A PERFORMANCE COMPARISON OF JAVA CARDS FOR MICROPAYMENT IMPLEMENTATION

Jordi Castellà-Roca, Josep Domingo-Ferrer, Jordi Herrera-Joancomartí and Jordi Planes
Universitat Rovira i Virgili
Dept. of Computer Engineering and Mathematics
Autovia de Salou s/n
E-43006 Tarragona, Catalonia, Spain
{jcaste,jdomingo,jherrera,jplanes}@etse.urv.es

Abstract With the development of pay-per-view and multimedia streaming services, there is an increasing demand for practical micropayment mechanisms. Smart cards are essential to implement micropayments on the buyer's side. This paper presents a comparative analysis of four commercial Java Cards for micropayment implementation.

Keywords: Smart cards, micropayments, Java Card

1. INTRODUCTION

In the last decade, electronic commerce has probably grown faster than any other form of commerce. Growth forecasts are enormous and so are economic opportunities. Public key cryptography [5] has been key to this growth. Indeed, public-key algorithms like RSA [18] overcome the problem of key sharing and offer realizations of the digital signature concept. Digital signatures allow proofs of authenticity to be produced electronically, which is key for implementing the cornerstone of e-commerce: electronic payment schemes.

The computational cost of using public-key cryptography and digital signatures for e-payment is not justified if the amount being paid is very small (micropayments). Micropayments are electronic payments of low value and they are called to playing a major role in the expansion of electronic commerce: example applications are phone call payment, access to non-free web pages, pay-per-view TV, any kind of multimedia streaming, etc. The reason for designing specific micropayment schemes is that, when standard e-payment systems (like CyberCash [4], e-cash [9]

, *i*KP [2], SET [22]) are used for low-value payments, they suffer from too high cryptographic transaction costs as compared to the amount of payments. However, micropayments do not need as much security as speed and simplicity (in terms of computing). For that reason, several micropayment proposals try to replace the use of digital signatures with faster operations, usually hash functions. Two main hash structures have been proposed to implement hash-based micropayments (see Appendix A for details):

- Hash chains, which allow a sequence of fixed-value and fixed-currency micropayments to be made.
- Spending programs, which allow complex micropayment patterns (several values and currencies and variable sequencing).

1.1. OBJECTIVE AND PLAN OF THIS PAPER

This paper presents a comparative analysis of four commercial Java Cards in view of using them to implement the buyer's side of micropayments based on hash functions and spending programs.

Section 2 contains some considerations on Java Card programming versus standard Java programming. The performance criteria for the comparison are enumerated in Section 3. Sections 4 and 5 report on the results of the comparison; speed and storage performance have been considered jointly in Section 4 and the programming environments have been compared in Section 5. Section 6 contains some conclusions. Appendix A surveys hash-based micropayments and a generalization of them called spending programs. Appendix B describes a card file system which allows effective implementation of spending programs and hash chains on any of the considered Java Cards.

2. CONSIDERATIONS ON JAVA CARD PROGRAMMING

From the discussion in Appendix A, smart cards used to implement the buyer's side of hash-based micropayment schemes (including spending programs) should implement at least a digital signature algorithm and a hash algorithm. Java Cards seem a better choice than classical smart cards and this for two main reasons: easy programming (which results in shorter development times) and smoother compatibility with Web-based e-commerce.

A Java Card is the implementation of an interpreter for a subset of the Java programming language on a standard smart card controller. This enables card applications to be written in a modern high-level language.

The runtime environment provided by Java Cards is optimized for smart cards, with the goal of bringing many of the benefits of Java software programming to the resource-constrained world of smart cards. The following features of standard Java are not supported in the Java Card language:

- Variable types like long, float or double.
- Multidimensional arrays.
- Unicode characters.
- Threads (because the card does not support multitasking). No threads means that synchronyzed or volatile methods used to control access to shared variables are not supported.
- Garbage collection.

Other differences with standard Java regard the implementation of the Java Virtual Machine (JVM). In a Java Card, the JVM is considered as part of the operating system and is placed in read-only memory by the card manufacturer. This can be a problem to achieve smart card upgradability and true multi-functionality, as was pointed in [15]. Furthermore, due to processing and memory constraints, the JVM converter (whose mission is to translate bytecode into compressed code to be loaded on the card) is located outside the card (on the host). Only the other component of the JVM, the Java Card run-time environment (JCRE,[25]), is stored on the card.

From the above considerations, some recommendations for Java Card programmers can be stated. Since RAM is mainly used by the Java stack to hold local variables and arguments of methods, method invocations, complex expressions and local variables and arguments should be used sparingly; otherwise the RAM memory becomes full and the system resorts to EEPROM, which is considerably slower. Another issue is that the lack of garbage collection favors the depletion of EEPROM, where objects are stored. A good idea is to minimize garbage by using static or final methods and attributes whenever possible (which increases visibility) and to avoid referencing them only from local variables (which would make it impossible to reference a method after the local variables disappear).

3. PERFORMANCE CRITERIA

In order to carry out a performance comparison between several Java Cards, the following criteria have been taken into account:

Speed: Measured as the time interval (in milliseconds) elapsed between command submission to the card by the host and receipt of card response by the host.

Storage: Capacity of the card to hold cardlets (Java Card applications) in its EEPROM.

Programming environment: Development environment supplied by the manufacturer for card programming. A good development environment is crucial since most of smart card application development is done on a standard computer and applets are loaded on the card only at the final stage for testing. Therefore, aspects like smart card simulation or graphical interface tools become very important.

4. SPEED AND STORAGE

To implement hash-based micropayments, a smart card is needed which offers enough speed and enough storage to accomodate spending programs (or hash chains) as well as a cardlet. Four Java Cards have been compared, namely GemXpresso (Gemplus,[10]), Odyssey (Bull,[3]), SmartCafé (Giesecke & Devrient,[11]) and Cyberflex (Schlumberger [21]). Due to the lack of commonly accepted benchmarks for smart cards (not to speak of Java Cards), synthetic benchmarks have been used. A synthetic benchmark is designed to measure the performance of individual components in a computer system.

Table 1 lists the main hardware features of the four Java Cards being considered. Units are as follows: EEPROM and ROM are in kbytes, RAM is in bytes, clock speed in MHz. GemXpresso EEPROM is physically divided into a 10kB general purpose memory and a 5kB stack; for Odyssey, SmartCafé and Cyberflex, the value between parentheses is the amount of free EEPROM once the system is loaded. The "Bits" column indicates the word length of the card processor. It can be seen that Odyssey is the card with the fastest clock, which will result in a shorter response time (see below).

Java Card	Company	EEPROM	RAM	ROM	Clock	Bits
GemXpresso	Gemplus	10+5	512	8	3-5	32
Odyssey	Bull	8 (7)	N/A	10	10	8
SmartCafé	G&D	16 (12)	1280	32	1-5-7.5	16
Cyberflex	Schlumberger	16 (13.5)	N/A	N/A	N/A	16

Table 1 Java Card hardware features

Table 2 compares the considered Java Cards from the price and the software standpoint:

- Price refers to approximate retail per card prices.
- The cardlet size column refers to the size of the bytecode required to encode the cardlet of the synthetic benchmark described below; a shorter bytecode suggests a more efficient use of available storage, which in turn allows more spending programs and cardlets to be kept on the card. Both SmartCafé and Cyberflex offer native cryptographic functions. Cyberflex uses more storage than SmartCafé to keep its cryptolibrary. On the other hand, Cyberflex has another storage penalty because cardlets must be instantiated before they can be executed. This is far from transparent: the programmer must compute the size of the additional storage required for instantiation using the information supplied by the verifier/converter. Wrong computation of the instantiation storage size may result in a run-time failure. The benchmark cardlet used requires 1900 bytes of instantiation storage, which should be added to the cardlet size in Table 2 below.
- The JC API column lists the Java Card Application Program Interface version [24]; a newer version means that more classes and methods are available, which results in improved bytecode generation.

Java Card	price (euro)	cardlet size	JC API
GemXpresso	32.81	4539	2.0
Odyssey	23.44	4445	2.1
SmartCafé	7.50 (*)	2174	2.1
Cyberflex	21.02	4088	2.0

Table 2 Java Card commercial price and software features. (*) is the price for a 1000-unit bulk sale

A synthetic benchmark should consider the four kinds of Application Program Data Unit (APDU) defined in the ISO 7816-4 standard (see [13]):

1 No incoming data and no outgoing data.

2 Outgoing data, but no incoming data.

3 Incoming data, but no outgoing data.

4 Incoming and outgoing data.

In practice, our benchmark tests response times for all APDU types above, except the fourth type, which was not tested due to transport protocol limitations that did not allow data to be simultaneously sent and received. Specifically, the following commands are accepted by the benchmark: reset to test the first kind of APDU, put to test the second kind, and get and getinput to test the third kind.

Our synthetic benchmark consists of two parts:

Cardlet This is a Java Card application which acts as a server by implementing the four commands described above. The cardlet installation process and the size of the corresponding bytecode differ for each card, as reflected in Table 2.

Host application This is a client application written in C which runs on a Linux host and allows to send APDUs to the cardlet to test its four commands and measure the corresponding response time. In order to measure response times in the same conditions for each considered card, the host application is the same for all cards, which is possible thanks to the PC/SC standard interface [16] between host and cards.

4.1. RESET

The main purpose of this command is to be able to mesure how long does it take for the card to answer a command. This is the simplest benchmark test, as it only implies sending one empty APDU and waiting for its acknowledgment. From Figure 1, it can be seen that Odyssey is the fastest card, followed by GemXpresso, SmartCafé and Cyberflex.

There is an explanation for SmartCafé and Cyberflex being slower: since several cardlets can be on a card, the host application must send an APDU to the Java Card run-time environment (JCRE) to select a particular cardlet; in SmartCafé and Cyberflex the JCRE forwards this selection APDU to the selected cardlet, which causes additional overhead. Cyberflex's overhead is highest, since it supports two types of APDUs (standard class 00 and legacy class F0). Sending the selection APDU to the cardlet is a requirement of some electronic purse protocols and is thus an additional feature of the cards.

4.2. PUT

This command is used by the host application to download spending programs onto the smart card. This actually amounts to sending a byte string to the card. Thus, put causes the host application to send an APDU conveying an N-byte string to the Java Card, which copies the

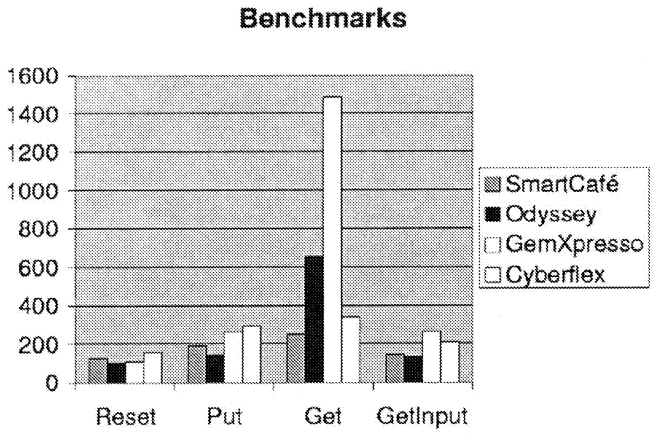

Figure 1 Average response time over 30 runs for each card and command (milliseconds).

received string to an internal buffer. In Figure 1 the response time to a put command with $N = 20$ bytes is given. It must be pointed out that the maximal value allowed for N depends on the Java Card considered and varies from $N = 112$ to $N = 128$, although smaller N are recommended to avoid timeouts. Thus, taking into account that hash values or spending program traces are typically 16 bytes long (the output length of MD5) or 20 bytes long (the output of SHA-1), it follows that a spending program cannot be downloaded as a block; a better solution is to use a put command for each hash value, so taking $N = 20$ is realistic.

Odyssey is the fastest card for the put command, followed by SmartCafé, GemXpresso and Cyberflex. It is somewhat surprising that the card with the most advanced processor ranks third, but possible explanations are that the benchmark does not take advantage of the 32-bit architecture and also that GemXpresso uses the JC API 2.0 specification. The two cards using JC API 2.1 are faster than the two using JC API 2.0.

4.3. GET

This command causes the Java Card to compute a hash function and return the answer to the host application. Since using spending programs requires the smart card to compute hash values, the time needed to compute a hash is a sensible thing to measure. The Secure Hash Algorithm

(SHA-1, [23]) has been adapted to fit into a Java Card. Adaptation can be done smoothly for Odyssey and GemXpresso, but the lack of the integer type in SmartCafé and Cyberflex is a shortcoming.

Implementing the integer type for SmartCafé and Cyberflex in Java results in a prohibitive response time. Fortunately, the latter cards are shipped with a basic cryptolibrary including DES, triple DES, RSA and also SHA-1. It is not surprising therefore that the native SHA-1 in SmartCafé and Cyberflex is much faster than the Java SHA-1 implemented on the other two cards (see Figure 1). Among native SHA-1, SmartCafé runs faster than Cyberflex. Among Java SHA-1, Odyssey is much faster than GemXpresso.

4.4. GETINPUT

In normal hash chain or spending program operation, hash chains or spending programs may be coded by the card or be downloaded onto the latter by the host application, but in any case they are stored by the card, who releases hash values or traces upon request by the cardholder willing to make a micropayment. The getinput command causes the Java Card to return a stored hash value, which amounts to returning a byte string. In Figure 1, it can be seen that Odyssey offers the fastest getinput, followed by SmartCafé, Cyberflex and GemXpresso.

5. PROGRAMMING ENVIRONMENT

In this subsection, the programming environments of GemXpresso, Odyssey, Cyberflex and SmartCafé are compared. The normal steps in developing a JavaCard application will be considered in turn: Java coding, code verification, bytecode conversion, bytecode download, cardlet test and host application development.

5.1. JAVA CODING

Documentation and help tools are fundamental components of a programmer-friendly development environment:

GemXpresso This card is shipped with a detailed manual [10] explaining the principles for developing Java Card applications and giving several examples to illustrate all functionalities offered by the card. The help tool is well organized into several complexity levels: from simple hints for the beginner to code optimization suggestions. Moreover, GemXpresso is the only card among those considered which offers a wizard to help the developer in creating a Java Card application. The wizard is based on the Direct Method Invocation

(DMI) developed by Gemplus, which relieves the programmer from handling APDUs by allowing her to specify a communication interface (a group of methods) between the host application and the cardlet; APDU communication is left to the DMI. The idea behind the DMI is similar to the principles of the Java Remote Method Invocation (RMI, [12]) and CORBA.

Cyberflex The documentation [21] in the development kit for this card is very comprehensive. The necessary steps to develop a cardlet are conveniently illustrated by verifying, converting, downloading, instantiating and testing one of the examples provided with the development kit. The developer's manual also includes detailed explanations on the Cyberflex API and error and exception codes. Like GemXpresso, Cyberflex documentation also contains code optimization suggestions. The only significant inconvenience is computation of instantiation storage by the programmer (see Section 4 above).

SmartCafé The manual for this card [11] is less complete and assumes some degree of familiarity with Java Cards. Still, the documentation contains a rich variety of examples.

Odyssey The Odyssey manual [3] is the briefest one and contains only basic information, with a few examples. It is a useful document for experienced programmers.

5.2. CODE VERIFICATION

For a Java applet to become a cardlet, a verification stage is needed to make sure that the applet complies with the requirements of the Java Card Application Programming Interface (JC API). The verification program (verifier) is supplied by the card manufacturer and obviously depends on the JC API version (2.0 or 2.1). The distracting trifle is that the verifier is also manufacturer-dependent, because it also checks card-specific restrictions. For example, there is no guarantee that a cardlet having passed SmartCafé verification for JC API 2.1 will pass Odyssey verification for JC API 2.1 (and conversely). This seriously hampers cardlet portability. It would be nice if verification was divided into two distinct substages: a first one to check compliance with JC API 2.x, and a second one to check card-specific constraints. It would even be nicer if the second substage could be eliminated.

The Odyssey verifier is the one returning the most information to the programmer, who can see the various steps of verification. GemXpresso and SmartCafé do not offer this feature, but require less parameters to be

supplied by the programmer. The Cyberflex verifier supplies information that is useful for cardlet instantiation, as commented in Section 4 above.

5.3. BYTECODE CONVERSION

The converter converts verified code to Java bytecode. Such code is compacted to save Java Card storage and may also be signed and encrypted for security purposes. Like for the verifier, there are differences between the converters provided by each manufacturer, which is a hindrance to bytecode portability.

As can be seen from Table 2, the Odyssey converter yields a shorter bytecode for the benchmark cardlet than the GemXpresso converter. SmartCafé and Cyberflex yield shorter bytecodes because the get command was implemented using the native SHA-1 function. However, it is noteworthy that Cyberflex uses almost twice as much storage than SmartCafé; a possible reason is that SmartCafé is a more evolved card.

5.4. BYTECODE DOWNLOADING

Once the cardlet bytecode has been obtained, it is ready to be downloaded to the Java Card or to the card simulator provided by the manufacturer (see discussion on test below). GemXpresso and SmartCafé implement the concept of workspace, which integrates verification, conversion and downloading into a single application. The workspace tool provided by SmartCafé is especially convenient, as it detects Java code modifications and automatically triggers verification, conversion and downloading. Odyssey offers separate applications for each functionality, which is slightly more inconvenient for developers. Cyberflex developer's environment integrates tools for verification, conversion, downloading and testing. Cyberflex has some added values:

- It is the only environment allowing convenient management of files, keys and certificates stored on card.

- It allows cardlets downloaded to the card to be easily signed.

5.5. TEST

There are basically three ways to test a JavaCard application:

Direct test: It is supported by all considered cards and it consists of sending APDUs to the Java Card and checking the correctness of the received responses; debugging erroneous responses can be quite arduous.

Simulator: If a simulator is available, it can be used to test the cardlet for correctness prior to downloading it onto the Java Card. Testing a cardlet on a simulator is faster and safer than direct test, but does not fully guarantee that the cardlet will run on the Java Card (the simulator runs on the host, which has more storage, stack and processing power than a card). Only GemXpresso and SmartCafé provide a simulator.

Simulator & Debugger: Simulator test with debugger is only possible with SmartCafé, whose debugger allows to really see what happens when a particular APDU is sent. With the other three cards, a cardlet must be debugged using a standard Java debugger, which is less accurate.

5.6. HOST APPLICATION DEVELOPMENT

To build an operational system, a host application must be developed which is able to interact as a client with the cardlet stored on the Java Card. Only GemXpresso supplies a client generator that automatically generates the client side and even allows the developer to choose in which language the client code is to be generated (C, C++ and Java are supported). In addition, GemXpresso provides information and examples to help the developer who prefers not to use the client generator. SmartCafé, Cyberflex and Odyssey do not offer a client generator, and the latter does not even provide examples nor information to help the developer. Several Java, Visual Basic and C++ examples of client applications are included in SmartCafé and Cyberflex documentations.

6. CONCLUSION

From the speed comparison of Subsection 4, it can be seen that Odyssey is the fastest Java Card in most cases, and GemXpresso tends to be the slowest one. Regarding storage, SmartCafé offers the shortest bytecode. From the developer's point of view (Subsection 5), Odyssey is outperformed by the other three cards. In addition, the following features should not be overlooked:

Context restoration GemXpresso is the only card among those considered which is able to resume execution of the last command when restoring an aborted connection.

Digital signatures For spending programs to be accepted by merchants, at least their initial trace (hash value) must be signed by the Java Card on behalf of the cardholder. Implementing in Java a fifth benchmark command RSAsignature on the considered Java Cards

was far from straightforward because of lack of storage. Fortunately, SmartCafé and Cyberflex offer a native code cryptolibrary including digital signatures (RSA). The average time required to sign a hash value is 1421 ms for SmartCafé and 1431 ms for Cyberflex.

File and certificate management Cyberflex is the only card offering an on-card system for file and certificate management. A card file system is definitely needed if several spending programs (see Appendix A) are to be kept on the card. Appendix B contains an implementation of a Java Card file system which allows several spending programs to be stored on SmartCafé, GemXpresso and Odyssey.

None of the considered cards outperforms the rest for all criteria. However, availability of cryptography in native code is an important requirement if Java Cards are to be used for micropayment implementation. Among the two cards offering native cryptolibraries, SmartCafé yields the shortest bytecode and provides the best programming environment; the lack of file system for spending program implementation is a real disadvantage in front of Cyberflex, but it can be overcome by the system proposed in Appendix B.

Appendix A. Previous work on hash-based micropayments

Quite a number of micropayment systems can be found in the literature that use hash functions instead of digital signatures to achieve higher speed. On a typical workstation, it may take half a second to compute an RSA [18] signature; in that period, 100 RSA signatures can be verified (assuming a small public exponent) and, more important, 10000 hash functions can be computed. So the advantages of dropping digital signatures in favor of hash functions should be clear.

Micropayment systems based on hash functions include NetCard [1], μ-*i*KP [14] and PayWord [19]. The principle behind those systems is similar. Let F be a computationally secure one-way hash function (i.e. easy to compute and hard to invert). Now the buyer takes a value X that will be the root of the chain and computes the sequence $T_n, T_{n-1}, \cdots, T_0$,

where

$$\begin{aligned} T_0 &= F(T_1) \\ T_1 &= F(T_2) \\ &\vdots \\ T_{n-1} &= F(T_n) \\ T_n &= X \end{aligned} \tag{1}$$

The values $T_1, \cdots, T_n$ are called coupons and will be used by the buyer to perform n micropayments to the same merchant. Each coupon has the same fixed value v. Before the first micropayment, the buyer sends T_0 to the merchant together with the value v in an authenticated manner. The micropayments are thereafter made by successively revealing $T_1, \cdots, T_n$ to the merchant, who can check the validity of T_i by just verifying that $F(T_i) = T_{i-1}$.

We next mention some differences between the main micropayment systems based on hash functions.

NetCard and μ-iKP are both micropayment schemes bootstrapped with normal e-payment systems, SET and iKP, respectively:

- With NetCard the bank supplies the root X of the hash chain to the buyer. The buyer then computes the chain, signs its last element T_0, the total number of elements n and the value of each chain element v. These signed values are sent by the buyer to the merchant, who uses the SET protocol to obtain on-line authorization for the whole chain.
- With μ-iKP, the root of the chain is a random value chosen by the buyer and the payment structure is the same as in the iKP payment system. In other words, the on-line authorization of the chain is performed by authorizing a single iKP payment of regular amount.

PayWord is a credit-based scheme that needs a broker. The buyer establishes an account with the broker who gives her a certificate that contains the buyer identity, the broker identity, the public key of the buyer, an expiration date and some other information. The hash chain is produced by the buyer using a random root. When the buyer wants to make a purchase, she sends to the merchant a commitment to a chain. The commitment includes the merchant's identity, the broker certificate, the last element of the chain, the current date, the length of the chain and some other information. In this scheme, the broker certificate certifies that the broker will redeem any payment that the buyer makes before

the expiration date, and the buyer commitment authorizes the broker to pay the merchant. Notice that in this scheme the chain is related to a pair buyer/merchant through the commitment. After that, each micropayment is made by revealing each element of the chain to the merchant.

Pedersen [17] also iterates a hash function with a random root to obtain a chain of coins but he does not provide much detail on what kind of system (credit or debit based) he implements and he does not give information about some other security issues.

The authors of μ-*i*KP emphasize that the use of hash chains implicitly assumes that micropayments take place repeatedly from the same buyer to the same merchant. Such stability assumption on buyer-merchant relationship can be relaxed at the cost of trusting an intermediate broker who maintains stable relationships with several buyers and several merchants: a buyer can send coupons to the broker and the broker is trusted to relay (his own) coupons to the merchant for the same value.

A weak point of the micropayment systems described so far is the low flexibility they offer, which results in at least two shortcomings: first, coupons have a fixed value, and second, fixed-value coupons do not allow to deal with different currencies. To overcome such lack of flexibility, a new concept called spending program was presented in [8]. Its essentials are recalled below.

Definition 1 (Spending program) A spending program $i_1, \cdots, i_n$ is a program whose instructions i_k are either value instructions, flow-control instructions, input-output instructions or assignment instructions.

Definition 2 (Value instruction) A value instruction is one that carries a specific sum of money in a currency specified in the same instruction. When a value instruction of a spending program is retrieved by the merchant, the corresponding sum of money is spent by the buyer.

Definition 3 (Flow-control instruction) A flow-control instruction allows to modify the flow of a spending program. Four types of flow-control instructions are used:

1 Forward unconditional branch

2 Forward conditional branch

3 Backward unconditional branch

4 Backward conditional branch

If i_k is a branch to instruction i_j, "forward" means that $k < j$ and "backward" that $k \geq j$. Backward branches allow instruction blocks to be executed more than once.

Input-output and assignment instructions are analogous to machine language instructions of the same type.

Clearly, a spending program generalizes the hash chain concept implemented by equations (1), since the value instructions are in fact coupons of arbitrary value. An essential issue is to find a way to encode spending programs such that value instructions in an encoded spending program are as unforgeable as coupons in a hash-chain. Structural coding [6, 7, 8] provides a secure solution. For more details on spending programs, refer to [8]. Next follows an example.

Example 6.1 The structural coding corresponding to a sequential block of n value instructions with values $v_1, v_2, \cdots, v_n$ is next given (sequential block means that no flow-control instructions are present). Let V_i be the amount v_i with some redundancy (in fact the currency name can be used as redundancy). Let F be a one-way hash function specified above and let $\oplus$ denote bitwise exclusive OR. Then the buyer computes the trace sequence $T_n, T_{n-1}, \cdots, T_1, T_0$ as follows:

$$\begin{aligned} T_0 &= F(T_1) \oplus V_1 \\ T_1 &= F(T_2) \oplus V_2 \\ &\vdots \\ T_{n-1} &= F(T_n) \oplus V_n \\ T_n &= F(V_n) \end{aligned} \tag{2}$$

After the above computation, the buyer signs T_0 to get t_0. Before the first micropayment, the buyer sends t_0 and the signed value $\alpha_{t_0} = \sum_{i=1}^{n} v_i$ to the merchant for authorization. Micropayments of values $v_1, v_2, \cdots, v_n$ can be then made by successively revealing $T_1, \cdots, T_n$ to the merchant. For instance, when the merchant gets T_1, then it can retrieve v_1 by computing $F(T_1) \oplus T_0 = V_1$. The same check is performed by the merchant's bank before crediting V_1 to the seller's account. $\diamond$

Appendix B. A Java Card file system

From Appendix A it is apparent that, if a Java Card is to implement the buyer's functionality in micropayments, then it must be able to store and manage hash chains or spending programs. In normal micropayment operation, hash chains or spending programs must be kept inside the Java Card and can be deleted once they are used up. Unlike hash chains, each spending program implements a different spending pattern (possibly with different currencies, coupon amounts, sequencing, etc.) and it should be possible to store several programs at the same time, which raises the need

for an on-card file system. However, Cyberflex is the only of the four Java Cards included in the comparison of this paper which implements a file system. We describe below the main implementation guidelines for building a generic Java Card file system allowing dynamical allocation and release of variable-sized memory blocks.

The main requirements for this file system are next stated:

A. Storage reusability. A shortcoming with Java Cards is storage reusability. Even if the memory blocks allocated for a class instance are released, they cannot be reallocated to any new instance. Thus, creating and destroying successive instances eventually uses up all available storage. A Java Card file system must find a way to sort out this problem.

B. Fast data access. To make up for the smart card limited processing capacity, data access should be as fast as possible. Data structures must be designed to meet this goal.

C. Optimized storage. Due to the limited storage capacity of a smart card, the Java Card file system must be designed to save as much storage as possible. Therefore, shorter data types should be preferred, and vectors and structures should be kept as small as possible.

To meet the requirements above, a Java Card file system with three main components (see Figure 2) has been developed:

- The pool of blocks, which is the collection of memory blocks where data are actually stored.
- The array of free blocks, which keeps track of the status of memory blocks (free or used).
- The file descriptors, which (like Unix i-nodes) keep track of which blocks are allocated to a given file. A specific feature of spending programs is that they can be stored as a collection of memory blocks of equal size (one program line corresponds to a block). Thus, for groups of contiguous blocks, the file descriptor corresponding to a spending program needs to store only the address of the initial block and the number of blocks used (see Figure 2). This simplification matches requirement C listed above (optimized storage).

The Java interface of the proposed Java Card file system is described next. To start using the system, the user must create an instance of the FileSystem object. This is the only object instance that is created during

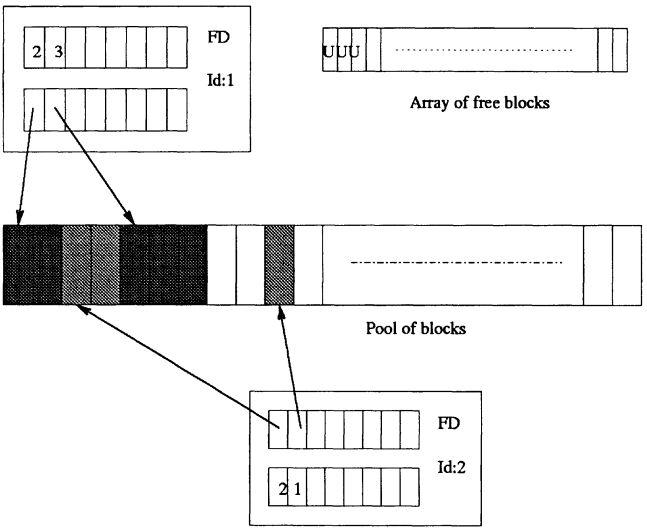

Figure 2 File system structure (FD stands for file descriptor)

the file system lifetime: to sort out storage reusability problems (requirement A listed above), file creation and deletion will be done within the FileSystem object instance and will be managed by the methods of this object. Five methods have been defined to manage files:

create() returns byte Create a new file within FileSystem and return the file identifier.

open(fileId : byte) Open a file for reading or writing. Before a file is first read from or written to, it must be opened.

read(fileId : byte, data : array of byte) Read an array of bytes from a specified file. The array must exist before calling this method.

write(fileId : byte, data : array of byte) Write an array of bytes to a specified file. The file is assumed to exist before this method is called.

erase(fileId : byte) Remove a file from the FileSystem.

Example 6.2 Next follows a simple example on how to deal with files:

FileSystem fs = new FileSystem();

```
byte file1, file2;
byte data = new byte[ AnAmountOfBytes ];
byte outputData = new byte[ AnAmountOfBytes * 3 ];
file1 = fs.create();
file2 = fs.create();
fs.open( file1 );
fs.open( file2 );
fs.write( file1, data );
fs.write( file2, data );
fs.write( file1, data );
fs.erase( file2 );
fs.write( file1, data );
fs.open( file1 );
fs.read( file1, outputData );
```

The sequence of file operations above is as follows:

1 Create the FileSystem.

2 Create two files and open them.

3 Write several times into the files.

4 Erase the second file.

5 Write another time into the first file. The FileSystem will probably write the new block on the storage previously released by erasure of the second file.

6 Open the first file and read all the data stored in it.

◇

Acknowledgment

This work was partly supported by the Spanish CICYT under contract no. TEL98-0699-C02-02.

References

[1] R. Anderson, C. Manifavas and C. Sutherland. NetCard - A practical electronic cash system. Available from author: Ross.Anderson@cl.cam.ac.uk, 1995.

[2] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik and M. Waidner. *i*KP - A family of secure electronic payments protocols. In First USENIX Workshop on Electronic Systems. New York, July 1995.

[3] Bull Inc. Odyssey Reference Manual. 1998.

[4] CyberCash, http://www.cybercash.com

[5] W. Diffie and M.E. Hellman. New directions in cryptography. IEEE Trans. on Information Theory, 22(6):644-654, 1976.

[6] J. Domingo-Ferrer. Software run-time protection: a cryptographic issue. In Advances in Cryptology - Eurocrypt'90, pages 474-480, 1991. Springer-Verlag, LNCS 473.

[7] J. Domingo-Ferrer. Algorithm-sequenced access control. Computers & Security, 10(7):639-652, November 1991.

[8] J. Domingo-Ferrer and J. Herrera-Joancomartí. Spending programs: a tool for flexible micropayments. In Information Security, pages 1-13, 1999. Springer-Verlag LNCS 1729.

[9] e-cash, http://www.digicash.com

[10] Gemplus, GemXpresso Rapid Applet Development. 1998.

[11] Giesecke & Devrient GmbH. SmartCafé 1.1 Card. Reference Manual. May 1999.

[12] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Mc Graw-Hill, 1997.

[13] S. B. Guthery and T. M. Jurgensen. Smart Card Developer's Kit. Macmillian Technical Publishing, 1998.

[14] R. Hauser, M. Steiner and M. Waidner. Micro-payments based on *i*KP. IBM Research Report 2791. Also appeared at SECURICOM'96. http://www.zurich.ibm.com/Technology/Security/publications/1996/HSW96.ps.gz

[15] C. Markantonakis. The Case for a Secure Multi-Application Smart Card Operating System. First International Informaton Security Workshop ISW'97, pages 188-197, 1997. Springer-Verlag LNCS 1396.

[16] PC/SC Specification 1.0, December 1996. http://www.pcscworkgroup.com

[17] T. P. Pedersen. Electronic payments of small amounts. In Security Protocols, pages 59-68, 1997. Springer-Verlag LNCS 1189.

[18] R. L. Rivest, A. Shamir and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120-126 1978.

[19] R.L. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. Technical Report, MIT LCS, November 1995.

[20] R.L. Rivest. RFC 1321. The MD5 Message-Digest Algorithm. MIT Laboratory for Computer Science, April 1992.

[21] Schlumberger. Cyberflex Access. Developer's Series. Version 3.1. July 1999.

[22] Secure Electronic Transactions.
http://www.mastercard.com/set/set.htm

[23] Secure Hash Standard, U. S. National Institute of Standards and Technology, FIPS PUB 180-1, April 1995.
http://csrc.ncsl.nist.gov/fips/fip180-1.txt

[24] Sun Microsystems. Java Card 2.1 Application Programming Interface, February 1999.

[25] Sun Microsystems. Java Card 2.1 Runtime Environment (JCRE) Specification, February 1999.