signed. This information can be used for audit purposes or for semantic checks in the external memory server. Version 3 is a straight forward encoding of this concept.

Fig. 4 shows the memory consumption for implementations of these versions, grouped by the functionality of the required on-card classes. The values reported are lengths of the method bytecodes as extracted from the smartcards cap file after compilation of the Java codes using IBM Visual Age for Java, version 1.2. It demonstrates the small overhead incurred by the extension architecture.

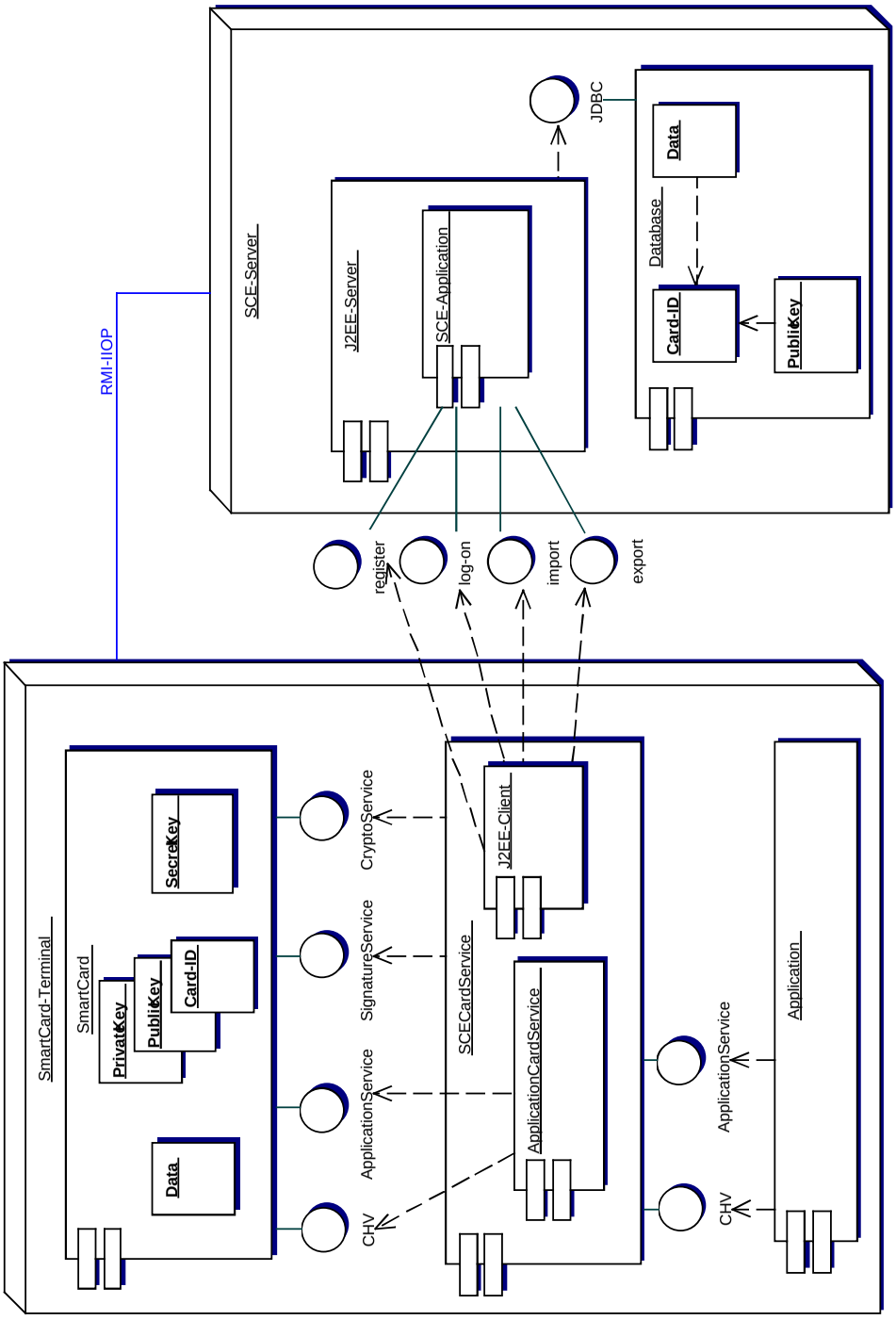

Figure 3. Smartcard Extension Deployed in Java 2 Enterprise Edition Environment.

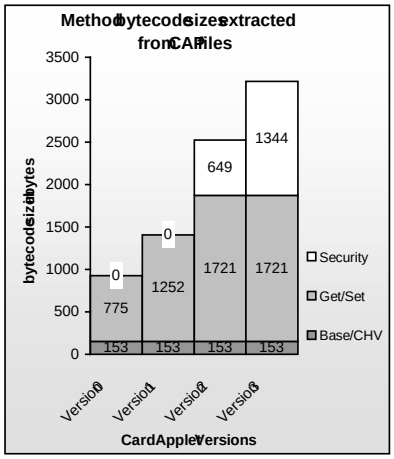

Figure 4. Overheads of the Extension.

5 Conclusion and Future Work

The on-card memory limitations for application data restrict many smartcard projects. The *smartcard extension*, presented in our paper, can be a solution for such difficulties. Depending on the capacity of the smartcard, the *card itself* is able to decide which data are stored on-card and which shall be stored remotely. In addition to that, the tag-length-value data storage enables an *automatic adaption* to the varying memory size of different cards.

Presently this *architecture is used* in the EU project FASME to store administrative documents in the form of XML-files on the Europe-wide citizen mobility JavaCard. **F**acilitating **A**dministrative **S**ervices for **M**obile **E**uropeans (FASME) maps the administrative processes required for mobile European citizens (eg. registering a car or a new place of living) to an electronic infrastructure. Administrative and personal data, digital documents, profile information and digital signatures are stored on a JavaCard. The prototype, presently under development, will be tested in three selected cities (City of Cologne [Germany], City of Grosseto [Italy] and City of Newcastle [United Kingdom]). Memory restrictions prevent the storage of all required digital documents on a citizen mobility JavaCard. In order to solve the problem the presented smartcard extension technology is used.

In this paper we concentrated on data accesses originating from the card terminal, which is the usual case in smartcard applications. Future research will deal with data accesses originating from the card itself and with *functions*, too complex to be executed on the card. These situations are more complex since off-card data access triggered by on-card functions as well as remote (off-card) function invocation disrupts the control flow between card and terminal. Fortunately it is less common on todays smartcard applications.

The presented architecture offers further advantages for *advanced application scenarios*: By moving all on-card data (with the exception of the cards private key) into the off-card storage area, the application data is available for *backup in case of loss of the smartcard*. Cryptographical techniques like key-recovery or key-escrow can guarantee the decryption of encrypted data for use by a freshly issued replacement card with a fresh private key [9]. In *multiapplication scenarios* several cardlets can share a common tag-length-value store service and thus share items like the name or address of the cardholder. Furthermore, a more efficient and dynamic allocation of data memory to multiple applications becomes possible by providing a larger portion of TLV-space to those cardlets which are often executed by the card owner. These advanced applications of the card extension are presently studied in detail.

References

[1] DISTINCT Project Consortium. *Technical Deliverable*. http://www.uninfo.polito.it/distinct/, 1999.

[2] OpenCard Consortium. *OpenCard Framework Programmer's Guide*. http://www.opencard.org, 1999.

[3] Uwe Hansmann, Martin S. Nicklous, Thomas Schäck, and Frank Seliger. *SmartCard Application Development Using Java*. Springer, 2000.

[4] Tilo Kienitz. RSA Chipkarte für Hbci, Implementierung auf einer Javacard. Master's thesis, Universität Rostock, 2000.

[5] Matthias Kaiserswerth and Joachim Posegga. Java auf Chipkarten – Das aktuelle Schlagwort. *Informatik Spektrum*, 21(1):27ff, 1998.

[6] Michail Ljubich. SET für Java. In *Proceedings of the Info98, Potsdam*, 1998.

[7] Michail Ljubich. *Working Title: Splitting JavaCard Application Protocols*. PhD thesis, University of Rostock, 2000.

[8] Nico Maibaum and Clemens H. Cap. Javacards as ubiquitous, mobile and multiservice cards. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques PACT2000, Philadelphia, USA*, 2000.

[9] Bruce Schneier. *Applied Cryptography*. John Wiley, 1995.