Pieter H. Hartel
Pierre Paradinas
Jean-Jacques Quisquater (eds.)

Proceedings

1996
CARDIS

Smart Card Research and Advanced Applications

2nd International Conference CARDIS 1996
CWI, Amsterdam, The Netherlands, September 16–18, 1996

Editors

Pieter H. Hartel
University of Amsterdam
Department of Computer Science
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Pierre Paradinas
Gemplus PSI
1 Place de la Méditerranée, F-95200, Sarcelles, France

Jean-Jacques Quisquater
Université Catholique de Louvain
Department of Electrical Engineering (DICE)
Place du Levant, 3 B–1348, Louvain-la-Neuve, Belgium

Smart Card Research and Advanced Applications: second International Conference; proceedings / CARDIS 1996, Amsterdam, The Netherlands, September 18-20, 1996. Pieter H. Hartel; Pierre Paradinas; Jean-Jacques Quisquater (eds.) - Amsterdam: Stichting Mathematisch Centrum 1996

CR subject classification (1991): B.1, B.2, B.3, C.1, D.2, D.4,
E.3, F.3, H.3, J.1, K.4

ISBN 90 6196 465 2
Published by Stichting Mathematisch Centrum
Amsterdam 1996

Preface

This volume contains the papers accepted for presentation at the second international conference on *Smart Card Research and Advanced Applications* (CARDIS) held in Amsterdam, The Netherlands, September 16–18, 1996. The first CARDIS conference was held in Lille, France in November 1994.

The first three papers discuss applications of cryptology to smart cards. The first paper by Kim *et al* presents a solution to efficiently implementing a stream cipher. The second paper by Kelsey *et al* uses cryptology to certify outcomes of programs. This makes it possible to bill for software usage. The third paper by Alexandre identifies a form of keyboard based biometrics that would probably be more acceptable than most other forms of biometrics.

The next set of three papers consider architectures for smart cards. A comprehensive overview of current arithmetic co-processors is provided by Naccache *et al.* The second architecture paper describes a public key co-processor and the last paper on this theme by Dhem *et al* describes different approaches to compressing information that is to be processed by a smart cards.

The third set of three papers deal with methodological issues. Glaser *et al* discuss ways of analysing cryptographic protocols using visual rendering. In the paper by Alberda *et al* an example is given of how formal methods can be used to reason about a programming language used to construct a smart card operating system. In the paper by Hollmann *et al* an application of statistically analysing data obtained from monitoring the hardware is discussed.

The last set of three papers deals with the environment in which smart cards operate. Domingo-Ferrer discusses how a client server approach helps to securely offload compute intensive operations. Carlier *et al* take this theme further by discussing how not only the card but also the user's mobility should be taken into account. The last paper by Biget *et al* discusses how object orientation and the CORBA architecture may help to provide a distributed environment for smart cards.

On behalf of the programme committee we thank all those who submitted papers. We thank the referees for their careful work in the reviewing and selection process. The organisation of the conference is grateful to CWI for allowing the use of its conference facilities. Simone Panka, Frans Snijders (both CWI) and Jon Mountjoy (UvA) have done most of the local organisation, which is gratefully acknowledged.

Pieter H. Hartel, Amsterdam
Pierre Paradinas, Sarcelles
Jean-Jacques Quisquater, Louvain-la-Neuve
July 1996

Programme committee

Stefan Brands	CWI, Amsterdam
André Gamache	Université de Laval, Québec, Canada
Louis Guillou	CCETT, France
Josep Domingo-Ferrer	Universitat Rovira i Virgili, Tarragona, Spain
Pieter Hartel	University of Southampton, UK and University of Amsterdam, The Netherlands
Hans-Joachim Knobloch	NTG/Xlink, Kalrsruhe, Germany
Pierre Paradinas	Gemplus, France
Reinhard Posch	Graz Institute of Technology, Austria
Jean-Jacques Quisquater	Université Catholique de Louvain, Louvain-la-Neuve, Belgium
Matt Robshaw	RSA Laboratories, USA
Bruno Struif	GMD, Darmstadt, Germany
Doug Tygar	Carnegie-Mellon University, Pittsburg, USA

Referees

Marjan Alberda
Pierre Ardouin
Stefan Brands
Josep Domingo
Hervé Guibert
Pieter Hartel
Hansi Knobloch
Çetin Koç
David Naccache
Pierre Paradinas
Thomas Pornin
Karl Posch
Gilbert Pradel
Jean-Jacques Quisquater
Matt Robshaw
Dirk Scheuermann
Bruno Struif
Nadia Tawbi
Nhan Le Thanh
Doug Tygar
Jean-Jacques Vandewalle

Supporting institutions

CARDIS 1996 was organised in cooperation with the following organisations:

- IFIP Special group 16.
- International Association for Cryptologic Research (IACR).
- Association Française des Sciences et Technologies de l'Information et des Systèmes (AFCET).

Sponsors

The following companies and organisations have generously provided financial support:

- CWI (Center for Mathematics and Computer Science), Amsterdam, The Netherlands
- Gemplus SA, Sarcelles, France
- Irdeto Consultants BV, Hoofddorp, The Netherlands
- Integrity Arts Inc, San Mateo, California
- KNAW (Royal Academy of Sciences), Amsterdam, The Netherlands
- QC Technology, Zaandam, The Netherlands
- SION (Foundation for Computer Science Research), Amsterdam, The Netherlands
- WINS (Faculty of Mathematics, Computer Science, Physics and Astronomy), Univ. of Amsterdam, The Netherlands

Table of contents

On the Design of a Stream Cipher and a Hash Function Suitable to Smart Card Applications

Yongdae Kim, Sangjin Lee, Choonsik Park

Section 0710, Yusong, P.O. Box 106,
Electronics and Telecommunications Research Institute,
Taejon, 305-600, Korea,
E-mail : kyd@dingo.etri.re.kr

Abstract. In this paper, we propose a stream cipher and a hash function based on the LFSR on $GF(2^8)$ suitable to a smart card application. Since the proposed stream cipher and hash function use 8-bit operation, it is easy to implement by software in a smart card. The proposed scheme needs 512 bytes memory space, is fast, and is easy to implement. Furthermore, if the length of a LFSR on $GF(2^8)$ is 20, the size of hash algorithm is 160 bit, and the period and linear complexity of the proposed stream ciphers are about 2^{320} and 2^{160}, respectively, which provides enough security in some smart card applications.

keywords : stream cipher, hash functions, smart cards, LFSR

1 Introduction

A smart card is a portable plastic device, the size and the shape of a credit card with an embedded microcomputer chip. The typical smart card contains a small microprocessor, memory, input/output, and an operation system that manages the hardware resources under its control. All accesses to the file(=memory) of the smart card are controlled by the microprocessor.

Thanks to its control capability and large user data storage, smart cards are becoming increasingly popular as security devices. Unfortunately, not all smart cards themselves provide enough security. To provide high level security of smart card, some cryptographic techniques should be considered. Among these techniques, as we know, a block cipher, DES is widely used for providing the smart card security. However, streams ciphers instead of block ciphers can be implemented as the improvement method of the smart card security.

In Crypto'94, H. Krawczyk[3] has presented simple and efficient hash functions based on LFSRs(Linear Feedback Shift Registers) applicable to secure message authentication. Among his constructions, LFSR-based Toeplitz method is efficient and can be theoretically analyzed. However, LFSR over $GF(2)$ which inevitably needs bit-oriented operations is less efficient for smart card application, for most operations of smart card are byte-oriented.

LFSR over $GF(2)$ has been used for fast binary random sequence generator, which can be applied to data protections and error corrections in the digital communications. Unfortunately, it is not suitable for software implementation, especially for smart cards. In this paper, we construct an LFSR over $GF(2^n)$, which consists of a primitive feedback polynomial of length m over $GF(2^n)$ and produces n-ary sequences of period $2^{mn} - 1$ efficiently. By applying Krawczyk's method to LFSR over $GF(2^8)$, we can prove ϵ-balancedness which means the security of the hash function depends on that of the stream cipher. Furthermore, the hash functions using LFSR over $GF(2^8)$ can be easily implemented, and provide fast hashing speed. Using LFSRs over $GF(2^8)$, we can easily implement well-known stream ciphers, e.g., BRM[2], *etc.* The construction proposed here can be used efficiently in the existing smart cards.

In section 2, we briefly review the smart card with cryptographic applications. We review Krawczyk's hashing algorithm constructions in section 3. In section 4, LFSR over $GF(2^8)$ is proposed, and applying Krawczyk's method to $GF(2^8)$, we prove the ϵ-balancedness of the proposed scheme, and propose some stream ciphers using LFSR on $GF(2^8)$. Section 5 is our conclusion.

2 Smart Card with Cryptographic Applications

Smart cards are portable devices that can help enhance security if they are properly used. However, many security problems are encountered with smart cards and with systems that use smart cards. To improve the security of smart card and related systems, user authentication, message authentication and encryption must be considered.

User authentication for smart cards can be divided into external validation and internal validation. External validation is that cardholders authenticate themselves to the smart card and the card then validates the cardholder. Such authentication is called the user identification. Internal validation is that the smart card authenticates itself to the smart card system and the system then validates the smart card. If needed, the smart card can verify the authentication of the card system(mutual authentication). As the cryptographic techniques for internal validation, challenge/response methods along with secret key cryptosystem or public key cryptosystem, or zero-knowledge based methods are well known. Actually, among the secret key cryptosystems, DES is widely used for the smart card authentication.

Any data transferred to or from the smart card have to be protected against unauthorized modifications. Such protection of the integrity of messages sent to the smart card can be achieved by means of electronic signatures, MAC(Message Authentication Code), CRC(Cyclic Redundancy Code), *etc.* The MAC or CRC is calculated using the cryptographic hashing function or a predetermined primitive polynomial. DES is also often used as the cryptographic hashing function.

In the smart card, sensitive storage data such as passwords, personal records, *etc.*, has to be ensured that these are read by authorized persons only. To do this, all informations of the card must be encrypted using secret or public key cryptosystems.

As mentioned above, DES, one of the most famous block ciphers is widely used for providing the smart card security. However, in this paper, we propose stream ciphers suitable to the smart card application. Especially, from the practical point of views, we have developed the hashing function based on the stream cipher and constructed a hash function which can be used as a modification detection method. The construction presented here can be used efficiently in the existing smart cards.

3 Previous Results

In Crypto'85, Y. Desmedt presented an unconditionally secure authentication schemes based on one-time pad[1]. In Auscrypt'92, X. Lai et al. presented a fast cryptographic checksum algorithm using nonlinear feedback shift register but not LFSR[4].

The practical LFSR-based hashing schemes, namely LFSR-based Toeplitz method, are proposed by H. Krawczyk in Crypto'94[3]. His construction method is as follows :

The Construction : Let $p(x)$ be an irreducible polynomial over $GF(2)$ of degree n. Let $s_0, s_1, \ldots$ be the bit sequence generated by an LFSR with connections corresponding to the coefficients of $p(x)$ and initial state $s = (s_0, s_1, \ldots, s_{n-1})$. For such polynomial $p(x)$ and initial state $s \neq 0$ we associates a hash function $h_{p,s}$ such that for any message $M = (M_0 M_1 \ldots M_{m-1})$ of binary length m $h_{p,s}(M)$ is defined as the linear combination $\bigoplus_{j=0}^{m-1} M_j \cdot (s_j, s_{j+1}, \ldots, s_{j+n-1})$ where $\cdot$ is a scalar product.

Theorem 1 (Krawczyk). *Let $p(x)$ be an irreducible polynomial of degree n over $GF(2)$ and let $s = (s_0, s_1, \ldots, s_{n-1})^T$ be an initial state for the LFSR defined by the connection polynomial $p(x)$. Let M be an m-bit long message. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n(different) roots of $p(x)$ over $GF(2^n)$. Then,*

$$h_{p,s}(M) = BD_{M,p}B^{-1}s$$

where B is a non-singular matrix which depends on $p(x)$ only and $D_{M,p}$ is an $n \times n$ diagonal matrix with $M(\lambda_i)$, $1 \leq i \leq n$, as its i-th entry.

In the above theorem, $M(x)$ is a polynomial defined as follows :

$$M(x) = M_0 + M_1 x + M_2 x^2 + \ldots + M_{m-1} x^{m-1}.$$

Definition 2 (Krawczyk). A family of hash functions H is $\oplus$-linear if for all M and M' we have $h(M \oplus M') = h(M) \oplus h(M')$.

Definition 3 (Krawczyk). A family of hash functions is called $\epsilon -$ *balanced* if

$$Pr_h(h(M) = c) \leq \epsilon, \text{ for all } M \neq 0 \text{ and } c.$$

Definition 4 (Krawczyk). A hash function is ϵ-otp-secure if the probability that an adversary succeeds in breaking the authentication is larger than ϵ, in case that a sender sends $(M, h(M) \oplus r)$ where r is a random pad.

Theorem 5 (Krawczyk). *The LFSR-based Toeplitz construction defined above is ϵ-balanced for $\epsilon \leq \frac{m}{2^{n-1}}$.*

Remark. If a hash function h is $\oplus$-linear, then h is ϵ-balanced if and only if h is ϵ-otp-secure. Hence, the LFSR-based Toeplitz construction defined above is ϵ-otp-secure. That is, if we use a stream cipher instead of an one-time pad, the security of his hashing algorithm depends on that of the stream cipher.

However, since his scheme is based on LFSR over $GF(2)$, software implementation of his scheme is inefficient in a smart card. Due to the property of finite field, this construction can also be extended to $GF(2^8)$, which means it is suitable to smart card application.

4 Stream Ciphers and Hash Functions over $GF(2^8)$

4.1 LFSRs over $GF(2^n)$ Revisited

An LFSR over $GF(2^n)$ as shown in Fig. 1 of length m consists of m memory cells which form the state $(s_0, s_1, \cdots, s_{m-1})$ of the registers. The function $f(x)$ is mapping of $\{GF(2^n)\}^m$ to $GF(2^n)$.

$$f(x) = c_0 \oplus (c_1 \otimes x) \oplus (c_2 \otimes x^2) \oplus \cdots \oplus (c_{m-1} \otimes s_{m-1}) \oplus x^m$$

where $\oplus$ and $\otimes$ denote the operations of addition and multiplication, respectively, and all coefficients $c_0, c_1, \cdots, c_{m-1}$ are elements in $GF(2^n)$.

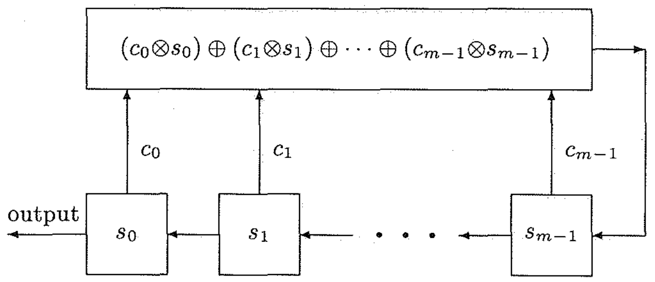

$(c_0 \otimes s_0) \oplus (c_1 \otimes s_1) \oplus \cdots \oplus (c_{m-1} \otimes s_{m-1})$
c_0
c_1
c_{m-1}
output
s_0
s_1
s_{m-1}

Fig. 1. An LFSR over $GF(2^n)$

Remark. The period of an LFSR over $GF(2^n)$ with a primitive polynomial $f(x)$ of degree m over $GF(2^n)$ is $2^{mn}-1$.

If we denote elements α and β in $GF(2^n)$ by binary blocks of length n, $\alpha = (y_1, y_2, \cdots, y_n)$ and $\beta = (z_1, z_2, \cdots, z_n)$, then the addition of two elements on $GF(2^n)$ is defined by

$$\alpha \oplus \beta = (y_1 \oplus z_1, y_2 \oplus z_2, \cdots, y_n \oplus z_n).$$

Hence the addition $\oplus$ over $GF(2^n)$ can be simply computed by the bitwise XOR of two binary blocks. However, in general, it is not easy to compute the multiplication of two elements on $GF(2^n)$. We adapt the method of multiplication introduced in [7]. In this method, any element $\alpha = (x_1, x_2, \cdots, x_n)$ is considered as an integer j such that

$$j = \sum_{i=1}^{n} x_i \times 2^{i-1} \tag{1}$$

<u>Multiplication method</u>

1. Preprocessing : Store two tables *log* and *antilog* of 2^n-1 n bits integers, which are defined as follows:

   $$log[\alpha] = i,\ antilog[i] = \alpha$$

   where $g^i = \alpha$ for all $0 \leq i \leq 2^n-2$ and g is a primitive element of $GF(2^n)$. α is considered as an integer j defined by the equation (1).

2. To compute $\alpha \otimes \beta$, we first access log table, and then calculate

   $$k = log[\alpha] + log[\beta] \bmod 2^n-1.$$

   The result is

   $$\gamma = \alpha \otimes \beta = antilog[k] \tag{2}$$

In the above multiplication method, we can easily see that the size of *log* and *antilog* table is $n(2^n-1)$ bits each.

For easy implementation, we introduce simple polynomials.

Definition 6. A polynomial over $GF(2^n)$ is *simple* provided that all of its coefficients except the constant term are either 0 or 1.

Example 1. The primitive polynomial $f(x) = x^{32} + x^{21} + x^{14} + (y^7 + y^6 + y^4 + y^3 + y + 1)$ is simple over $GF(2^8)$, where $GF(2^8) = GF(2)[y]/(y^8 + y^4 + y^3 + y^2 + 1)$.

In order to apply to smart card, $n = 8$ is suitable. Then, for implementing an LFSR over $GF(2^8)$, only 512 bytes memory is needed for multiplication. By adopting the notion of simple polynomial, we need only one multiplication(actually one table lookup) to implement LFSR over $GF(2^8)$. Due to 8-bit CPU of smart card, this implementation is more efficient than DES, which is widely used for providing smart card security.

The following algorithm generates LFSR sequence over $GF(2^n)$ with simple primitive polynomial $f(x)$ and initial vector s.

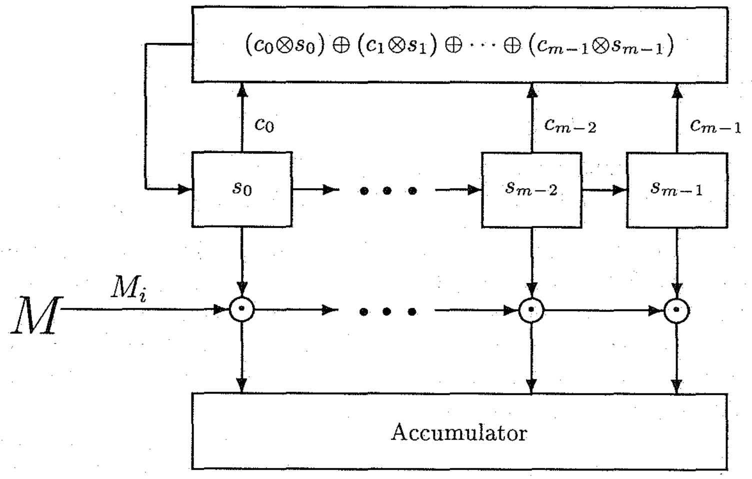
```
Algorithm 1: The LFSR over $GF(2^n)$

Input.  A simple primitive polynomial $f(x)$ of degree $m$ and two tables defined by
        the preprocessing. Let c[k] be the coefficients of $f(x)$ for all $0 \leq k \leq m-1$
Step 1. For $k = 0, \cdots, m-1$, initialize $s[k]$ by a random byte.
Step 2. Compute $\gamma = c[0] \otimes s[0]$.
Step 3. For $k = 1, \cdots, m-1$, if $c[k]$ is 1 then $t = t \oplus s[k]$.
Step 4. For $k = 1, \cdots, m-1$, set $s[k] = s[k-1]$. And then, set $s[0] = t$.
Step 5. Repeat Step 2 – Step 4 to produce sufficiently many random bytes.
```

4.2 Hash Function using LFSRs over $GF(2^8)$

In this subsection we extend Krawczyk's LFSR hashing to hash algorithm based LFSR over $GF(2^8)$ that is suitable for the smart card. The following is the proposed construction method of hash function using LFSR on $GF(2^8)$.

The Construction : Let $p(x)$ be an irreducible polynomial over $GF(2^8)$ of degree n. Let $s_0, s_1, \ldots$ be the byte sequence generated by an LFSR with connection polynomial $p(x)$ over $GF(2^8)$ and initial state $s_0, s_1, \ldots, s_{n-1}$. For each such polynomial $p(x)$ and initial state $s \neq 0$ we associate a hash function $h_{p,s}$ such that for any message $M = (M_0 M_1 \ldots M_{m-1})$ of m byte blocks $h_{p,s}(M)$ is defined as the linear combination $\bigoplus_{j=0}^{m-1} M_j \cdot (s_j, s_{j+1}, \ldots, s_{j+n-1})$ where $\cdot$ is a multiplication on $GF(2^8)$ being able to be implemented by look-up table.



$(c_0 \otimes s_0) \oplus (c_1 \otimes s_1) \oplus \cdots \oplus (c_{m-1} \otimes s_{m-1})$
c_0
c_{m-2}
c_{m-1}
s_0
s_{m-2}
s_{m-1}
M_i
M
Accumulator

Fig. 2. Hash Algorithm using LFSR over $GF(2^8)$

Theorem 7. *Let $p(x)$ be an irreducible polynomial of degree n over $GF(2^8)$ and let $s = (s_0, s_1, \ldots, s_{n-1})^T$ be an initial state for the LFSR defined by the connection polynomial $p(x)$. Let M be an m-byte long message. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n(different) roots of $p(x)$ over $GF(2^{8n})$. Then,*

$$h_{p,s}(M) = BD_{M,p}B^{-1}s$$

where B is a non-singular matrix which depends on $p(x)$ only and $D_{M,p}$ is an $n \times n$ diagonal matrix with $M(\lambda_i)$, $1 \leq i \leq n$, as its i-th entry.

Proof. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n(different) roots of $p(x)$ over $GF(2^{8n})$. Consider a diagonal matrix D whose ith diagonal entry is λ_i. Let P be an LFSR generating matrix related with the irreducible polynomial $p(x)$. Since the minimal polynomial of P is $p(x)$, there exists a non-singular matrix B such that $P = BDB^{-1}$.

In the other way, hash function $h_{p,s}(M)$ can be written as follows :

$$h_{p,s}(M) = M(P) \cdot s$$

where $M(x) = M_0 \oplus M_1 \cdot x \oplus \ldots \oplus M_{m-1} \cdot x^{m-1}$. Then,

$$\begin{aligned} M(P) &= M_0 \oplus M_1 \cdot P \oplus \ldots \oplus M_{m-1} \cdot P^{m-1} \\ &= M_0 BIB^{-1} \oplus M_1 BDB^{-1} \oplus \ldots \oplus M_{m-1} BD^{m-1}B^{-1} \\ &= B(M_0 I \oplus M_1 D \oplus \ldots \oplus M_{m-1}D^{m-1})B^{-1} \\ &= BD_{M,p}B^{-1} \end{aligned}$$

Hence, the conclusion follows. □

Theorem 8. *The proposed hash function is ϵ-balanced for $\epsilon \leq \frac{m}{2^{8(n-1)} \cdot 255}$.*

Proof. For a fixed message $M \neq 0$ and a fixed hash value c, we need to bound the probability $h_{p,s}(M) = c$ for randomly chosen irreducible polynomial $p(x)$ and initial state s.

Case I : Let $c = 0$. Since $s \neq 0$,

$$h_{p,s}(M) = 0 \Longrightarrow D_{M,p} \text{ is singular } \Longrightarrow M(\lambda_i) = 0 \Longrightarrow p(x)|M(x)$$

The probability that $p(x)|M(x)$ is at most the number of possible irreducible factors of $M(x)$ divided by the total number of irreducible polynomials of degree n over $GF(2^8)$, i.e.

$$\frac{\frac{m}{n}}{\frac{2^{8n} - 2^{8(n-1)}}{n}} = \frac{m}{2^{8(n-1)} \cdot 255}$$

Case II : Let $c \neq 0$. In order for $h_{p,s}(M) = c$, we need $D_{M,p}$ to be nonsingular and s be the unique vector that is mapped by $BD_{M,p}B^{-1}$ into c. The vector s assumes this value with probability of $\frac{1}{2^{8n}-1}$, and therefore $h_{p,s}(M) = c$ happens with at most this probability.

In either case the probability that $h_{p,s}(M) = c$ is at most $\frac{m}{2^{8(n-1)}\cdot 255}$ and then our construction is $\frac{m}{2^{8(n-1)}\cdot 255}$-balanced. □

If initial values s and irreducible polynomial $p(x)$ is known, we can find collusion by taking $M'(x) = M(x) + f(x)p(x)$ for any polynomial $f(x)$. Even though s is a secret key with a known polynomial $p(x)$, s can be easily calculated if $(M, h_{s,p}(M))$ was sent by the sender. In order to overcome such serious security faults, Krawczyk considered two methods[3] : One is to change irreducible polynomial, i.e. different hash function is used in each communication, which is not efficient for implemetation on a smart card. The other is to use a stream cipher with the hash function having secret key s and unknown irreducible polynomial $f(x)$. Hence, for a smart card application, the latter is more suitable, and the sender transmits M together with "tag" $t = h_{p,s}(M) \oplus r$ where r is generated by a stream cipher.

Remark. By theorem 7, as same as Krawczyk's scheme, the security of the proposed hash function depends on that of the stream cipher. Next subsection gives secure stream ciphers using LFSRs over $GF(2^8)$.

4.3 Some Secure Stream Ciphers over $GF(2^8)$

Most of stream ciphers using LFSR on $GF(2)$ is known to be converted to those on $GF(2^8)$[5]. In this subsection, we give an example, the BRMs over $GF(2^8)$ which is considered as a secure stream cipher.

A BRM n-ary sequence generator consists of two LFSRs on $GF(2^8)$ of length m and k, respectively. At time t, the two LFSRs are both clocked. LFSR2 is then clocked a further a_t times, where a_t is the number represented by the contents of l fixed bit positions among mn positions of LFSR1. We assume $l < mn$ so that $0 \leq a_t \leq 2^l - 1$. We define the output of the system to be the output of LFSR2 after both registers have been clocked once and LFSR2 has been clocked a further a_t times. The following theorem is similar to the main theorem in [2].

Theorem 9. *Let $M = 2^{mn}-1, K = 2^{kn}-1$, and $p = M+2^{mn-1}(2^l-1)$. Then if every prime factor of M divides K and p is relatively prime to K, the BRM on $GF(2^8)$ produces n-ary sequences of period $(2^{mn}-1)(2^{kn}-1)$ and linear complexity $k(2^{mn}-1)$.*

Remark. We usually choose $M = K$ in practice. In this case, if $\gcd(mn, l) = 1$, the n-ary sequences generated by the BRM is of period $(2^{mn}-1)^2$ and linear complexity $m(2^{mn}-1)$.

4.4 Evaluation of the Proposed Stream Cipher and Hash Function

In this subsection, we briefly evaluate the algorithm size and speed of the proposed stream cipher and LFSR hashing method. Furthermore, we roughly compare our proposed schemes with DES and MAC.

In a view of algorithm size, when we implement both by software, it is easily seen that algorithm itself is very simple, for we can use LFSR over $GF(2^8)$ for stream cipher and hashing simultaneously, that is, we only need to have a subroutine of LFSR, and source code of LFSR itself is simple also. Compared with complex DES and MAC code, implementing the proposed schemes needs much short code. In a view point of the size of look-up table, our proposed methods only including look-up table for multiplication(512 byte) have as almost same memory needed as DES and MAC including S-Boxes, and various permutations.

In a view of algorithm speed, though the implementing technique is important, it is known that stream cipher is faster than block cipher. But, in a smart card implementation the situation is different, LFSR on $GF(2)$ needs bit operation. But, the proposed schemes need only byte operation, and it is likely that it is faster than DES or the other block ciphers which have been considered as secure. In order to compare in a same situation with DES, our hypotheses are that the length of LFSR over $GF(2^8)$ is 8 which means hash values are 64-bit long, message length is 64 bit long(hence, 8 iteration is needed). We use simple polynomial for easy implementation. In this situation, we need 8 times table look-up for LFSR, and 8×8 times table look-up for hashing, while DES needs 8×16 times table look-up even when we only think of S-boxes. And, we need 8×8 times byte exclusive-or for LFSR and 8×8 times byte exclusive-or for hashing, while DES needs 16×3 times 32 bit exclusive-or for encryption and 64 times exclusive-or for hashing. Though we compare DES and the proposed scheme for common operation, it is easily seen that our proposed scheme is much faster than DES when we implement by software.

5 Concluding Remarks

In this paper, we proposed a stream cipher and a hash function based on the LFSR on $GF(2^8)$ suitable to a smart card application. Since the proposed stream cipher and hash function use 8-bit operations and small memory size, it is easy to implement the proposed one by software in a smart card. Furthermore, we can provides enough security to the proposed stream cipher by taking longer LFSR on $GF(2^8)$.

However, in a view point of the security of the proposed hash function, $p(x)$ must be secret and s be secret key. Furthermore, the proposed hash function inevitably needs a secure stream cipher for a random pad generation. This is a major weak point of the proposed hash function, but using predefined secure stream cipher, we can achive both secure hashing and encryption.

References

1. Y. Desmedt, "Unconditionally Secure Authentication Schemes and Practical and Theoretical Consequences", *Advances in Cryptology - Crypto'85*, pp. 42 – 55, Springer-Verlag, 1986.

2. W. G. Chambers and S. M. Jennings, "Linear Equivalence of Certain BRM Shift Register Sequences", *Electronics Letters*, Vol. 20, No. 24, pp. 1018 – 1019, 1984.
3. H. Krawczyk, "LFSR-based Hashing and Authentication", *Advances in Cryptology - Crypto'94*, pp. 129 – 139, Springer-Verlag, 1995.
4. X. Lai, R. A. Rueppel, and J. Woolven, "A Fast Cryptographic Checksum Algorithm Based on Stream Ciphers", *Advances in Cryptology - Auscrypt'92*, pp. 339 – 348, Springer-Verlag, 1993.
5. S. Goh, S. Lee, S. Chee, S. Park, and D. Lee, "The Extended LFSRs and their Application to the High Speed Data Protections", *Proc. of ICCC'95*, pp. 629 – 632, 1995.
6. B. Schneier, *Applied Cryptography, Second Edition*, p. 587, John Wiley & Sons, Inc., 1996.
7. G. Harper, A. Menezes and S. Vanstone, "Public-key Cryptosystems with Very Small Key Lengths", *Advances in Cryptology - Eurocrypt'92*, pp. 163 – 173, Springer-Verlag, 1993.

This article was processed using the LaTeX macro package with LLNCS style

Authenticating Outputs of Computer Software Using a Cryptographic Coprocessor*

John Kelsey Bruce Schneier

Counterpane Systems, 101 E Minnehaha Parkway, Minneapolis, MN 55419
{schneier,kelsey}@counterpane.com

Abstract. A cryptographic coprocessor is described for certifying outcomes of software programs. The system for certifying and authenticating outputs allows a third party who trusts the secure components of the system to verify that a specified program actually executed and produced a claimed output.

Keywords: authentication, digital signatures

1 Introduction

We present an application of digital signatures [Dif76]. Through a cryptographic coprocessor [GUQ92, YT95, Sch96] — here called an "Authenticator" — software can certify particular outputs. Software can use this capability to allow the Authenticator to certify that some specified outputs or outcomes of the software have actually been achieved. These protocols can be implemented on a variety of hardware designs using any of several digital signature algorithms: RSA [RSA78], ElGamal [ElG84], DSA [NIST94], etc.

Normally if a person claims that he has performed some task or achieved some result with a software program it can be difficult to verify that this actually occurred. The output itself may be displayed (printed or photographed) but this is not always reliable evidence of the process that was followed to produce that result. The Authenticator can produce a digitally signed statement which securely and reliably attests to the actual output of the program. The Authenticator has access to the internal operation of the software (as will be described in detail), and its ability to produce secure and untamperable output allows this authentication functionality.

A good example of this capability would be the certification of a high score in some game software. Perhaps the manufacturer wishes to set up a promotion where prizes are offered to people who complete a game or achieve a specified high score. Such promotions are difficult to offer at present due to the difficulty of fairly verifying claims of success. The Software Authenticator would allow

* This work was done under contract with Walker Digital, Inc.; U.S. and international patents are pending.

those players who have actually met the goals to certify their results and win the prize, while preventing cheaters from sending in doctored output or copying output from true winners and claiming it as their own.

The same system has another functional capability independent of the authentication scheme: software metering. The Authenticator can be used to check aspects of a program's functioning related to the amount of time spent or number of uses. As with an electric meter, the Authenticator records information about how much the software has been used, in a form which the end user cannot tamper with. At some regular interval (probably every month) the Authenticator is read by a remote connection to a central computing service. The central computer then bills the user for his actual usage of the software during the month just as with other utilities. Several variations on this general scheme will be described using the same overall system configuration.

2 System Configuration

There are three main hardware components to the authentication system: the Computer, the Authenticator, and the Data Source. Generally, these will be connected in one of the following two configurations:

Authenticator $\longleftrightarrow$ Computer $\longleftrightarrow$ Data Source

Computer $\longleftrightarrow$ Authenticator $\longleftrightarrow$ Data Source

The Computer is the main computing unit owned by the end user, which runs the main part of the software programs he uses. It may be a general purpose computer such as a PC, or it may be more specialized, such as a dedicated game playing unit or TV-based computerized entertainment system. It will have some input and output capabilities, typically including a video/sound display and possibly a printer for output, with input ranging from simple joystick systems to full keyboard and/or video capability. The Computer will include RAM and usually ROM memory, and may also have non-volatile memory such as a disk drive or Flash RAM. For most applications the Computer must have a modem or other network connection to allow it to communicate with the Central Computer.

The Authenticator is a small piece of hardware enclosed in a tamper-resistant box which includes memory and CPU. The Authenticator is a computer in itself, but generally a less powerful one than the main Computer in the system. It will have much less RAM and its CPU will probably be less powerful. The Authenticator must include some non-volatile memory as described above, some RAM, and some ROM which holds the program which implements its basic functionality. The non-volatile RAM of the Authenticator will hold the cryptographic keys used for communication and authentication. The Authenticator will also require a hardware random-number source to be used for initializing keys.

This paper assumes that a software program can be divided into two components–one running on the Authenticator and the other running on the computer–such that the Authenticator can determine the output of the entire software program.

The Data Source represents the place from which software programs which will be run on the metering system come. Depending on the configuration, this may be a local disk drive or CD-ROM, a game cartridge, or a remote computer connected by telephone line or computer network connection.

One system component which is not shown in the diagrams above is the centralized computer system which communicates with the Authenticator (possibly via the Computer). This server will be referred to as the Central Computer (or CC). A communications link must exist for the Authenticator to talk to the Central Computer at regular intervals in order to transmit authenticated output information. In many ways this is similar do Chaum's electronic database with an "observer" chip [Cha93].

As shown in the diagrams above, software programs flow from the Data Source into the Computer/Authenticator system. This flow is essentially one directional, although in some circumstances (such as if the Data Source is a disk drive or a computer on the Internet) data may flow in the opposite direction during program execution or at other times. (Another example of this model is a satellite-to-PC system: lots of continuous bandwidth down, and a small trickle up via a dial-up line.) But for the unique features of the metering/authentication system, the Data Source can be thought of as a read-only source of programs to execute.

In addition, the diagrams both show a two-way link between Authenticator and Computer. This link is active during most phases of the protocol, and although the amount of data to be sent across the link is normally not large, it is necessary during software execution that the link latency (the time needed to get a short message across) is small. The Computer and the Authenticator work in close cooperation during program execution and so their communication must not introduce noticeable delays. [2]

The two diagrams differ in whether the Data Source connects to the computer directly, as shown in the first diagram, or whether it connects via the intermediary of the Authenticator, as shown in the second. Example configurations for the first case might include a general-purpose computer, with the Authenticator in the form of a smart card, PCMCIA card (now called a PC Card), dongle, or add-on card which is connected to the computer via a serial or parallel port or is connected as an expansion card internally. A dedicated videogame machine, like a Sega Saturn, which had an expansion port or slot where the Authenticator could plug in would also fit this configuration.

Examples of the second configuration would include systems where the Authenticator is built into a special device to access the Data Source. For example,

[2] In the protocol descriptions and other discussion below, the Authenticator will be referred to as though it is capable of performing actions which only the Computer can do, such as accessing the Data Source in the first diagram above, or storing data in the Computer's non-volatile memory. It is understood that in such situations it is actually a cooperation between the Computer and the Authenticator which occurs, with the Computer performing these functions at the request of the Authenticator.

the Authenticator could be built into a custom CD-ROM drive which would then be able to use special CD's customized for the metering/authentication system. Alternatively, the Data Source could be a remote computer reached via the Internet or some other computer network, and the Authenticator would be built into a special modem used to access the data. Another example of this configuration would build the Authenticator into a pass-through box, like a Game Genie, which would plug into a cartridge slot on a game machine and allow cartridges to plug into the Authenticator and be the Data Source.

The functional differences between the two configurations shown are not very significant, and both will provide the same basic functionality. The choice will usually be dictated by other considerations in terms of the expansion capabilities of the system as to where the best place is to attach the Authenticator. If all else is equal, there may be some small advantage to the second configuration, where the Data Source passes through the Authenticator en route to the Computer. One option for the use of the system is for the Data Source to store the whole software program in encrypted form, both the secure part which will run on the Authenticator and the insecure part which will run on the computer. If this option is used, then passing the data through the Authenticator will allow it to be conveniently decrypted as it is loaded into the memory of the Computer. With the other configuration this is still possible, but the data would have to be transferred from the Computer to the Authenticator and back in order to be decrypted, requiring more data transfers and so taking more time to load the program into memory. As will be discussed below, the security increase provided by this option is only marginal so it would not normally be the determining factor in selecting which configuration option to use.

3 Applications

Certifying High Scores: One application of the output authentication system is high score verification in the context of a dedicated game system, or entertainment software on a general purpose computer. This could be used as part of a contest as in the earlier example, or alternatively continual high score rankings could be maintained on an Internet server system where people could gain bragging rights by seeing how they ranked against other players around the world. This could open up possibilities where some of the characteristics of online games, specifically the element of competition against other players, would be available in the context of home game systems. Action, strategy, and puzzle games could all be made more fun and exciting if successful players could demonstrate their achievements publicly.

Distributed Key Search: Another possible application of output certification could arise in the context of distributed computing. Several research groups are working on systems to harness the massive collective computational power of the Internet and apply it to hard problems. One example which has already had some success is in factoring large numbers. Another area which has been explored is exhaustive search for cryptographic keys. If these research systems could be commercialized so that people were paid for letting their computers be used for problems like these, it could make available a huge new source of computer cycles. But one problem with these ideas is the issue of verifying that each participant actually performed the computational work he had agreed to. Some kinds of applications are self-checking, such as graphics rendering, but others, such as the key search or factoring examples, may legitimately end up with no successful outputs. In those cases a user could cheat by simultaneously allocating his computers to multiple projects and report no results to all of them, collecting more pay than he is entitled to. The authenticated output system can fix this problem, by making sure that even if the output is null, that that is the legitimate result of running the software. People would be required to present authenticated output from their program runs if they expect to get paid for their compute cycles. This technology can reduce cheating and thereby bring the whole approach closer to economic feasibility.

Metering: If the authenticated output were how long an application was running or how much data an application displayed or processed, then the system can be used as a software meter. This could be used as a "pay-for-play" arcade system for home video game machines, a "pay-for-use" or "pay-for-feature" system for personal computers, or a "pay-per-page" system for database programs. The meter could be installed on the much-hyped Internet terminal, allowing users to pay only for the time they use the machine; enhancements could even allow metering of World Wide Web pages.

4 System Overview

4.1 Signature and Trust Issues

One issue relating to any form of authenticated output is what the trust relationships are between the Authenticator and those who view the output. In our system, the output is authenticated by the Authenticator. This means that those who trust the Authenticator and are in a position to verify its signatures are the ones who will be able to trust and accept the authentication. This will include most particularly the Central Computer, which shares a key with the Authenticator and which, if symmetric cryptography is used for the authentication, is the only entity (other than the Authenticator itself) which can verify the signatures.

In some applications the authenticated output needs to be verified by other parties. The CC and its affiliated organizations can verify that data is accurately certified by an Authenticator which is part of the authentication system, and then provide public key signatures on that data. This second-order certification can use a widely known and respected public key and is suitable for wide distribution and acceptance.

4.2 Communications

Unlike the software metering application, the authentication application can be designed to have very modest data transmission requirements between the Authenticator and the CC. This raises the possibility that it could be used even in an environment where no direct electronic link exists between the Authenticator and the CC, but in which the information is displayed on the screen by the Authenticator and the user manually transfers it to the CC, say by calling the CC on the telephone and entering the data on his telephone keypad. Similarly data could be returned from CC to Authenticator by alphanumeric data being provided over the phone from the CC (via voice synthesis) and entered into the Authenticator through the regular input device, and keyboard or even a simple joystick interface.

4.3 Authenticator/Computer Interface

In order for the Authenticator to be in a position to authenticate the output of the program, it must be able to know that the output actually did occur. This means that the Authenticator must be intimately involved in the calculation of the output, such that even if the part of the program running in the insecure Computer is tampered with, the Authenticator is able to know whether a given output actually occurred or not. This may require a larger fraction of the program execution to occur on the Authenticator than in the case of the metered software application.

5 Protocols

Protocols will be described for two basic cases, the first being an electronic connection between the Authenticator and the Central Computer, and the second being the simple case described above where all such communication is via the human user of the system. This second system will be referred to as the "low bandwidth" case, and probably consists of a human on the telephone, entering characters read to him over the telephone into the computer, and typing digits from the computer screen into the telephone.

1. Initialization of the Authenticator. This is used when the Authenticator is first activated, to generate keys and communicate them to the CC. For the low-bandwidth case it may be preferable to have the Authenticator's unique secret key calculated at the factory and programmed into its ROM at manufacturing time, recorded in the CC's database.
2. Adding a New Program. This protocol is used when the user has acquired a new software program which requires information from the CC in order to run. As with the metering application, some programs may be runnable without any new information from the CC but others will require keying

information to be acquired. The low-bandwidth case will require the use of programs which do not require interaction with the CC and so this protocol will not be used in that case.

3. Starting Authenticated Software. This protocol involves the Authenticator, Computer, and Data Source. It describes how these components interact at the time an authenticated software program is loaded and execution begun.
4. Authenticate Output. This is the main protocol of the system, used when a program produces some output which the user wants to have authenticated.

Each program is generally encrypted using a different key, unique to that program. There are some advantages to using a single key to encrypt a large number of programs, but there is some security risk in doing so, since that key would be more valuable than others and if it were somehow exposed the set of programs which use it would all become insecure. So it is expected that in most cases programs will be encrypted using a unique key.

In order to begin running such a program, then, it will be necessary for the metering system to acquire the key for that program. This will be done as part of the "Adding a New Program" protocol.

In some cases the convenience of being able to run a program for the first time without any interaction with the CC will be important. In that case the Authenticator must already have a key for that program. This could be handled by having all such programs be encrypted with the same key (or possibly all programs from a given manufacturer encrypted with the same key), and having the CC send that key to all Authenticators during their initialization. At a slight cost in memory the system can be made somewhat more secure by using several different keys for the programs, with the key being chosen based on the Software ID (ID_S), a unique ID associated with each program (discussed in more detail below). In this way the keys will be shared approximately equally across all such immediately-runnable programs, reducing the value of each individual key of this type. In the low-bandwidth implementation it is expected that all programs will be of this type due to the difficulty of acquiring a key for each different program without an electronic connection to the CC.

In the resulting system all programs are of one of two types. They can be immediately-runnable, and hence encrypted using one of the shared keys; or they can requiring interaction with CC before first run, in which case the program is encrypted with a unique key.

5.1 Data Structures

There are many important pieces of data associated with the Authenticator.

1. Authenticator Keys: The Authenticator requires several keys in order to perform its varied functions, including communicating securely with the Central

Computer. Keys are of three types: "secret" keys are those used with conventional cryptosystems such as DES [NBS77]; "private" and "public" keys are those used with public key cryptosystems such as RSA [RSA78].

2. Authenticator's Secret Key (SK_A): Also known to CC. This key is used for secret and authenticated communication between the CC and the Authenticator, possibly via the Computer and/or an insecure communications link. SK_A is normally generated by the Authenticator during the initialization phase, and is then transmitted to the CC in a message secured by PK_{CC}. Alternatively it may be programmed into the Authenticator's ROM at manufacturing time, and recorded into the CC's database at that time. This will be necessary for the low-bandwidth authentication application.
3. PK_{CC}: Central Computer's public key, known to Authenticator. This key is used for initial communications between the Authenticator and the CC before SK_A is created. It is burned into ROM at manufacturing time.
4. SK_{IR}: Secret key for immediately-runnable programs. As described above, it may be desirable to support a class of programs which can be run immediately upon acquisition, without running the Adding a New Program protocol. (This is especially necessary for the low-bandwidth case.) For this to be possible the Authenticator must already store the key for such a program. All such programs share a special ID_S, and the Authenticator will recognize that ID and use the SK_{IR} key to decrypt the program, as described in the Using Authenticated Software protocol. As mentioned above, a variation on this idea would define several ID_S's of the immediately-runnable class, each of which would be associated with a different SK_{IR} key.
5. ID_A: An identification number unique to each Authenticator, burned into ROM at manufacturing time
6. $Table[Software, Key]$: This table has a list of ID_S and $(Software, Key)$ pairs. Each pair contains the key which will be needed to decrypt the encrypted portions of the software with the specified ID_S.

There are also data structures associated with each piece of metered software that comes from the Data Source. Each piece of authenticatable software from the Data Source is divided into three parts. The Software Control Block has information about the software which identifies it. The executable software itself occupies the two remaining parts. Part of the software is designed to run securely on the Authenticator, while part is designed to run in the insecure environment of the Computer.

1. Software Control Block: The Software Control Block has information about the software which will be used by the metering system to run it. The SCB is signed by the private key of the CC, and the Authenticator checks the signature when the software is loaded. SCB fields include the ID_S: This is a unique number identifying this piece of meterable software. Every piece of software and every revision of a software item have unique ID_S's. As discussed in the context of the metering application, there are two general kinds of ID_S's, "program" and "component," distinguishable by their high

order bits. Program ID_S's are used to refer to programs as a whole, while Component ID_S's refer to specific features of a program. Only Program ID_S's are necessary for the authentication application.

2. Insecure Software Component: The Insecure Software Component is the bulk of the software program and runs on the Computer. It may be stored in encrypted form in the Data Source, in which case the Authenticator will be responsible for decrypting it at the time the program is loaded into memory. If this decryption step will add unacceptable delay to program loading, this insecure component can be stored unencrypted at only a slight loss of security. Since the memory of the Computer is insecure by definition, a determined attacker can gain access to the plaintext of the Insecure Software Component in any case. So the additional security added by storing it in secure form is limited in value.
3. Secure Software Component: The Secure Software Component runs on the Authenticator itself. It is stored in encrypted form in the Data Source and must be decrypted by the Authenticator as the program is loaded into memory. As will be described below, the Secure Software Component contains software which implements selected but crucial functionality on which the larger body of software in the Insecure Software Component depends. The encryption of the Secure Software Component is the primary feature by which the overall security of the system is maintained. It prevents attackers from replacing this component with software which will authenticate outputs which did not occur.

5.2 Encryption of messages

Where electronic communication is used, all messages between the CC and the Authenticator are encrypted and authenticated. Either public-key or symmetric encryption can be used, although symmetric encryption appears to provide sufficient security for most of the protocols. The encryption used is assumed to be a strong, modern cipher with key sizes in the range of 64 to 128 bits. Examples include IDEA [LMM91], or Blowfish [Sch94]. Public key encryption would most commonly use RSA [RSA78].

Once initialization is complete, CC and Authenticator share SK_A, which can be used with a conventional encryption system to provide for both encryption and authentication. Because the Authenticator has limited access to sources of entropy, and because the total volume of data to be communicated between Authenticator and CC is small, a few hundred bytes per month in typical usage, using SK_A as the key for all communications between the two systems should provide adequate security for this application. In this configuration, messages from the Authenticator are preceded by sending ID_A (a unique identifier specific to the Authenticator and burned into its ROM at manufacturing time) in the clear, allowing the CC to lookup the encryption key used, followed by the message itself encrypted with SK_A. Responses from CC are sent encrypted with SK_A.

All communications between CC and Authenticator are initiated by the Authenticator, with the CC acting as a server. As described above, it is expected that the user will actually initiate such communications rather than having the Authenticator spontaneously issue requests. Messages sent by the Authenticator will include a sequence number which will increment each time a message is sent in that direction. Reply messages from the CC will include that same sequence number. This will allow both sides to detect message replay attacks, in which messages are captured and then replayed at a later time in order to disrupt the protocols.

The packet formats shown below do not include encryption headers or the account and sequence numbers, which are included as described above except where indicated. Each packet begins with an unique identifier value describing the kind of packet it is, and is followed by data as described below.

For the low-bandwidth case, the protocols used are more modest in their communication requirements, and no implicit sequence numbers or encryption are used other than those explicitly called out.

5.3 Initialization of the Authenticator

For the low-bandwidth case, no special protocol is needed at initialization time. SK_A is programmed into ROM at manufacturing time like ID_A and PK_{CC}. This protocol is only used in the case of electronic communication between Authenticator and CC.

This is used when the Authenticator is first activated, to generate keys and communicate them to the CC. Note that at that time the Authenticator has access to PK_{CC}, the Central Computer's public key, and ID_A, its own unique ID. We assume that the Authenticator has access to a good source of random numbers via a hardware random number generator.

Unlike other protocols, these packets are not implicitly encrypted with the keys shared between CC and the Authenticator. Instead, the encryption used is explicitly identified at each step of this protocol.

1. Authenticator calculates SK_A, the random key which will be used for communication between itself and the CC.
2. Authenticator creates an Initialization Message block of the following format:

   Initialization Message
   ID_A
   Current date and time
   SK_A

3. Authenticator encrypts the Initialization Message block with SK_A, then encrypts SK_A using PK_{CC} and sends both blocks to CC.
4. CC recovers first SK_A, then the Initialization Message block. It verifies that date and time are approximately current, and records the new ID_A, checking

that it has not been used before. It remembers SK_A and associates that value with ID_A.

5. CC creates an Initialization Message Response block of the following form:

   Initialization Message Response

6. CC encrypts the Initialization Message Response block under SK_A and sends it back to the Authenticator.
7. Authenticator decrypts and verifies the IM Response block.

5.4 Adding a New Program

This protocol is used when the user has acquired one or more new software programs which require information from the CC in order to run. All messages in this protocol are sent protected by encryption and sequence numbers as described above. In the low-bandwidth case all programs are of the immediately-runnable type and so this protocol is not used in that case.

1. Authenticator reads new programs' Program Control Block(s) from Data Source, and extracts ID_S for (each) program.
2. Authenticator creates a New Program Message of the following format:

   New Program Message
   Number of programs requested
   ID_S
   ID_S
   ...

3. Authenticator securely transmits the New Program Message to the CC.
4. CC looks up the ID_S's for which keys are requested to determine the keys needed to decrypt those programs.
5. CC creates a New Program Message Response block of the following format:

   New Program Message Response
   Number of programs
   ID_S, Key
   ID_S, Key
   ID_S, Key
   ...

6. Authenticator records the key information for each software program in its $Table[Software, Key]$ structure.

5.5 Starting Authenticated Software

This protocol describes how the Authenticator, Computer, and Data Source interact at the time an authenticated software program is loaded and execution begun. As described above, the software which comes from the Data Source contains a Software Control Block and two executable components, an insecure component which executes on the Computer and a secure component which executes on the Authenticator. At least the secure component is encrypted, and the insecure component may be encrypted as well. Note that in the low-bandwidth case the software will be immediately-runnable.

1. Authenticator reads the Software Control Block from the Data Source and extracts the ID_S for the software.
2. Authenticator determines whether the required key is available to decrypt the program. The key will be found either by looking up the ID_S from the Software Control Block in the $Table[Software, Key]$, or else will be SK_{IR} for immediately-runnable programs (recognized by their ID_S). If the key is not available the Authenticator displays a message informing the user that he needs to add the new program to the current list of software and enable the communication with CC to acquire the needed keys, and the protocol terminates.
3. Authenticator and Computer then read and decrypt the Insecure and Secure Software Components from the Data Source. As noted above, the Secure Software Component will always be encrypted, and the Insecure Software Component may or may not be encrypted, depending on design tradeoffs.
4. Authenticator and Computer then transfer control to the newly read software components, Authenticator running the Secure Software Component and Computer running the Insecure Software Component.

5.6 Authenticate Output

This is the principle protocol of the system, used when a program produces some output which the user wants to have authenticated. We assume that the division of software between the secure and insecure components is such that the Authenticator can in fact determine that the specified output actually occurred. For the electronic communication case it works as follows.

1. Authenticator creates an Authenticated Output block of the following form:

   Authenticated Output
   ID_S
   Null terminated text string describing output

2. Authenticator sends the Authenticated Output block to CC, encrypted as usual with SK_A.

3. CC validates that the block correctly decrypts using SK_A and accepts output on that basis. As described above CC may then re-authenticate the output under its own PK_{CC} or perform whatever other actions are appropriate.
4. CC Returns an Authentication Output Response block to confirm that it has accepted the authenticated output. The form is:

   Authenticated Output Response
   ID_S
   Null terminated text string from CC with implementation-specific response.

5. Authenticator displays response from CC.

In the low-bandwidth case a different protocol is used, one suitable to the limited communications bandwidth available if a person is manually transferring the data between the Authenticator and CC via a telephone connection:

1. Authenticator displays program output on the screen, along with its ID_A (possibly just some fraction of the bits of ID_A is shown, enough to narrow down the possibilities to no more than a handful of Authenticators).
2. User dials CC on the telephone and enters this information using his touch-tone keypad, as prompted by a recorded voice. (This option is severely limited by the number of digits a user can reasonably be expected to type in.)
3. CC tells the user to enter a specific random challenge string using the input devices available, a keypad or a joystick interface.
4. Authenticator calculates a cryptographic hash of the program output and challenge string and encrypts it using SK_A, displaying the result.
5. The User enters this result into the CC again using the keypad.
6. CC calculates the hash and encryption on its own and confirms the value entered by the user.

Several variations are possible. To improve reliability, the values which the user is asked to transfer can be padded with some redundancy to allow some level of error correction if he gets a few digits wrong. A variation on the above protocol uses a separate key than SK_A, one which is the same for all meters and is programmed into their ROM and manufacturing time. If this is done there is no need for the Authenticator to display ID_A in step 1 or the user to enter it in step 2. Step 4 can be done by using a keyed one-way hash function insteat of an encryption function.

6 Conclusions

Real-world applications of cryptography often only require the simplest of algorithms and protocols. We have shown how to take the simple notion of digital signatures and, by combining it with the notion of a secure processor, create a robust application for authenticating the outputs of software. Protocols such as these will most likely play a large role in electronic commerce application.

Further research is required in dividing software into secure and insecure components in such a way that if the Authenticator executes only the secure components, then the Authenticator can determine that specified output for the whole software has actually occurred.

7 Acknowledgments

The authors would like to thank James Jorasch, Jay Walker, and the CARDIS program committee for their helpful comments.

References

[Cha93] D. Chaum and T. Pedersen, "Wallet Databases with Observers," *Advances in Cryptology — CRYPTO '92*, Springer-Verlag, 1993, pp. 89–105.

[Dif76] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, v. IT-22, n. 6, Nov 1976, pp. 644-654.

[ElG84] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, *IEEE Transactions on Information Theory*, v. IT-31, n. 4, 1985, pp. 469-472.

[GUQ92] L.C. Guillou, M. Udon, and J.-J. Quisquater, "The Smart Card: A Standardized Security Device Dedicated to Public Key Cryptography," *Contempory Cryptology: The Science of Information Integrity*, G. Simmons, ed., IEEE Press, 1992, pp. 561-613.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91*, Springer-Verlag, 1991, pp. 17–38.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[NIST94] National Institute of Standards and Technologies, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[RSA78] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, Feb 1978, pp. 120-126.

[Sch94] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 191–204.

[Sch96] B. Schneier, *Applied Cryptography, 2nd Edition*, John Wiley & Sons, 1996.

[YT95] B. Yee and J.D. Tygar, "Secure Coprocessors in Electronic Commerce Applications," *The First USENIX Workshop on Electronic Commerce*, USENIX Association, 1995, pp. 155-170.

This article was processed using the LaTeX macro package with LLNCS style

Biometrics on Smartcards: An Approach to Keyboard Behavioral Signature

Thomas J. Alexandre
PhD., University of Lille
e-mail: Thomas.Alexandre@cs.cmu.edu

Abstract

To overcome the lack of security provided by passwords for authentication and access control, some researches have investigated the field of biometrics for individual identification, such as voice recognition, fingerprints or handwritten signature [CART94][PRIC86][ALEX94a]. Because of the significant amount of processing and memory space required by those approaches, implementing them in a physically secure environment such as a smartcard remains difficult. Moreover, biometric systems based on physiological criteria suffer from the possibility for a potential intruder to imitate these features.

In this paper we propose a new system of biometric identification based on the behavioral recognition of keyboard signature. Such a system provides the user with more security in the sense that a behavior is difficult to copy. The simplicity and reliability of our approach compared to the high power of discrimination it provides makes it suitable for built-in smartcard applications.

A neural network implementation through supervised and self-organizing technics is also discussed in this paper and evaluated in terms of efficiency and performance.

Because this new biometric system must be further evaluated with a large community of users, we finally discuss a proposal for testing purpose on the World Wide Web using the emerging Java language.

Keyword: *Biometrics - Smartcards - Neural Networks - Keyboard Signature*

1. Introduction

The basic idea behind the keyboard behavioral signature we propose is to recognize an individual by the way he's typing on a keyboard. No matter what his ability of typing on a keyboard is (i.e. if he is familiar with typing on a keyboard or not), this recognition process does not deal with extracting dynamic aspects such as the time between two keys are pressed, but resides in measuring the correlations between the pressed keys. These correlations have appeared to be a good way of discrimination between people. The whole process remains very easy to use.

Furthermore, as keyboards are naturally present where biometric systems are usually used (for example to get access to a computer, or to withdraw cash at an ATM), this system does not need any additional hardware whereas other systems do (a microphone is at least required for a voice recognition system, a scanner for fingerprints recognition, or a camera for face recognition). The absence of any additional biometric sensor largely contributes to cost reduction.

Although physiological or behavioral criteria that may be employed in a biometric identification mechanism can be of various types (e.g. voice patterns or fingerprint images), the basic operations used in the identification process are mainly concentrated around the same concepts. The definition of a biometric application generally involves the following two steps:

- An enrollment phase where an individual provides a *Reference* pattern which is a set of features that characterize his identity, such as a fingerprint image or a handwritten signature.
- A verification phase that appears each time this person wants to authenticate to the biometric system, when his *Reference* is compared to a newly offered set of characteristics in order to determine if these two pieces of information correspond to each other.

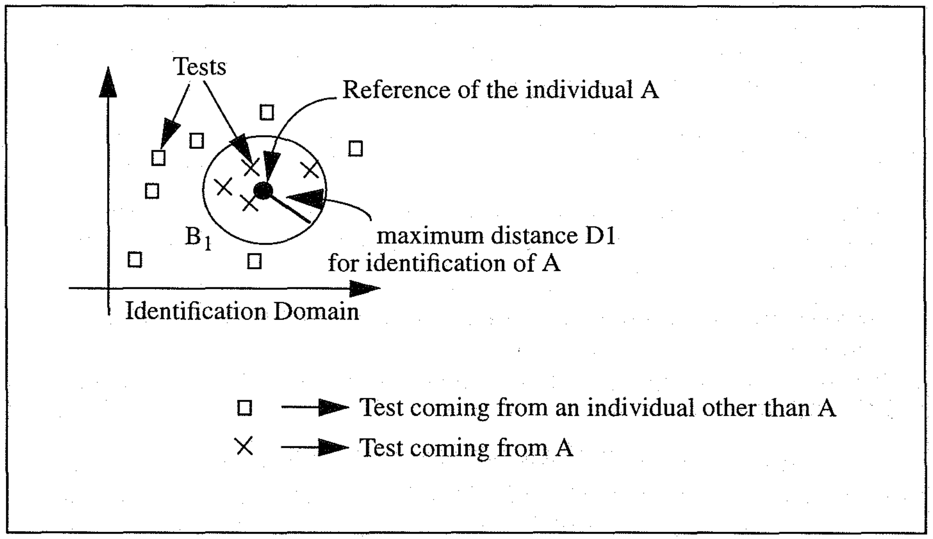

Tests
Reference of the individual A
B1
maximum distance D1 for identification of A
Identification Domain
□ → Test coming from an individual other than A
X → Test coming from A

Figure 1 Identification phase: measure of a distance between a Test and a Reference

As both information (the *Reference* and the *Test* data) usually don't exactly match, a recognition error may appear if an individual's *Test* data is considered too far from its *Reference* or if another individual (an intruder) can provide a *Test* close enough to that particular *Reference* to intrude the system. Thus the two rates commonly admitted for determining the reliability of a biometric system are the FRR (False Rejection Rate that corresponds to a genuine user flagged as fraudulent) and the FAR (False Acceptance Rate that corresponds to undetected fraud) *[CASC94].*

In the following part we explain the reasons why it is interesting to achieve the recognition algorithm in the smartcard.

In part 3 we use the notions described above to build and evaluate our biometric identification system with a simple computation of distance between the *Reference* and the *Test* to implement the verification phase.

In part 4 we introduce the neural network technology *[LIPP87]* to make our behavioral identification more secure through the use of supervised and unsupervised learning methods.

We then discuss the results in terms of efficiency (power processing and template size), reliability and overall performance of the implementation.

The last section of this paper is devoted to testing the reliability of the biometric system through the use of the large community of users present on the World Wide Web.

2. Interest of Performing the Biometric Process in the Card

Smartcards have become widely popular essentially for the two following reasons:

- Security of information: the tamper-proof module of the smartcard ensures the physical security of the data present in the card. Additional software security is provided through the use of authentication protocols to grant access to its owner.
- Portability of information: the credit-card size of the smartcard (compliant to ISO 7816 standard) enables its bearer to carry personal information in a wallet for instance.

To benefit from the tamper-resistance of the smartcard during biometric authentication, the *Reference* template should never be communicated to the outside world. In this sense, the algorithm used to achieve the recognition should be completed inside the card.

Keeping such an algorithm as simple as possible and consuming very little memory space enables to run it on a low performance and affordable smartcard. This is the major reason why we want to propose a biometric recognition process involving little computation.

3. Keyboard Behavioral Signature

3.1. General description of the process

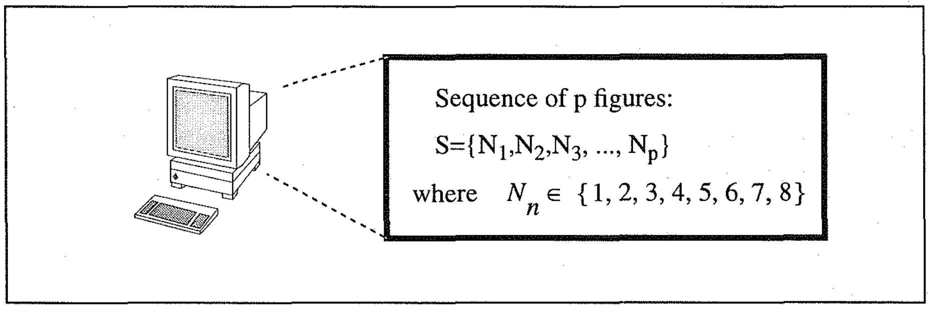

Sequence of p figures:
$S=\{N_1,N_2,N_3, ..., N_p\}$
where $N_n \in \{1, 2, 3, 4, 5, 6, 7, 8\}$

Figure 2 Data recorded for the construction of the *Reference* template

In our system, the data acquisition to build a *Reference* template for an individual consists of the recording of a sequence of 1000 keystrokes, assuming that each finger of the two hands of that person corresponds to exactly one key (except the thumbs for our first set of experiments, so that this individual feels more comfortable if he is unfamiliar with typing). That means the left little finger on the key "1", the left ring finger on key "2" ,...., the right little finger on key "8". The individual has to type p of these 8 possible keys without caring of what he is typing, i.e. in a manner he might think random. Actually this "random" sequence contains some correlations that are the basis of our measurements for discriminating between people. p keystrokes (usually p=1000) should take an approximate time of only 1 or 2 minutes, therefore remains very competitive to some other biometric processes, usually requiring several handwritten signatures or voice sentences.

These 1000 keys can be considered as 10 patterns of the individual's typing characteristics each consisting of a sequence of 100 figures. Therefore we retained as a *Reference* template one pattern computed as the average of the 10 recorded sequences, so that this pattern closely corresponds to the individual's way of typing.

The verification phase involves creating a *Test* pattern of 100 figures that will be confronted to the *Reference* to determine if they both match in respect to a certain tolerance. Thus the time needed for data acquisition during the verification phase is only of a few seconds (about 10 seconds).

3.2. The verification phase computation

In our first approach, we have computed two distances between the *Reference* template and the *Test* data.

The first one is based on the frequencies of appearance of each of the 8 possible keys. A table of frequencies of appearance could look like the following:

Key ID : i	1	2	3	4	5	6	7	8
Frequency: F_i	7	12	14	8	24	17	8	10

Table 1 Frequencies of appearance of each of the 8 possible keys

Assuming Ft_i the frequency of appearance of key i for a Test pattern T and Fr_i the frequency of appearance of key i for a Reference pattern R, a simple euclidian distance between the two patterns can be given by:

$$D_1(t, r) = \sqrt{\sum_{i=1}^{8} \left(Ft_i - Fr_i\right)^2}$$

The second distance involves the frequencies of appearance of (N_i,N_{i+1}) pairs which represent the correlations between two successive keys, such as the example below:

N_i \ N_{i+1}	1	2	3	4	5	6	7	8
1	**0**	**48**	**9**	**10**	**31**	**21**	**12**	**14**
2	**15**	**0**	**36**	**9**	**<u>16</u>**	**20**	**7**	**7**
3	**8**	**19**	**0**	**17**	**21**	**22**	**5**	**8**
4	**40**	**13**	**8**	**1**	**65**	**37**	**12**	**13**
5	**39**	**14**	**35**	**81**	**2**	**11**	**3**	**8**
6	**23**	**11**	**8**	**40**	**22**	**0**	**17**	**8**
7	**7**	**3**	**0**	**11**	**25**	**12**	**0**	**9**
8	**12**	**2**	**5**	**20**	**11**	**6**	**11**	**0**

Table 2 Frequencies of (N_i, N_{i+1}) pairs during an enrollment phase: this is an example of distribution of the (1000-1) pairs of 2 consecutive keys. For instance, the number of times that the key "5" has been typed right after "2" in the sequence of 1000 keys is 16.

The euclidian distance here between the array corresponding to a Test pattern T and the array corresponding to a Reference pattern R can be calculated by:

$$D_2(t, r) = \sqrt{\sum_{i=1}^{8} \sum_{j=1}^{8} \left(Ft_{(i,j)} - Fr_{(i,j)} \right)^2}$$

Although D_2 is a lot more discriminative than D_1 to classify patterns, we have used both distances for identifying an individual.

3.3. Experimental results

We have evaluated the performance of this biometric identification system on a set of 30 people each performing 1 enrollment phase and 10 test procedures. The whole mechanism was written in C.

For each individual, comparing his *Reference* pattern to his 10 *Test* patterns resulted in no failure, i.e. none of the genuine users trying to authenticate were rejected (FRR=0).

For each individual, we have presented to his *Reference* the 10 *Test*s from the 29 other individuals (i.e. 290 attempts to imitate the genuine user). 30 References each being tested on 290 possibilities make a total of 8700 identification attempts. Only 8 out of the 8700 Tests gained access, thus showing a FAR of 0.092%. Because 10 *Test* patterns from the same individual are quite close to each other it makes sense to only use 1 out of the 10 to challenge the *References* from other individuals. Therefore a more likely FAR calculated from only 870 attempts would be much higher but still below 1%.

In order to evaluate more accurately the system, we should be able to test it on a much larger database of patterns, in a similar way than other biometric processes, including profiles of users chosen in a wide population. Such a database with keyboard signature patterns is not yet available and is needed. We further discuss this issue at the end of the paper.

To limit the exposure to a party who could monitor the enrollment or verification phase of another party and subsequently misrepresent himself, we need to measure the time taken to complete an enrollment or verification phase and validate only those achieved within a certain amount of time.

Because our first analysis was restricted to correlations of first order for only pairs of sucessive data, we have considered in a second approach correlations of any order to improve the accuracy of identification. The section that follows introduces the Neural Network Technology to optimize the *Reference* template constitution and the distance computation. However, the smartcard implementation using neural networks requires more processing power than the statistical analysis described so far.

Furthermore, all of the pairs (N_i, N_{i+1}) don't necessarily have the same power of discrimination between people. Some of them are useless for the identification because they are not invariant at all, and should not been taken into account during the identification process. Self-organizing technics such as Kohonen's Topological Maps *[KOHO84]* can help on this matter and are therefore also examined in the next part.

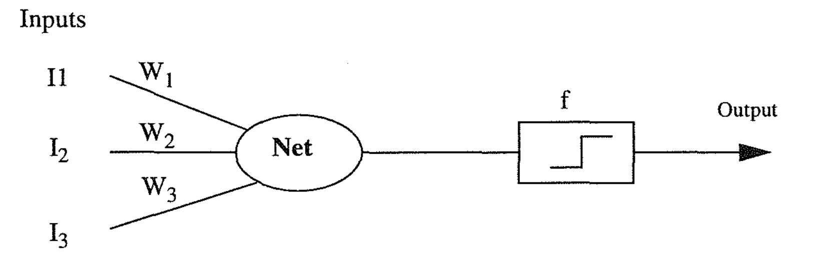
4. A Neural Network Approach

4.1. Introduction to Neural Networks

The neural technology is based on the modelization of the neural structure of the brain on which scientists have been doing intensive research to understand its biological structure and behavior. Basically, the human brain is composed of more than 100 billion highly interconnected neurons delivering electric pulses of various intensity depending on their activity due to the operations to be achieved.

From biological experiments have resulted different artificial computer modelizations of various complexity. Here we are going to briefly detail the few concepts of Neural Networks useful for our implementation. A more detailed introduction to Neural Networks is available in *[LIPP87] and [MINS69].*

4.1.1. A feed forward Neural Network



$$Output = f\left(\sum_{i=1}^{n} Wi \times Ii + b\right)$$

where (W1,....,Wn) are the weights
(I1,..., In) the inputs of the network
b the bias
and f a threshold function

Figure 3 A single neuron with 3 inputs

A neural network can be seen as a black box with inputs and outputs performing a special function or task. For any given set of inputs (I_1, I_2,..., I_N) applied to a single processing unit (neuron) the ouput is calculated by summing these inputs, weighted by elements (W_1,W_2,...,W_N) of the weight matrix W which represent the contribution of each input. The resulting value is then compared to a threshold. If the threshold is bigger than the net input then a zero value is given at the output, otherwise a one is given.

4.1.2. The Back-propagation algorithm

Being a supervised learning algorithm, the Backpropagation learning rule relies on a teacher which is a set of example pairs of patterns. The basic idea of this learning process is to minimize the error commited at the output (which is the difference between the output given by the network and the target output) by retropropagating the error backward the network for adjusting the connection weights. The complete description of the Back-propagation algorithm is given in *[RUME86]*.

4.1.3. Unsupervised learning

In the preceding section we have introduced a supervised learning method, the *Backpropagation*. This mechanism requires to have available example patterns that include targeted outputs, i.e. how to organize the data to correctly respond to a problem.

Self-organizing in Networks doesn't need to rely on a teacher which tells how to classify the training data. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Feature maps learn to recognize groups of similar input vectors in such a way that neurons physically close together in the neuron layer respond to similar input vectors. Kohonen's work on this matter has led to a different approach from the supervised learning methods and based on biological experiments *[KOHO84]*.

4.2. The keyboard behavioral signature using Back-propagation

We have experimented the verification phase of the keyboard signature through the use of Back-propagation, which is the most popular algorithm for supervised learning. Because the basic algorithm suffers from a number of disadvantages (long training times depending on initialization values *[KOLE91]*, possibility for the error to be stuck in a local minima, overfitting etc...) *[ALEX94b]*, we have implemented variations on this algorithm detailed in *[VOGL88]* and *[NGUY90]*.

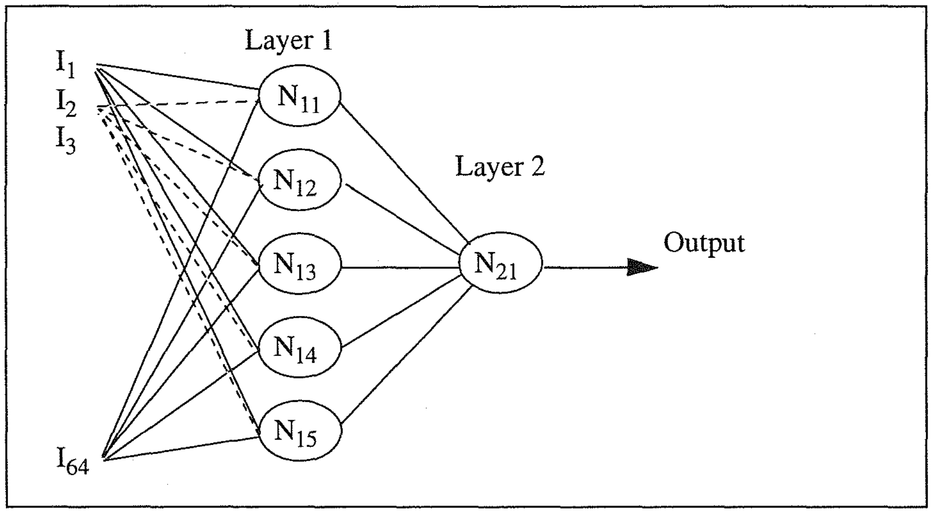


Figure 4 Neural Network for Keyboard Signature Verification

The network, consisting of 64 inputs (+ a bias) fed to a two-layer network, is trained with backpropagation on a 10 pattern problem. The inputs correspond to the 64 frequencies of appearance of (N_i,N_{i+1}) pairs.

Layer 1 consists of 5 tan-sigmoid neurons, while layer 2 is made of one linear neuron. The network is initialized with the Nguyen-Widrow method *[NGUY90]*. An adaptive learning rate which adjusts to be as large as possible without taking too large of steps speeds up the network while avoiding large increases in error. The momentum technique to avoid being trapped in local minimas is also implemented.

We have written all the network description and the training algorithm using the Matlab programming language *[DEMU92] [NAZA92]*. 11 epochs were sufficient for providing a global sum-squared error as low as 0.0023. Training took about 2 seconds on a 80486 microprocessor running 50 Mhz. Such trained networks have been tested both by new patterns of the genuine user and other patterns. None of the genuine users were rejected, and none of the other patterns intruded the system. However, we couldn't test this system on a very large population because such a database does not exist.

The intention being to implement the recognition process into a smartcard, i.e. with a reduced storage capacity and a limited processing power, we have calculated the memory space needed for storing the neural network coefficients. Assuming 4 bytes for storing a weight coefficient (i.e. floating point values), the first layer of 65 inputs connected to 5 neurons represents 65*5*4 = 1300 bytes. The second layer can be described in the same way as 6*1*4 = 24 bytes (5 connections + 1 bias), leading to a total size of 1324 bytes.

Because 1324 bytes is already a lot for trying to implement the feed forward process on a smartcard, we have used Kohonen Self-Organizing Feature Maps to extract from the 64 inputs only those that are actually the most useful for the reliability of the system (because some (N_i,N_{i+1}) pairs are more discriminative than others). In that sense, the additional first layer we have implemented using Kohonen Feature Maps was acting as a preprocessing.

5. Reliability Testing

Despite the pretty good performance that we have obtained in testing the keyboard signature process on a restricted set of individuals with the two methods presented in the previous sections, some strong additional testing must be performed on a much larger community to validate the biometric process for potential smartcard applications.

Large databases of keyboard signature profiles don't currently exist and therefore make the testing process somehow difficult. However, the recent tremendous evolution of the Internet has suddenly grouped together a large community of people ready to discuss, use, experiment and evaluate a number of software, reports or ideas.

Because our biometric process doesn't require any other biometric sensor than a keyboard for data acquisition, we are implementing a test version on the World Wide Web so that this large community of users can provide us with new patterns to further evaluate the reliabilty of the system.

The easiest and fastest way to implement it is to set up a web server with HTML pages explaining how the system works and containing HTML forms where potential users can complete the enrollment and verification phases by filling out some fields with sequences of figures that they would type in a "random" way, assuming that they are playing the game. The information submitted via their web browser can then be processed by a program sitting on the server's side (typically what is popularly called a CGI script (common gateway interface)) that would create a large database of keyboard signature profiles.

The main issue that comes up with such an implementation is one of test validation. How can we know that the person is not cheating by typing a sequence of figures in a prearranged order, or how can we know that the person submitting the enrollment data is the same person than the one filling out the verification data? These kinds of unvalid data might compromise the real evaluation of the biometric reliability if they are not eliminated.

To limit the exposure to such attacks, we are now rewriting the implementation of the testing process using Java applets *[VANH95]*. Java is an object-oriented programming language that has become one of the most popular on the world wide web because of a number of advantages among which its portability and its flexibility. A Java applet is a Java program that can be included in an HTML page, much like an image can be included. When you use a Java-compatible browser to view a page that contains a Java applet, the applet's code is transferred to your system and executed by the browser.

Because the java applet is executed on the client side and not the server side like a CGI script would be, we are able to perform some real-time data checking while the person performing the biometric enrollment or test phase is typing on the keyboard. For example we can measure the time between two consecutive pressed keys or the time for typing the entire sequence of 1000 figures. The faster the sequence is completed, the higher the chance that the person is not cheating.

6. Conclusion

In this paper we have introduced a new form of biometric identification, the keyboard behavioral signature. Such an algorithm has first been implemented by the computation of first and second order distances only and therefore can run into a low-performance and very affordable smartcard, especially since no sensor is required. To take into account correlations of higher orders, we have used the Neural Networks technology. The verification algorithm can be seen as a feed forward neural network process and therefore can easily be achieved by a robust smartcard such as the RISC-based smartcard architecture described in *[PEYR94]*.

We are still optimizing the size of the template and the efficiency of the process by using new learning methods, especially the Cascade-Correlation algorithm *[FAHL90] [FAHL91]*, where the size of the trained network is always kept minimum unlike the traditionnal Back-propagation, thus accelerating the convergence and training time. We are also using additional technics for optimizing the topology of self-organizing feature maps, with the computation of a Topographic Product detailed in *[BAUE92]*.

The acquisition of much more keyboard signature profiles through the use of the Internet community will enable to further evaluate the reliability of the entire process.

REFERENCES

[ALEX94a] Thomas Alexandre, Vincent Cordonnier, «An Object-Oriented Approach for implementing Biometrics in Smart Cards», in *Proceedings of the CardTech/SecurTech'94 International Conference*, Arlington,Virginia, USA, April 10-13, 1994, pp 149-160. Can be found at: http://www.cs.cmu.edu/afs/cs/user/alexandr/WWW/publis.html.

[ALEX94b] Thomas Alexandre, «The Radar Concept using Neural Networks», in *Proceedings of the First Smart Card Research and Advanced Application Conference CARDIS '94*, Lille, France, october 24 - 26 1994. http://www.cs.cmu.edu/afs/cs/user/alexandr/WWW/publis.html.

[BAUE92] Hans-Ulrich Bauer, Klaus Pawelzik, Theo Geisel, "A Topographic Product for the Optimization of Self-Organizing Feature Maps", in *Lippmann, Moody, Touretzky (eds), Advances in Neural Information Processing Systems 4, Morgan Kaufmann*, San Mateo, CA, 1992.

[CART94] Bob Carter, «The Present and Future State of Biometric Technology», in *Proceedings of the CardTech/SecurTech'94 International Conference*, Arlington,Virginia, USA, April 10-13, 1994, pp 401-415.

[CASC94] CASCADE Consortium, «Biometrics», Deliverable 1 of the ESPRIT - CASCADE European Project EP8670: Technology Assessment, May 4, 1994, Section5.

[DEMU92] H. Demuth, M. Beale, Neural Network Toolbox for Use with Matlab, The MathWorks Inc., 1992.

[FAHL90] Scott E. Fahlman, Christian Lebiere, "The Cascade-Correlation Learning Architecture", CMU-CS-90-100 Technical Report, Carnegie Mellon University, Pittsburgh, USA, 1990.

[FAHL91] Scott E. Fahlman, "The Recurrent Cascade-Correlation Architecture", in *Lippmann, Moody, Touretzky (eds), Advances in Neural Information Processing Systems 3, Morgan Kaufmann*, San Mateo, CA, 1991.

[GROS91] Stephen Grossberg, Gail A. Carpenter, «Pattern recognition by self-organizing neural networks», *The MIT Press*, Cambridge, MA, 1991.

[HOLD89] Ronald M. Holdaway, "Enhancing Supervised Learning Algorithms Via Self-Organization", in *Proceedings of the International Joint Conference on Neural Networks*, Washington D.C., June 18-22, 1989, Vol.II, pp 523-530.

[KOHO84] Teuvo Kohonen, *Self-Organization and Associative Memory*, Springer-Verlag, 1984.

[KOLE91] John F. Kolen, Jordan B. Pollack, "Back-Propagation is Sensitive to Initial Conditions", in *Lippmann, Moody, Touretzky (eds), Advances in Neural Information Processing Systems 3, Morgan Kaufmann*, San Mateo, CA, 1991.

[LIPP87] R. P. Lippmann, "An introduction to computing with Neural Networks", *IEEE ASSP Magazine*, 3:422, April 1987.

[LUCA90] S. M. Lucas, R. I. Damper, "Signature verification with a syntactic neural net", in *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, June 17-21, 1990, Vol.I, pp 373-378.

[MINS69] M. Minsky, S. Papert, *«Perceptrons»*, The MIT Press, 1969.

[NAZA92]	Jamshid Nazari, Okan K. Ersoy, "Implementation of back-propagation Neural Networks with Matlab", EE 92-39 Technical Report, School of Electrical Engineering, Purdue University, West Lafayette, Ind., 1992.
[NGUY90]	D. Nguyen, B.Widrow, «Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights», *International Joint Conference on Neural Networks*, Vol3, July 1990, pp 21-26.
[PEYR94]	P. Peyret, «RISC-based, Next-generation Smart Card Microcontroller Chips», in *Proceedings of the CardTech/SecurTech'94 International Conference,* Arlington,Virginia, USA, April 10-13, 1994.
[PRIC86]	W. L. Price, "A review of methods of personal identity verification", DITC 73/86 Technical Report, Division of Information Technology an Computing, National Physical Laboratory, Teddington, GB, 1986.
[RITT88]	H. Ritter, K. Schulten, "Convergence Properties of Kohonen's Topology Conserving Maps: Fluctuations, Stability and Dimension Selection", *Biology Cybernetics* 60, pp 59-71, 1988.
[RUME86]	D. E. Rumelhart, G. E. Hinton, R. J. Williams, «Learning internal representation by error propagation», D.Rumelhart and J.McClelland editors, *Parallel Distributed Processing: exploring the Microstructure of Cognition*, Vol.1, MIT Press, Cambridge, MA, 1986.
[SHEP94]	Colin Sheppard, «A Neural Network Approach to Fingerprint Verification», in *Proceedings of the CardTech/SecurTech'94 International Conference*, Arlington,Virginia, USA, April 10-13, 1994, pp 183-190.
[SONA89]	N. Sonehara, M. Kawato, S. Miyake, K. Nakane, "Image Data Compression Using a Neural Network Model", in *Proceedings of the International Joint Conference on Neural Networks*, Washington D.C., June 18-22, 1989, Vol.II, pp 35-41.
[VANH95]	Arthur Van Hoff, Sami Shaio, Orca Starbuck, *Hooked on Java,* Addison-Wesley, 1995. See also http://java.sun.com/.
[TOLA89]	Viral V. Tolat, Allen M. Peterson, "A Self-Organizing Neural Network for Classifying Sequences", in *Proceedings of the IJCNN'89*, Washington D.C., June 18-22, 1989, Vol.II, pp 561-568.
[VOGL88]	T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, D. L. Alkon, «Accelerating the convergence of the backpropagation method», *Biological Cybernetics*, Vol.59, 1988, pp 257-263.

[illegible]

Arithmetic co-processors for public-key cryptography : The State of the Art

David Naccache **David M'Raïhi**

Gemplus PSI - Cryptography Department
1 Place de la Méditerranée, F-95200, Sarcelles, France
email : {100142.3240, 100145.2261}@compuserve.com

(Adapted from various manufacturer data-sheets and an earlier work by ***V. Cordonnier*** *- Gemplus RD2P)*

Abstract. Smart-cards have the tremendous advantage, over their magnetic stripe ancestors, of being able to execute cryptographic algorithms locally in their internal circuitry. This means that the user's secrets (be these PIN codes or keys) never have to leave the boundaries of the tamper-resistant silicon chip, thus bringing maximum security to the overall system in which the cards participate.

Smart-cards contain special-purpose microcontrollers with built-in self-programmable memory and tamper-resistant features intended to make the cost of a malevolent attack far superior to the benefits. As cryptography progresses, so does silicon technology : new silicon geometries and cryptographic processing refinements are introduced at a fast pace by the semiconductor manufacturers serving this industry. This paper is both a survey of the existing crypto-dedicated microprocessors and an attempt to describe some of their possible evolutions.

☞ 600 million IC cards will be manufactured in 1996 throughout the world.

keywords : smart cards, public key, crypto co-processors, modular multiplication

1. What is a smart-card ?

The idea of inserting a chip into a plastic card is as old as public-key cryptography. The first patents are now twenty year's old but practical public-key applications emerged only a few years ago due to limitations in storage and processing capacities of past circuit technology. New silicon geometries and cryptographic processing refinements lead the industry to new generations of cards and more ambitious applications such as RSA[1] or the new DLP-based[2] Digital Signature Standard [6].

Over the last four years there has been an increasing demand for public-key smart-cards from national administrations and large companies such as telephone operators, banks and insurance corporations. More recently, another market opened up with the increasing popularity of home networking and Internet.

The physical support of a conventional smart-card[3] is a plastic rectangle on which can be printed information concerning the application or the issuer (even advertising) as well as readable information about the card-holder (as for instance, a validity date or a photograph). This support can also carry a magnetic stripe or a bar code label. An array of eight contacts is located on the micromodule in accordance with the ISO 7816 standard but only six of these contacts are actually connected to the chip, which is (usually) not visible. The contacts are assigned to *power supplies* (Vcc, Vpp), *ground*, *clock*, *reset* and a *serial data* communication link (commonly called I/O). Their specification part in the standard is currently reconsidered upon requests from various parties (suppression of the two useless contacts, creation of a second I/O port, I2C bridging etc.).

For the time being, card CPUs are still 8 bit microcontrollers and the most common cores are Motorola's 68HC05 and Intel's 80C51 but new 32-bit devices will soon begin to appear. RAM capacities (typically ranging from 76 to 512 bytes) are very limited by the physical constraints of the card. The program executed by the card's microprocessor is written in ROM at the mask-producing stage and cannot be modified in any way. This guarantees that the code is strictly controlled by the manufacturer. For storing user-specific data, individual to each card, the first

[1] signature and encryption algorithm by *Rivest, Shamir and Adleman* [10], based on the difficulty of factoring large integers.
[2] DLP stands for *Discrete Logarithm Problem*.
[3] contactless smart-cards are dedicated to hands-off applications such as public transportation or toll highways. They rely on radio transmission or high frequency technologies and are not systematically embedded in a plastic support. Since these devices, rarely include public-key accelerators, we excluded them from this survey.

generation of non-volatile memories used EPROMs[4] which required an extra "high" voltage power supply (typically from 15 V to 25 V). Recent components only contain EEPROM which requires a single 5 V power supply (frequently that of the microprocessor) and can be written and erased thousands of times (cycles). Sometimes, it is possible to import executable programs into the card's EEPROM according to the needs of the card holder. EEPROM size is a critical issue in the design of public-key applications (where keys are relatively large). Consequently, smart-card programmers frequently adopt typical optimization techniques such as re-generating the public-keys from the secret-keys when needed, re-generating the secret-keys from shorter seeds, avoiding large-key schemes (for instance Fiat-Shamir [5]) or implementing compression algorithms for redundant data (text, user data, etc.) and EEPROM garbage collection mechanisms. Real and complete operating systems have been developed for this purpose by several manufacturers[5]. Finally, a communication port (serial via an asynchronous link) for exchanging data and control information between the card and the external world is available. A common bit rate is 9600 bits/s but much faster interfaces are commonly being used (from 19,200 up to 115,200 bits/s) in full accordance with ISO 7816.

A first rule of security is to gather all these elements into a single chip. If this is not done, the external wires, linking one chip to another could represent a possible penetration route for illegal access (or use) of the card. ISO standards specify the ability of a card to withstand a given set of mechanical stresses. The size of the chip is consequently limited and present constraints (especially memory and cryptographic capabilities) mainly follow from this limitation.

Smart-card chips are very reliable and most manufacturers guarantee the electrical properties of their chips for ten years or more. ISO standards specify how a card must be protected against mechanical, electrical or chemical aggressions but for most existing applications, a card is far obsolete before it becomes damaged. A well known example is the French phone card where the failure rate is less than three per 10,000 pieces.

[4] Electrically PROgrammable Memory (EPROM) and Electrically Erasable PROgrammable Memory (EEPROM or E^{2}PROM); non-volatile memory (NVM) means usually provided for data storage.

[5] for further details, contact `ppeyret@integrityarts.com`

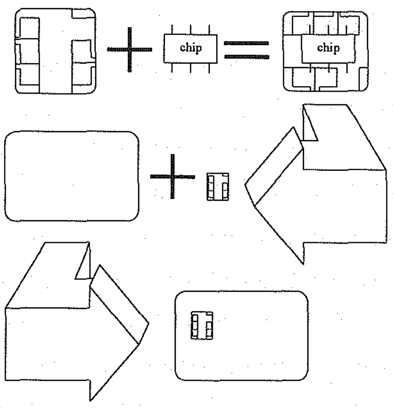


Figure 1. Smart-card manufacturing

The following table lists some of today's most common (not cryptography-dedicated) chips as well as their characteristics :

Name	Core	Manuf.	RAM	ROM	NVM	Detectors	Surface	◷
SC01	68HC05	Motorola	36	1.6 K	1 K ⇩	N.S.	3.5 × 5.5	4
SC03	68HC05	Motorola	52	2 K	2 K ⇩	N.S.	3.5×5.6	4
SC11	68HC05	Motorola	128	6 K	8 K ⇩	F, V	3.5×5.6	4
SC21	68HC05	Motorola	128	6 K	3 K ⇳	F,V	3.5 × 5.6	4
SC24	68HC05	Motorola	128	3 K	1 K ⇳	F, V	4.14×3.44	5
SC26	68HC05	Motorola	160	6 K	1 K ⇳	F, V	13.4 mm2	5
SC27	68HC05	Motorola	240	16 K	3 K ⇳	F, V	5×5.4	5
SC28	68HC05	Motorola	240	12.8 K	8 K ⇳	F, V	27 mm2	5
ST1821	68HC05	SGS	44	2 K	1 K ⇩	F, T, V, L	3.4 × 5.36	5
ST1834	8048	SGS	76	4 K	3 K ⇩	F, T, V, L	3.61× 5.79	5
ST16612	8048	SGS	224	6 K	2 K ⇳	F, T, V, L	5.66 × 5.87	5
ST16601	68HC05	SGS	128	6 K	1 K ⇳	F, T, V, L	10.1 mm2	
ST16623	68HC05	SGS	224	6 K	3 K ⇳	F, T, V, L	3.66 × 6.61	5
ST16F44	68HC05	SGS	512	16 K	8 K ⇳	F, T, V, L	18.6 mm2	5
ST16F48	68HC05	SGS	512	16 K	8 K ⇳	F, T, V, L	4.8 × 4.9	5
ST16301	68HC05	SGS	160	3 K	1 K ⇳	F, T, V, L	3.66 × 4.93	5
65901	proprietary	Hitachi	128	3 K	3 K ⇳	W	secret	5
6483108	H8300	Hitachi	256	10 K	8 K ⇳	W	secret.	5
H8310	H8300	Hitachi	256	10 K	8 K ⇳	W	5.3×5.2	5
H83102	H8300	Hitachi	512	16 K	8 K ⇳	W	18.08 mm2	5
62720	proprietary	Oki	128	3 K	2 K ⇳	secret.	secret	N.S.
62780	proprietary	Oki	192	6 K	8 K ⇳	secret.	secret	N.S.
44C10	80C51	Siemens	128	4 K	1 K ⇳	«hardware»	13 mm2	5
44C40	80C51	Siemens	256	8 K	4 K ⇳	«hardware»	18.39 mm2	5
44C80	80C51	Siemens	256	16 K	4 K ⇳	«hardware»	24.49 mm2	5

Table 1. Common smart-card chips

Where 🕒 = maximal clock rate given in MHz, ⇳ = EEPROM, ⇩ = EPROM, *Witness cell (detects if the EEPROM was erased abnormally), Clock Frequency., Temperature, Abnormal Voltage (Vpp), Light exposure and passivation sensor.* Information about security detectors and tamper-resistance capabilities is usually rather hard to obtain from the manufacturers (for obvious security reasons).

In general, smart-cards can help whenever secure portable objects are needed, and in particular whenever the « external world » needs to work with data without knowing its actual value. The card's tamperproofness, combined with public-key cryptography (secretless terminals), generally provide adequate solutions to many everyday security problems.

2. Smart-card communication and command format

Communication with smart-cards is ruled by the (previously-mentioned) ISO standard 7816/3. Only two protocols are currently defined in this standard (byte-oriented T = 0 and block-oriented T = 1) although up to 14 are reserved (T = 14 is very rare, and means that the communication protocol is proprietary). Thus the electrical levels and error handshakes as well as the frequency used impose a specific hardware on the « external side » which is the equivalent of a UART with more sophisticated functions.

This minimal hardware, needed to operate a card, consists of :

- a mechanical interface : the *connector*
- an electronic interface : the *coupler*
- and a box containing the above two elements : the *smart-card reader* (or simply « *reader* »)

The simplest readers are quite similar to modems and manage only the ISO communication protocol without interacting « intelligently » with the operating system of the card. They are called « transparent readers » and should (but in practice may...) operate with any smart-card from any vendor which complies with the ISO standard.

The most sophisticated readers can be programmed with parts of the application logic and contain data (for instance RSA or DSA public keys), files and programs. They can execute cryptographic functions, replace completely a PC, have keyboards, pin-pads or displays and generally use a specific programming language and do not support all types of smart-cards even if they comply with the ISO standard (because they often integrate particular commands dedicated to a given card).

To operate a card, the reader needs to implement the following four functions :

➊ Power on/off the card

➋ Reset the card

➌ Read data from the card (*get commands*).

➍ Write data to the card (*put commands*).

Get and put commands contain a header (actually a function code consisting of 5 bytes designated by CLA, INS, P1, P2 and LEN) according to which the card processes the incoming data. An acknowledge byte and a couple of status bytes (SW1 and SW2) are sent during (and after) the execution of each command.

3. Arithmetic co-processors

3.1. Mathematics

Modular multiplication ($d = t \bmod n$ where $t = a \times b$) is probably the most frequently performed operation in modern cryptography [2] and efficient ways[6] for performing this operation, are a major issue in most applications. However, in most cryptosystems, a and b change much more frequently than n and therefore, most cryptographic accelerators are optimized for performing modular arithmetic over a « rarely-changing » n.

An *arithmetic co-processor* (*ACP*) is a dedicated hardware for computing d, or in some cases, an entire modular exponentiation. Generally, the ACP is « seen » by the card's microcontroller as a set of special RAM addresses where data is written (a, b, n) or read (d). ACPs usually operate in four steps[7] :

① hardware reset and initialization

[6] measured in terms of *speed* and/or *hardware complexity*.
[7] possibly integrated as a macro-command in a library provided by the silicon manufacturer.

② loading (or refreshing) the operands a, b and (optionally) n

③ multiplication (possibly repeated)

④ unloading the result

Step ① (which does not depend on the operand size N) is performed in a constant number of clock cycles (constant time for a given clock frequency), operations ② and ④ (which consist in manipulating data quantities which length is generally equal or proportional to N) are linear in N. Classically, multiplications (③) requires a number of clock cycles which grows as the square of N. The resulting total time is therefore generally proportional to a second degree polynomial of N.

Since the algorithms for computing d are generally public, we encourage (the *really interested*) readers to solicit a copy of **Gemplus**' reference list (over 60 entries) for accurate details about these methods and their applications. Nevertheless, here is a (brief) presentation of the mathematics behind five of these schemes :

Montgomery (Motorola, Thomson, Univ. Catholique de Louvain)

In 1987, Montgomery [8] published an elegant algorithm for computing $d' = ab2^{-N} \bmod n$. Montgomery simply cancels the least significant ones of t by adding to t appropriate multiples of n and shifting t to the right. As this scheme outputs d' (instead of d), a correction of the result must be done by applying Montgomery's algorithm to d' and the (typically pre-calculated) constant $4^N \bmod n$. In itself, the algorithm is very simple (at initialization $d' = 0$):

```
for $i = 0$ to $N-1$
        $d' = d' + b * a[i]$
        $d' = d' + n * d'[0]$
        shift $d'$ to the right by one bit
if $d' < n$ then return ($d'$) else return ($d' - n$)
```

de Waleffe & Quisquater (Philips)

de Waleffe and Quisquater observe that the « black-box » $f(y,x,c) = yx + c$, where x is an N-bit integer, y a v-bit register (typically $v = 24$) and c an $N + v$-bit accumulator, can be used for computing the product t (c is then the multiplication accumulator, x is a and y is the i-th v-bit chunk of b) *and* reducing the so-obtained t modulo n by adding to $c = t$ an appropriate multiple $y = k$ of $x = -n = 2$'s complement of n.

k is easy to estimate when n's v MSBs are 1000000... (it can be proved that any n can be multiplied by an appropriate factor u such that $u*n$'s v MSBs are 1000000..., the reader can find further details about these normalization techniques in [7]).

d is computed by the following loop[8] (when computation starts, $d = 0$) :

for $i = n-1$ to 0 by steps of -v reduce $d = f(2^v, d, f(b[i], a, 0))$ by v-bits using f.

Further details concerning the internal structure of f can be found in [12].

Levy-dit-Vehel & Naccache (Gemplus)

This algorithm computes t with a triplexed serial-parallel multiplier (that is, a hardware block that multiplies simultaneously $a*x$, $a*y$ and $a*z$ provided that $x \wedge y = x \wedge z = y \wedge z = 0$), reduction is done by a variant of Barrett's algorithm (see below).

① split b to two $N/2$-bit blocks $b = b''\|b'$ and let $x = b' \wedge b''$

② compute

$$\{u = a*x, v = a*(b' \oplus x), w = a*(b'' \oplus x)\} = \text{multiplier}[a; \{x, b' \oplus x, b'' \oplus x\}]$$

③ compute $t = \text{delay}_{N/2}[u + w] + u + v$

Bucci and other variants of Barrett (Amtec)

Barrett [1] approximates d by a quasi-reduced $d' \equiv d$ such that $d' < d + 2n$. The algorithm uses a parameter L which limits in advance the maximal size of t ($L = 2N$ is the optimal choice for exponentiation) and a pre-computed constant :

$$k = \lfloor 2^L / n \rfloor$$

In optimized designs, Barrett and Montgomery are roughly equivalent in gatecount (for a given performance and 512-bit arguments, although our benchmarks reveal a slightly bigger second-degree coefficient for Barrett) :

[8] frequently called « *English multiplication* » by French-speakers

① $d = t - n((k(t >> (N-1))) >> (L-N+1))$

② while $d \geq n$, subtract n from d (one can prove that this step is done at most twice)

③ return(d)

where $x >> i$ denotes a shift-to-the-right of x by i bits.

Sedlak (Siemens)

In this reduction algorithm, comparisons of t with $1/3$, $1/6$,... of n play a very important role. Assume that, during the division of t by n, n was already shifted and subtracted several times from t and let the remainder be stored in t. Sedlak's algorithm ensures that $|t| \leq n/3$.

This will be also true for the following steps as will be shown now.

By assumption, t falls into one of the following ranges:

range 2 $\qquad \frac{n}{6} < |t| < \frac{n}{3}$

range 3 $\qquad \frac{n}{12} < |t| < \frac{n}{6}$

$\vdots \qquad \vdots$

range i $\qquad \frac{n}{3*2^{i-1}} < |t| < \frac{n}{3*2^{i-2}}$

for some $i > 1$.

Now n is shifted i bits to the right and for the result $n' = \frac{n}{2^i}$ we have :

$$\frac{2^i n'}{3*2^{i-1}} < |t| < \frac{2^i n'}{3*2^{i-2}} \Rightarrow -\frac{n'}{3} < |t| - n' < \frac{n'}{3}$$

For the result, $t' = t \pm n' \leq n'/3$ holds again and since (we hereby proved that) this condition is always met, n is shifted in every step by at least $i = 2$ bits. However, the expected value of i is 3, thereby improving the reduction complexity by an average factor of 1/3, when compared to the basic bit-by-bit reduction.

3.2. Technology

We solicited inputs, concerning the commercial, technical and algorithmic aspects of the ACPs from the following manufacturers :

✉ **SEPT**	✉ **Thomson**
✉ **Cylink Corporation**	✉ **Atmel**
✉ **Philips**	✉ **Siemens**
🗁 **Oki**	✉ **Pijnenburg**
✉ **Fondazione Ugo Bordoni**	🗁 **Toshiba**
🕮 **NTT**	**? Hitachi**
✉ **Motorola**	**? SANDIA**
✉ **Univ. Catholique de Louvain**	🕮 **Queen's Univ.**

✉ answered our questionnaire (some answers were possibly missing or incomplete).

🕮 no answer, but data is available from conference proceedings, patents and publications.

🗁 no answer, but some data is available from **Gemplus**' competition analysis department.

? no answer, nearly nothing is known about the chip (actually *one bit* of information : *it exists*...).

Readers can contact directly the following *chip manufacturers* for precise details on a given ACP :

for information about	*contact Mr*	*at*	☎ Int +
ST16CF54 or KL74	*Thomason*	B.P. 2, 13106, Rousset, **France**	33 + 42.25.89.44
SLE44C200	*Kiebler*	Po.B. 801709, Munich, **Germany**	49 + 89.41.44.47.79
P83C85/2/5/8	*Philipp*	Po.B. 540240, 22502, Hamburg, **Germany**	49 + 40.56.13.27.47
MC68HC05SC29	*Goupil*	18 rue Grange Dame Rose, 78143 Vélizy, **France**	33 + 1 34.63.59.00
RSA512	*Secco*	via P. da Palestrina 63, 00193, Roma, **Italy**	39 + 63.22.21.70
SCALPS	*Quisquater*	place du Levant 3, Louvain-la-Neuve, 1348, **Belgium**	32 + 10.47.25.41
CY512i	*Rodriguez*	910 Hermoza Court, Sunnyvale, CA 94086, **USA**	1 + 408 735.66.90
CRIPT	*Venard*	B.P. 6243, 14066, Caen Cedex, **France**	33 + 31.75.92.12
PCC200	*v.d. Vleuten*	Po.B. 330, 5260, AH Vught, **The Netherlands**	31 + 73.84.84.50

Table 2. Information sources

The following tables provide synthetic data (technical, commercial and benchmark) about these different ACPs. The table 3.a presents the PCMCIA/Terminal and Custom Chips and the table 3.b summarizes technical characteristics of the main facturers offer.

commercial name	***RSA512***	***SCALPS***	***CY512i***	***CRIPT***	***PCC200***
manuf	Amtec	UCL	Cylink	CNET	Pijnenb.
core µP	co-µP	custom	80C31	custom	co-µP
chip area	71K gates	11.8 mm2	73 mm2	32 mm2	60 mm2
ACP area	irrelevant	irrelevant	n.c.	irrelevant	16 mm2
RAM	co-µP	128	768	ext.	co-µP
K ⇕	co-µP	0.25	8	ext.	co-µP
technology	0.5µ CMOS8L	1.5µ CMOS2ML	1.5µ EEPROM	1.2µ CMOS	1µ CMOS2ML
algorithm	Bucci	Montgomery	Massey-Omura	bit-by-bit	secret
◷	100	6	15	25	20
ROM	irrelevant	hardwired	8 flash	hardwired	hardwired
emulators	the chip itself	C+toolbox	adapter board	available	the chip itself
librairies	irrelevant	hardwired	announced	available	hardwired
N max	513	512	512	1024	1023
price/unit	n.c.	prototype	n.c.	not for sale	n.c.
patents	no	no	yes	no	no
some ∃ applications	Banking, X25	university prototype	not comm.	PCMCIA	GSM, Banking
constants	needed	none	none	none	none
number formats	redundan + msb-lsb	not comm.	lsb-msb	not comm.	msb-lsb
chip availability	n.c.	prototypes	samples	not for sale	available

Table 3a. Terminal and Custom Chips

This table summarizes the information concerning the Terminal/PCMCIA chips and the custom designs of CRYPT from CNET and SCALPS from UCL. For futher information on SCALPS (Smart-CArd for Limited Payment Systems) technology, read the complete description and performance analysis in [3].

commercial name	*ST16CF54*	*ST16KL74*	*SLE44C200*	*SLE44CR80S*	*P83C852*	*P83C858 - FAME*	*P83C855*	*MC68HC05 SC29*
manuf	SGS	SGS	Siemens	Siemens	Philips	Philips	Philips	Motor.
core μP	68HC05	68HC05	80C51	80C51	80C51	80C51	80C51	68HC05
chip area	secret	secret	24.5 mm2	24.5 mm2	22.3 mm2	unkown	31.1 mm2	27 mm2
ACP area	secret	secret	5.7 mm2	unknown	2.5 mm2	unknown	2.5 mm2	5 mm2
RAM	352	608	256	256	256	640	512	512
K ⇕	16	20	9	8	2	8	2	4
technology	1.2μ CMOS	1.2μ CMOS	1μ CMOS	0.7 μ CMOS	1.2μ CMOS	0.7 μ CMOS	1.2μ CMOS	1.2μ HCMOS
algorithm	Montgomery	Montgomery	Sedlak	Sedlak	Quisquater	Quisquater	Quisquater	Montgomery
🕒	5	5	5	5	10	8	10	5
ROM	16	20	9	17	6	20	20	13
emulators	available	available	available	available	available	available	available	available
librairies	available	available	available	available	simple	N.S.	simple	available
N max	768	1024	540	540	648	20484	1328	1024
price/unit	$ 4.5	n.c.	n.c.	n.c.	n.c.	n.c.	n.c.	$ 4.5
patents	yes	yes	yes	yes	yes	yes	yes	yes
some ∃ applications	GPK2000, Pay-TV	not comm.	CAFE, DSM-fax, CP8	not comm.	DX, Mimosa, Starcos	DX2	not comm.	French health card
constants	needed ACP-computable	needed ACP-computable	needed	needed	needed	needed	needed	needed ACP-computable
num. formats	lsb-msb	lsb-msb	msb-lsb	msb-lsb	both possible	both possible	both possible	lsb-msb
chip availability	available	announced	available	available	available	available	available	available

Table 3b. Main Manufacturers' Offer

We can observe from the information provided that the new FAME component proposes the largest RAM (640 bytes) while SGS-Thomson's chip offers a huge 16 Kbytes of EEPROM for data applications. Furthermore, the new Siemens chip S series offer very atttactive performances and may occur an extension of the product range with various memory/performance trade-offs, still well within the chip area required by ISO 7816, thanks to the new 0.7 technology.

3.3. High-level implementation

It is generally recommended to separate the cryptographic *schemes* (RSA, DSA, Rabin, GOST etc.) from the cryptographic *operations* (sign, verify, encrypt, decrypt, hash, key-exchange) by implementing in the card an « I/O buffer » into which data to be processed is written by the terminal. In this model, the following steps are executed whenever a cryptographic operation is performed by the card :

① a key file (specific to a *scheme*) is selected by a put command carrying the file ID.

② data to process (message,etc.) is written to the I/O RAM buffer by a put command.

③ a get command (specific to an *operation*) retrieves the result from the card.

Such an approach results in a simplified command-set and allows to upgrade the card without adding new command-codes. Here is a toy-example illustrating the encryption of the message « process me that » with the RSA keys contained in file 2401 followed by the signature of the message « 123 » by the DSA file 334A and a Diffie-Hellman key exchange with the keys contained in file E1F3 :

```
select file 2401                ISO exchange : 🖳 ⇦ 5
{RSA, 768, s/e/i}               /* ≅ TYPE in DOS    */
```

file selection returns the key-file type (here 768-bit RSA) and the cryptographic operations allowed with this file (s = signature and verification e = encryption and decryption, i = identification and verification, k = key exchange).

```
put data                        ISO exchange : 🖳 ⇨ 5
{« process me that »}           /* data to process */
get data : encrypt 0000         ISO exchange : 🖳 ⇦ 5
{« E32A371B908AB37 »}           /* ≅ ENCRYPT.EXE    */
```

the 0000 sent to the card (P1=P2=00) means that the result (here the ciphertext E32...) should be sent to the terminal (a non zero code would have been interpreted as a file ID where the ciphertext should be written).

```
select file 334A                ISO exchange : 🖳 ⇦ 5
{DSA, 512, s}                   /* ≅ TYPE in DOS    */
```

```
put data                              ISO exchange : 🖳⇨5
{« 123 »}                             /* data to process */
get data : sign 0000                  ISO exchange : 🖳⇦5
{« ADE603B826FDE04 »}                 /* ≅ SIGN.EXE       */
select file E1F3                      ISO exchange : 🖳⇦5
{D-H, 512, k     }                    /* ≅ TYPE in DOS    */
put data                              ISO exchange : 🖳⇨5
{« process me that »}                 /* a^x mod p        */
get data : key-exchange 2010          ISO exchange : 🖳⇦5
{« AE589EB6A564CDD »}                 /* ≅ KEY_EXCH.EXE
                                      returns a^y mod p */
```

during a key-exchange, the user must specify a destination file ID (here 2010) for the common key $a^{xy} \bmod p$ (the outside world can never access this value). This file is typically used for keying triple-DES operations.

manufacturer / card	*chip*	*public-key algorithms*	*done by software*
Bull « TB Crypt »	Siemens 44C200	RSA, DSA, DH	DES, SHA
CP8 Oberthur « TB 98 »	Siemens 44C200	RSA	DES
Datakey « Signasure »	Siemens 44C200	RSA, DSA, DH	DES, SHA
Giesecke & Devrient « Starcos PK »	Siemens 44C200	RSA	DES
Gemplus « GPK 2000 »	SGS-Th. ST16CF54	RSA, DSA, DH, Rabin, GQ	DES, SHA, MD5, GOST
McCorquodale « AMC04 »	?	RSA, DSA	DES
Oldenbourg « ODS83 »	Philips 83C852	RSA, DSA	?
PC3 « Smart-Card RSA »	SGS-Th. ST16CF54	RSA, DSA	« secret-key »
Philips TRT « DX »	Philips 83C852/8	RSA, DSA	DES
Schlumberger « Multiflex 8K »	SGS-Th. ST16CF54	RSA	DES
Setec « Setcard 5K RSA »	Siemens 44C200	RSA	DES

Table 4. Some recent applications

We incite the reader to contact directly the *card manufacturers* for further details concerning a given mask (unlike ACP designs, masks are constantly changed and improved : for instance, Siemens offers elliptic-curve functions, Gemplus offers several EEPROM options (El-Gamal [4], Schnorr [11], Rabin, GOST 34.10 etc.). Other publications [8], [9], [12], [13] or Web sites frequently provide valuable information about product evolution and specific « cookbook » optimizations.

3.4 Performance

Philips, Siemens, Thomson and Motorola will certainly dominate the mass-market during the next three years. The existing applications and prototypes (CAFE[9], the French Health Card, CP8 crypto-card, Gemplus' GPK 2000 etc..) seem to confirm this trend.

We will therefore mainly focus the remaining sections of our survey on these chips. First, a benchmark including 4 performance tests was performed. The idea was to determine precisely the computation speed without any custom optimization, i.e. using only the ressources of the chip itself.

chip	***RSA 512***	***RSA 768***	***RSA 1024***	***DSA***
ST16CF54/A	*s*=385 *c*=195 *v*=50 *k*=5,000	*c*=870		*s*=150 *v*=350
ST16CF54/B	*c*=150	*c*=*v*=185		
SC49 = SC29	*s*=500 *c*=125 *v*=35 *k*=5,600	*s*=4,480 c=1,112 *v*=168	*s*=5,600, *c*=1,499 *v*=168	*s*=114, *v*=250
SLE44CR80S	*s*=300 *c*=60 *v*=40 *k*=20,000		*s*=630, *c*=450	*s*=95 *v*=200
SLE44C200	*c*=60	*c*=*v*=271	*c*=*v*=456	*s* =92
P83C852	*s*=225 *c*=70	*c*=2,400		
P83C858	*c*=600		*s*=2,000, *c*=500	*s*=70 *v*=130

Table 5. Smart-card chips performance (in ms) for a 5MHz clock

where : *s* = signature time without CRT[10] *c* = signature time with CRT

v = verification time *k* = (on-board) key generation time

[9] Control Access For Europe; an EEC Esprit project for designing a wallet due to provide privacy and security to both issuers and users.

[10] Chinese Remainder Theorem, see [7] for theory and implementation aspects.

Performance is also closely related to the ACP's area (assuming the same technology):

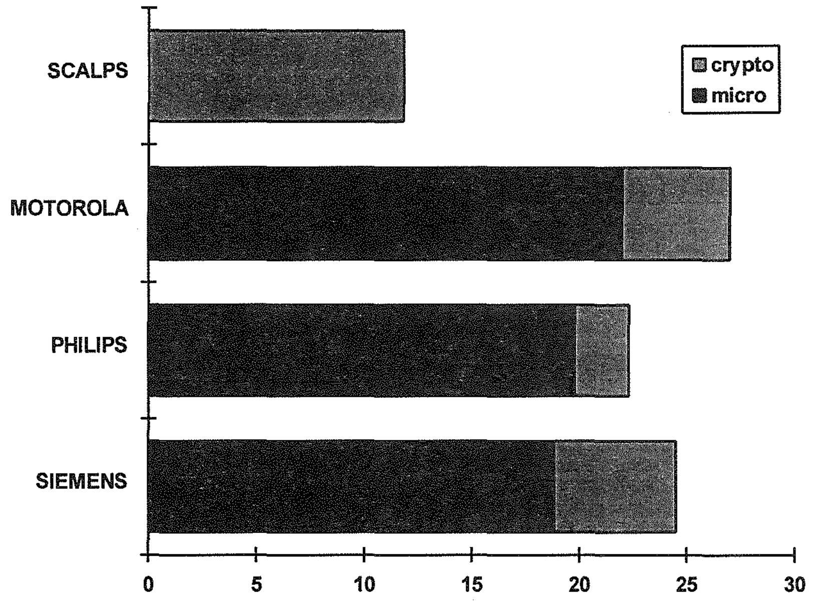


Figure 2. microprocessor area (square mm)

From our benchmarks, it appears that the speed difference between the ST16CF54A and the SLE44C200 is relatively important.

Although slower than the previous two, Philips' old design (83C852) seems the most elegant (both in terms of silicon area, algorithm and circuit simplicity).

SCALPS deserves its acronym very-well : chip size is about the half of microcontroller-based ones.

Furthermore, the figure 3 illustrates the fierce battle on the 240-512 bit battlefield and underlines the impact of size parameters and specific usage (CRT for instance) in the choice of an ACP solution.

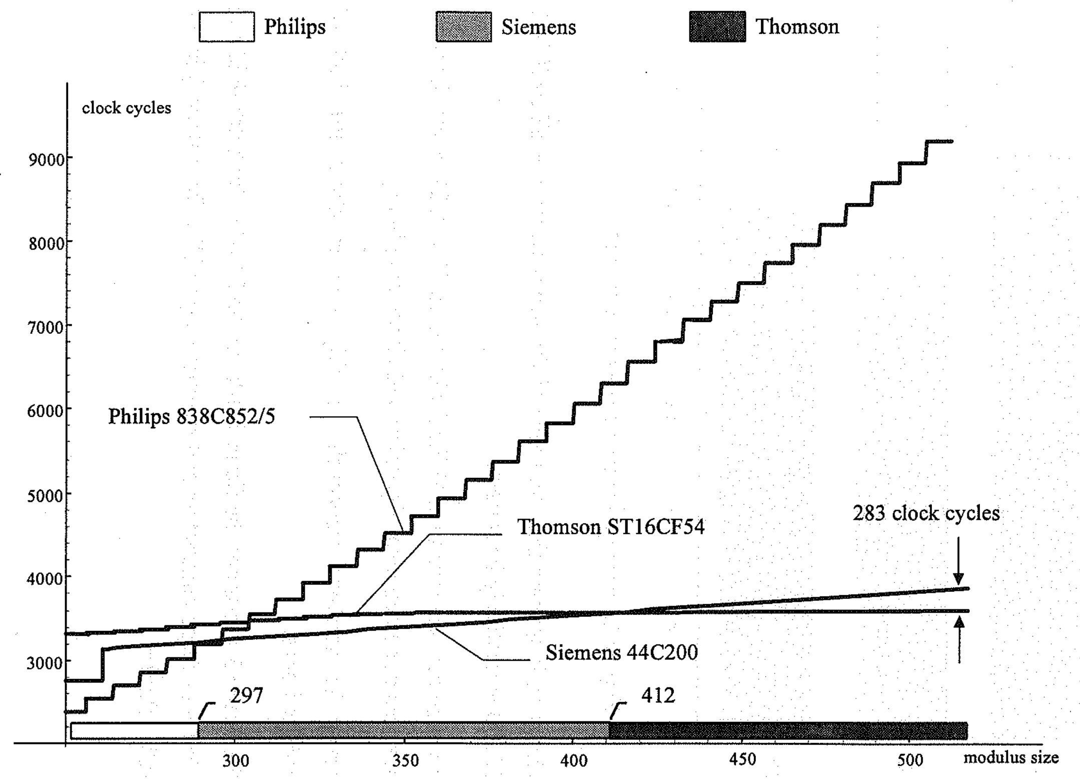


Figure 3. Smart-card ACPs : the 240-512-bit battlefield

Eventually, the figure 4 illustrates the main characteristics of the PCMCIA/Terminal chips which are flexibility for Cylink and Pijnenburg in terms of operand size and speed for Amtec.

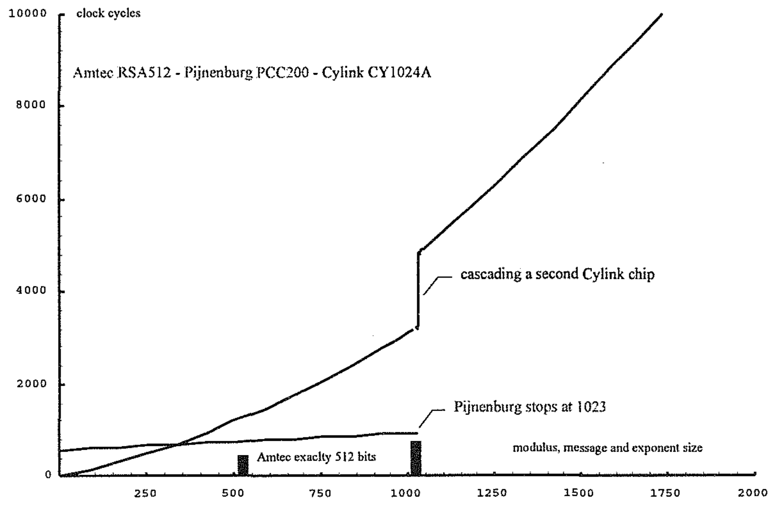


Figure 4. PCMCIA / terminal chips (clock cycles for 0.5 weight RSA)

One can observe that the most flexible (N) designs are generally slower :

chip	*Amtec (RSA512)*	*Pijnenburg (PCC200)*	*Cylink (CY1024A)*
advantage	*very fast*	*fast*	*limit is 16,384 bits*
disadvantage	*exactly 512 bits*	*limited to 1,023 bits*	*relatively slow*

Table 6. PCMCIA/Terminal chips performance

4. Conclusion

Before choosing a crypto-processor, contact as many manufacturers as possible, ask questions (are RSA/PKP patent licenses included in the deal ? will you have access to source codes ? can you modify these ? which agencies got technical details about the chip and/or your application ? etc.) and compare the answers, *consider unanswered questions and unauditable codes as gray spots and potential problem sources.*

Most manufacturers sell three or four generic products that cover together a big percentage of the market needs. Consider the possibility of purchasing and customizing such a general purpose mask. Would your application be too specific, decide exactly what's the most important : time performance, price, development ease, physical security, tools (libraries, emulators), modularity, portability (very easy when microprocessor cores are identical but relatively complicated if not), EEPROM space for dedicated applications, availability of a true random number generator etc.

5. References

[1] **P. Barrett**, *Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor*, Advances in Cryptology : CRYPTO'86 (A. M. Odlyzko ed.), LCNS 263, Springer-Verlag, pp. 311-323, 1987.

[2] **W. Diffie and M. Hellman**, *New Directions in Cryptography*, IEEE Transactions on Information Theory, vol IT-22, n°6, pp. 644-654, november 1976.

[3] **J-F. Dhem, D. Veithen and J-J Quisquater**, *SCALPS : Smart CArd for Limited Payment Systems*, IEEE Micro, Public Key Security Systems issue, vol 16, n°3, pp. 42-51, june 1996.

[4] **T. El-Gamal**, *A public-key cryptosystem and a signature scheme based on discrete logarithms*, IEEE TIT, vol. IT-31:4, pp 469-472, 1985.

[5] **A. Fiat and A. Shamir**, *How to prove yourself: Practical solutions to identification and signature problems*, Advances in Cryptology : CRYPTO'86 (A. M. Odlyzko ed.), LCNS 263, Springer-Verlag, pp. 181-187, 1987.

[6] FIPS PUB 186, February 1, 1993, *Digital Signature Standard.*

[7] **D. Knuth**, The Art of Computer Programming, vol. 2, Seminumerical Algorithms, pp. 248-250, 1969.

[8] **P. Montgomery**, *Modular multiplication without trial division*, Mathematics of Computation., vol. 44(170), pp. 519-521, 1985.

[9] **J. Omura**, *A Public Key Cell Design for Smart Card Chips*, IT Workshop, Hawaii, USA, November 27-30, pp. 983-985.

[10] **R. Rivest, A. Shamir and L. Adleman**, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, vol. 21, n°2, p.120-126, February 1978.

[11] **C. Schnorr**, *Efficient identification and signatures for smart-cards*, Advances in Cryptology: EUROCRYPT'89 (G. Brassard ed.), LNCS 435, Springer-Verlag, pp. 239-252, 1990.

[12] **H. Sedlak**, *The RSA cryptographic Processor : The First High Speed One-Chip Solution*, Advances in Cryptology : EUROCRYPT'87 (C. Pomerance ed.), LNCS 293, Springer-Verlag, pp. 95-105, 1988.

[13] **D. de Waleffe and J.J. Quisquater**, *CORSAIR, a smart card for public-key cryptosystems*, Advances in Cryptology : CRYPTO'90 (A. Menezes and S. Vanstone ed.), LNCS 537, Springer-Verlag, pp. 503-513, 1990.

FAME: A 3rd Generation Coprocessor for Optimising Public Key Cryptosystems in Smart Card Applications

Ronald Ferreira (1),
Ralf Malzahn (2),
Peter Marissen (1),
Jean-Jacques Quisquater (3),
Thomas Wille (2).

(1) Philips Smart Cards & Systems
30 Rue Boussingault, 75013 Paris, France
ferreir5@frccmail.snads.philips.nl
marisse1@frccmail.snads.philips.nl
(2) Philips Semiconductors
P.O. Box 540240, D-22502 Hamburg, Germany
(3) Math RiZK
Avenue des Canards 3, B-1640 Rhode-Saint-Genèse, Belgium
Quisquater@dice.ucl.ac.be

Abstract

Interest in implementing public key cryptosystems (PKCs) in Smart Cards is ever increasing as evidenced by the growing number of Smart Card products and Smart Card applications based on PKCs. This paper discusses the motivation for this interest, and analyses the computational cost of modular exponentiation in different Smart Card architectures currently being exploited by typical Smart Card environments. An efficient architecture for implementing PKCs in third generation Smart Cards taking into account the already implemented first generation PKC based Smart Cards is next described. The advantages of this architecture (FAME) and its adaptability for evolution are pointed out.

Key words: smart card, coprocessor, modular arithmetic, public key cryptosystem.

1 Introduction and Background

The Smart Card has become a de facto tool in providing security in today's ever increasingly computerised society. The security requirements (typically non-repudiable digital signatures and key separation), essential to many Smart Card applications, both mass consumer and professional, can best be provided by Public Key Cryptosystems (PKCs). For example in many EDI applications non-repudiable digital signatures, which can only really be provided by PKCs, are the basis of the secure communication and trust between financial institutions and their corporate customers.

The distributed nature of mass consumer smart card applications also favour the use of PKCs. A typical example is the electronic purse where the distributed and off-line characteristics of the application require the storing of sensitive key information in potentially unsafe physical locations on a large scale. Basing such an application on a symmetric secret key cryptosystem means replicating global system keys in all physical locations (terminals) if unrestricted electronic purse transactions are to be permitted. This exposure of global keys makes the system potentially vulnerable, because compromise of a single physical location (terminal) could in principle allow cloning of any electronic purse in the system. This vulnerability can be reduced by using various key partitioning schemes at the expense of increased key management and protocol complexity without attaining the perfect key separation achievable using PKCs.

Additionally, many PKCs (e.g. RSA, DSS, Zero Knowledge schemes such as FS, GQ) have reached maturity, whilst simultaneously Smart Card memory and processing resources have increased sufficiently to allow their practical implementation in many applications. The use of PKCs in Smart Card applications is ever growing and irreversible, and as the level of sophistication increases, the demand for increased Smart Card performance keeps pace. A key factor in meeting this demand is the implementation of optimised techniques for accelerating PKC performance, particularly those based on specialised coprocessors.

Section 2 gives a brief history of Smart Cards, criteria for efficient PKC architecture oriented to Smart Card implementation and basic algorithms. Section 3 describes a new architecture called FAME (Fast Accelerator for Modular Exponentiation) for PKCs based on a coprocessor approach optimised for implementation in Smart Cards, together with the key hardware features in a specific implementation, section 4 discusses the performance attainable for different PKCs, and section 5 evolution and adaptability. We conclude by highlighting the advantages of the FAME design.

2 Lessons From the Current Smart Card Architectures

2.1 A Brief History of Smart Cards

Technological barriers impose limitations on cryptographic algorithms (specifically PKCs), which can be implemented in Smart Cards due to limited resources of ROM/RAM/EEPROM and specific instruction sets which may be required for adequate performance. Typically, for high reliability and yield, experience has demonstrated that a Smart Card chip should be less than 25 mm^2.

The first Smart Cards in the early 1980's could only implement very restricted algorithms such as TELEPASS coded in a few hundred bytes and requiring very limited RAM. However, such algorithms did not gain universal acceptance. The successful Smart Card implementation by Philips of DES in 1985 was a breakthrough which expanded the market for Smart Card security applications.

However DES, as already pointed out, raises security problems in some application areas, although its cost effectiveness, high performance, and security attributes, will ensure that it remains a popular algorithm in appropriate Smart Card applications.

In earlier generations of Smart Cards, implementation of PKCs, particularly those based on modular exponentiation encountered resource and performance limitations. Typically, a 512 * 512 bit modular multiplication executes in around 50 to 100 milliseconds in a Smart Card depending on the CPU core and clock speed available. Thus generation of an RSA signature (see [19]) with a 512 bit key length takes around 40 to 80 seconds. Implementation of the Chinese Remainder Theorem (see [5]) would reduce this to around 10 to 20 seconds, unacceptable for real time applications using Smart Cards (ignoring moduli of larger sizes).

From 1985-1991 specific variations of PKCs e.g. the zero knowledge protocols such as Fiat/Shamir (see [7]) or Guillou-Quisquater (see [8], [9]), invented with Smart Card applications in mind, were implemented, also special PKC like protocols (viz. David Chaum), as these could yield acceptable performance.

Today, progress in factorisation and the discrete log will impose moduli of 640, 768, 1024 bits or even larger. Most current Smart Cards are not particularly adapted to handle large moduli due to memory limitations, low performance and/or lack of flexibility.

2.2 Current Hardware Architectures for PKCs

The first Smart Card capable of executing PKCs with acceptable performance for real time applications (e.g. RSA signature with 512 bit key length in ≤ 0.5 second) was commercialised by Philips in 1991 (CORSAIR, see [15]). This can be considered a first generation Smart Card for PKCs.

The new architecture included an autonomous arithmetic unit optimised for modular multiplication (a 25-50 improvement factor over straightforward software implementations). Simultaneously RAM size was increased to 256 bytes, about the minimum necessary to achieve acceptable performance. In 1.2 μ technology the arithmetic cell occupied about 2.5 mm^2 and the complete Smart Card chip about 22 mm^2. The architecture seems flexible but the technology (1.2 μ CMOS) is not sufficient to implement the hardware resources (arithmetic operations, RAM, registers, etc.) necessary to achieve high performance with large moduli.

Since then several second generation Smart Card compatible chips comprising hardware oriented to exploiting modular exponentiation have been commercialised. More advanced technology (≤ 1 μ) permits implementation of larger hardware resources and coprocessors dedicated to efficiently executing modular exponentiation. Performance, using a 512 bit key length RSA signature as a benchmark, is claimed to be in a range of 50 to 100 milliseconds. However, second generation chips suffer from disadvantages (probably due to technological constraints) that might be eliminated in later versions. The principal disadvantages appear to be the inability to efficiently handle large key lengths (typically ≥ 1024 bits) and lack of flexibility, as key length increases performance degrades more rapidly than generally accepted theory predicts (except for specific values).

Other current projects not yet commercialised are based on a RISC architecture. Another approach was used by SCALPS (by UCL, Louvain-la-Neuve, Belgium): the design is a dedicated processor for PKCs with some instructions added for the relevant application (electronic money).

2.3 Criteria for a Good Cryptosystem Implementation

The third generation Smart Card chip architecture (somewhat arbitrarily defined to differentiate it from current chips) to be described aims to overcome these disadvantages. The principal criteria for a third generation chip might be summarised as follows:

- Flexibility: the architecture should be general enough to efficiently execute any PKC based on modular exponentiation, using most algorithms with variable key length, without degrading performance more rapidly than the bounds imposed by the hardware limitations. This requires the use of a common RAM (the random-access aspect is very important).
- RAM should be sufficiently general and large to avoid too frequent interlaced multiplication followed by reduction, at least for some optimum benchmark key length.
- Optimum use of the hardware resources is necessary to obtain the maximum from the available technology to avoid potential performance degrading characteristics such as padding, truncation, and to minimise the degradation due to normalisation and/or corrections, whatever the technique chosen.
- The coprocessor should be ≤ 10 % of the total silicon area of a Smart Card to minimise cost and maximise reliability and yield.
- The different busses should be optimised (i.e. no additional silicon penalty) by innovative exploitation of the technology. This point is important because it should be possible to use the coprocessor without idle time and the busses should be designed accordingly.
- Preloading of parameters must be possible to compensate for unavoidable under-utilisation of busses.
- Complete and independent control for the coprocessor should be possible, via fast and dedicated RAM access, independent access to both ROM and EEPROM, the provision of local pointers for fast context switching and asynchronous control with respect to the CPU.

- Additionally the coprocessor should be able to process a basic multiplication (e.g. 32 * 32) very rapidly in a minimum of clock cycles, should be fully occupied and should be user-friendly (require a minimum of commands to perform a complete modular multiplication).

Clearly, fulfilling all of these criteria requires the modelling of different architectures to find the optimum. The flexible architecture of CORSAIR (see [15]) is a good basis if we add the improvements possible due to technology advances. Such an improved architecture will be described.

2.4 Algorithms

There is an evident relation between the algorithm(s) in mind for the computation of the modular multiplication and the coprocessor architecture.

The following operations are to be considered:

- Multiplication of two large integers:
  While there are many ideas on how to perform the multiplication only the classical ones are practically effective. Maybe for integers with more than 512 bits it is possible to use the acceleration of Karatsuba. The gain in performance is not so significant. Other ideas (Brent, etc., see Knuth [11]) need more study to be effectively applied to the Smart Card. The only useful practical idea seems to be to accelerate the operation of *squaring* relative to the general multiplication (already in use for software implementations of PKCs in Smart Cards).
- Modular reduction with large integers:
  There are many methods to perform modular reduction. These are based on the use of:
  → precomputation (see [4]);
  → pre-processing and post-processing of the numbers in use (to avoid the computation of approximate partial quotients);
  → operations interleaved or not with the multiplications;
  → computation starting from the LSB or from the MSB of some operand;
  → division with correction, without correction or accumulation and correction at the end.

The two main categories of methods are:
1. Method of Knuth (see [11], Barrett [2], Quisquater [15]): computation starting from the MSB.
2. Method of Montgomery (see [12], Arazi [1]): computation starting from the LSB.

The next section will describe an architecture permitting efficient implementations of the two categories.

3 A new Architecture

3.1 Algorithms

The multiplication algorithm is simple and self evident. Let us denote by X and Y, the two numbers to be multiplied. The needed basic step is well described by:

$$R \leftarrow Carry + (Y \times x_i) \qquad (1)$$

where R is the register for storing the result, the *Carry* is some intermediate short result (generally between 1 bit and 32 bits) and the number x_i is the i-th word ("digit") of X.

The reduction algorithm will use the following basic step:

$$R \leftarrow Carry + Z + (Y \times x_i) \qquad (2)$$

where Z is the current value of the accumulated partial result.

The use of computation starting from the MSB or the LSB for the reduction is only a question of incrementing or decreasing some pointers. Both can be implemented.

The coprocessor must be *smart* enough to implement efficiently many of the methods for computing the partial quotients needed by the best methods in use (including Montgomery see [12]).

The conclusion of this section is that the chosen architecture should not be dedicated to one method of computing the modular multiplication: in fact many methods should be possible. This ensures compatibility with existing software, tests and technical literature. However, not all methods can be optimised with equal efficiency in a specific implementation of the architecture. In this architecture we have chosen to optimise the Quisquater method. This method implies the use of normalised moduli, in this specific implementation the 64 MSBits of the modulus must be equal to '1'. The reason for using normalised moduli is that with this particular method modular multiplication can be performed efficiently. If we have an n bit normalised modulus and we want to reduce a number, say a, with a length of $n+m$ bits, then we multiply the m bit overflow with the modulus and subtract it from a. Notice that since the numerical value of a normalised modulus is 'almost' 2^n, the result of the subtraction will be very close to the value of the modulus. By choosing sufficiently many bits equal to '1', one can assure that at most one more subtraction of the modulus is sufficient to complete the reduction.

If free format moduli are used, then the normalisation and subsequent denormalisation can be done on chip. This adds some overhead which varies depending on the type of cryptosystem, specific implementation and size of parameters.

3.2 Hardware

In this section we describe the architecture of the coprocessor FAME (Fast Accelerator for Modular Exponentiation, [16] and [17]) and its interconnection within a general Smart Card architecture.

The following choices were made after a careful analysis of several architectures:

- The CPU and the coprocessor are two *separate entities* with very few interactions: the two processors can work in parallel.
- Sharing of the storage facilities (RAM, EEPROM, ROM). In fact, the RAM is separated into two parts, one accessible by the coprocessor and the CPU (XRAM: the extended RAM) and one only accessible by the CPU (IRAM: internal RAM). This division facilitates the solution of access conflicts. The XRAM comprises 512 bytes and is used by the coprocessor to execute basically modular operations. The IRAM comprises 128 bytes program memory plus 128 bytes SFRs (Special Function Registers). The interface between the coprocessor, CPU and the various memories is illustrated in figure 1.

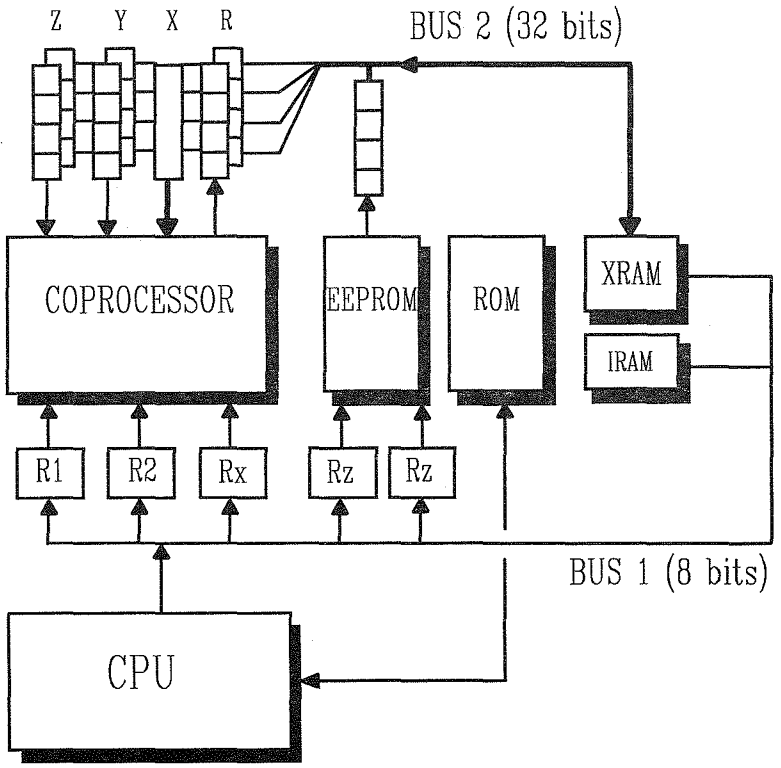


Figure 1: the interface between the coprocessor and the memories.

The analysis of the current technological possibilities led us to consider the implementation of an 8 × 32 bit multiplier as the optimum architecture for 0.7 μ technology.

Due to the choice of the multiplier, the equation (2) will be performed by the repetition of the following basic step:

$$r_j \leftarrow Carry + z_j + (y_j \times x_i) \qquad (3)$$

There are two possibilities:

x_i has 8 bits and thus y_j has 32 bits; this solution needs a large bus because, for each clock cycle, there is a transfer of 32 bits for each z_j, y_j, and r_j. The width of the bus is thus at least 32+32+32 bits.

x_i has 32 bits and thus y_j has 8 bits. The width of the bus required is now only 24 bits, i.e. 8+8+8 bits.

The two solutions need the same multiplier and two adders (not in the same order!): the difference will only be for the *Carry* (8 bits for the first solution and 32 bits for the second).

Both coprocessor architectures are compact: the bus itself is not a problem but the second solution is very well suited for a small interface (second bus with 24 bits). In fact, using a bus of only 24 bits is not very efficient because the bus is already fully used for the current operation: there is no cycle time available for preloading new values for the next operation (better use of the shadow registers *Z'*, *Y'*). A bus of 32 bits is better, permitting the desired preloading. Figure 2 shows the architecture of the 8*32 multiplier which is the core of the FAME coprocessor.

The following example shows the use of the 32 bit bus:

→ **cycle 1:** load *X* (32 bits),
→ **cycle 2:** load *Z'* (32 bits),
→ **cycle 3:** load *Y'* (32 bits),
→ start of operation, copy *Z'* into *Z* and *Y'* into *Y*.
→ **cycle 4:** no need of loading; output R_0,
→ **cycle 5:** load *Z'* (32 bits); output R_1,
→ **cycle 6:** load *Y'* (32 bits); output R_2,
→ **cycle 7:** output R_3, output *R'*,
→ **cycle 8:** same as cycle 4.
→ ...

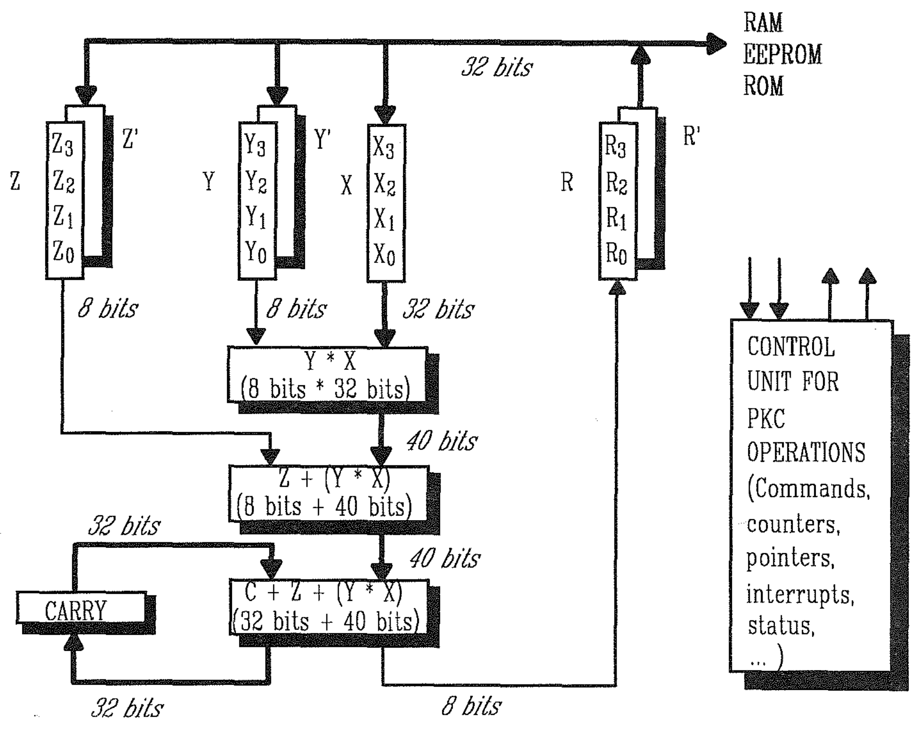


Figure 2: a configuration for the multiplier 8 × 32 bits and two relevant adders.

Except for the initialisation and termination phases, the basic algorithm is the repetitive use of the cycles 4 to 7. In this implementation the bus is not used one cycle out of 4; this time slot can be used for other purposes, e.g. preloading. Clearly the shadow registers are required as buffers or pipelines between the 32 bit bus and the multiplier unit to achieve proper synchronisation. For example, *Y'* and *Z'* are each loaded with 32 bits once every 4 cycles whilst 8 bits of *Z* (Z_0 to Z_3) and *Y* (Y_0 to Y_3) are input to the multiplier unit during every cycle and therefore the contents of *Y'* and *Z'* are copied to *Y* and *Z* respectively once every 4 cycles, when the previous Y_0 to Y_3 and Z_0 to Z_3 have been shifted out. Without the shadow registers to act as buffers the inputs from the bus to *Y* and *Z* would need to be held steady during 4 cycles which would require a 96 bit bus (a similar argument applies to *R* and *R'*. In this case R_0 to R_3 (8 bits each) are loaded during 4 cycles and the 32 bit result *R'* output on the bus every fourth cycle.

This architecture is well suited for most multiplication and reduction algorithms: if some negative operations are needed, the availability of the 2's complement form of some operands will be useful. Many efficient algorithms need only one form, either negative or positive: thus there is no penalty for the needed number of bytes in RAM or EEPROM.

3.3 Hardware Features for Optimising FAME

In this section we summarise the principal hardware features implemented to optimise the performance of FAME.

- Optimisation of modular squaring gives a considerable improvement in performance with respect to modular multiplications.
- Hardware programming (cabled logic) of specific FAME modes to execute macro functions such as:

  → Multiplication only (pure multiplication, multiplication with addition, multiplication with use of last value, etc.).

  → Multiplication with reduction

  → Squaring with reduction

  → Specific FAME mode multiplication with reduction

  → Binary operations with one operand fixed (logical AND, OR, XOR, addition).

  → Binary operations with both operands variable (logical AND, OR, XOR, addition).
- Flexibility to specify combination of operand sources destinations, e.g. all operands from RAM or RAM/EEPROM or all from EEPROM.
- Several sets of context switching registers.

Typical modular operations, such as modular multiplication or modular squaring, are implemented in hardware as high level macrofunctions. To execute a macrofuction, it is sufficient to place the operands (typically 512 bits) in XRAM, specify the addresses and lengths of these operands in Special Function Registers (SFRs), and give a start command to the coprocessor. The coprocessor then executes the required macrofunction independently of the CPU. The CPU can execute some other task in parallel, such as initialisation of the SFRs for the next computation to be executed by the coprocessor. When the coprocessor has terminated execution it returns control to the CPU by setting a flag. The CPU can now retrieve the results from XRAM.

In normal circumstances the operands should be placed in XRAM, although an operand fetch from EEPROM is also supported, to allow very large key sizes (e.g. 2048 bits or even more) for specific implementations.

FAME also provides binary operations (addition, logical AND/OR/XOR) to accelerate the programming and performance of symmetric cryptosystems such as DES or hash functions such as SHA.

4 Performance of FAME

Table 1 shows typical performance figures for the FAME coprocessor for RSA and DSS [13] based on silicon level emulations of specific implementations in a Smart Card component. The component is the Smart Card controller P83C858 (20 KBytes ROM, 640 Bytes RAM, 8 KBytes EEPROM) which will be available in the third quarter of 1996. We have not attempted to make detailed comparisons with other PKC chips due to the difficulty of obtaining accurate and reliable figures.

Crypto function	Modulus (bits)	Exponent (bits)	Computation time (5 MHz external clock)
RSA Signature (classical)*	512	512	156 ms
RSA Signature (CRT)**	512	512	47.3 ms
RSA Signature (optimised)**	512	512	25 ms
RSA Signature (classical)*	768	768	520 ms
RSA Signature (CRT)**	768	768	157 ms
RSA Signature (optimised)**	768	768	83 ms
RSA Signature (classical)*	1024	1024	1.08 seconds
RSA Signature (CRT)**	1024	1024	305 ms
RSA Signature (optimised)**	1024	1024	93 ms
RSA Signature (classical)*	2048	2048	18.4 seconds
RSA Signature (CRT)**	2048	2048	2.18 seconds
RSA Signature (optimised)**	2048	2048	661 ms
DSS signature generation*	512/160	160	75 ms
DSS signature verification*	512/160	160	130 ms

* Measured times

** Calculated times

Table 1: FAME (P83C858) crypto-performance for RSA and DSS using optimised macrofunctions implemented with modular multiplication starting from the MSB (Quisquater method).

- Only computation time for crypto-functions are taken into account.
- CRT signature is computed with 2 prime factors.
- "Optimised" signature computed under specific conditions, but without any loss in security based on best factoring algorithms known today.
- Classical RSA signature for 2048 bits gives degraded performance due to lack of sufficient RAM.
- Signature verification with public exponent of 3 (2 bits) in all cases gives negligible computation time (< 2 ms) except for 2048 bits where the computation time will be 5 ms - 10 ms.

- The above computation implies the use of normalised parameters for the implemented algorithm for RSA (i.e. 64 most significant bits set to '1' for the Quisquater method).
- Very large key sizes can be handled.
- The DSS performance figures are based on free format parameters, normalisation and denormalisation is performed on chip.

5 Evolution and Adaptability

The basic algorithm and consequently the architecture for performing modular exponentiation for different key sizes is very efficient and flexible, i.e. performance degradation is due only to limitations of hardware. This is evident from the performance figures.

The obvious way to increase performance is to take advantage of technological advances, e.g. going to 0.5 µ technology. This will enable the hardware resources to be increased, for example the size of the XRAM could be increased, or the arithmetic unit could be expanded from 8*32 to 16*32 bits. Additionally the higher speed potential of more advanced technology can be used to advantage to increase the internal clock speed. A combination of these factors will allow a PKC of given key size to be executed more efficiently, PKCs with large key sizes to be executed in a given time, or both. Thus it will not be necessary to fundamentally modify the basic architecture of FAME in the future. Moreover some additional hardware could be added to very significantly speed up symmetric cryptosystems (some basic operations for this purpose are already incorporated in FAME). Such an architecture would be upwards compatible i.e. crypto applications implemented on the current architecture could be easily ported to the evolved architecture. Also the modular architecture of FAME allows easy integration into different Smart Card configurations, i.e. FAME will be used in the PHILIPS family of Smart Card components dedicated to different application areas.

6 Conclusion

Evolution of an already established architecture to take advantage of technological progress gives the maximum trade-off in performance/flexibility/compatibility whilst reducing to a minimum the risk factors involved in developing a new product. This approach has been substantiated in practice because the FAME architecture gives excellent performance for various PKCs, and can support a range of key lengths without any performance degradation other than that predicted by theory. The evolutionary approach was justified as we did not find any problems with FAME which worked without errors from the very beginning.

References

[1] B. Arazi, *Variations on the Montgomery Theme*, private paper.

[2] P. Barett, *Implementing the RSA public key encryption algorithm on a standard digital signal processor*, Proceedings of CRYPTO '86. Springer-Verlag.

[3] P. Béguin and J.-J. Quisquater, *Secure acceleration of DSS signatures using insecure server*, *in* Proceedings of Asiacrypt '94.

[4] E. Brickell, D. M. Gordon, K. S. McCurley, D. Wilson, *Fast exponentiation with precomputation*, *in* Advances in Cryptology, Proc. of EUROCRYPT '92, Lecture Notes in Computer Science, vol. 658, Springer-Verlag, 1993, pp. 200-207.

[5] C. Couvreur and J.-J. Quisquater, *Fast decipherment algorithm for RSA public key cryptosystem*, Electronic Letters, Vol. 18, Oct. 1982.

[6] W. Diffie and M. E. Hellman: *New directions in cryptography*, IEEE Trans. Inform. Theory, vol. IT-22, 1976, pp. 644-654.

[7] A. Fiat and A. Shamir, *How to prove yourself: Practical solutions to identification and signature problems*, Proc. of CRYPTO '88.

[8] L. C. Guillou and J.-J. Quisquater, *A practical zero-knowledge protocol fitted to security microprocessors minimising both transmission and memory*, Proc. of EUROCRYPT '88, Lecture Notes in Computer Science, vol. 330, Springer-Verlag, 1989, pp. 123-128.

[9] L. C. Guillou and J.-J. Quisquater, *A "paradoxical" identity-based signature scheme resulting from zero-knowledge*, Proc. of CRYPTO '88, Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 216-231.

[10] L. C. Guillou, M. Ugon and J.-J. Quisquater, *The Smart Card: A Standardised Security Device Dedicated to Public Cryptology*, *in* Gus J. Simmons (Editor) Contemporary cryptology. The science of information integrity, IEEE Press, 1992, pp. 561-613.

[11] D. Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, Vol. 2, 1981.

[12] P. L. Montgomery, *Modular multiplication without trial division*, Math. of Computation, Vol.44, 1985.

[13] P. Nguyen, *Une Implémentation de DSS sur Carte à Puce*, Rapport de stage,

June 1996, Université Denis Diderot, Ecole Normale Supérieure de Lyon.

[14] NIST: FIPS 186 for Digital Signature Standard (DSS).

[15] Philips Semiconductor Microcontroller Products, *83C852 product specification.*

[16] Philips Semiconductors. *Objective Specification: P83C858, Secured 8-bit Smart Card Controller.* Version 1, 1995.

[17] Philips Semiconductors. *Objective Specification: FAME, Fast Accelerator for Modular Exponentiation for integration in Smart Card Controllers.* Version 1, 1995.

[18] Quisquater, J.-J., De Soete, M.: *Speeding up smart card RSA computation with insecure coprocessors, in* Proceedings of Smart Cards 2000 (1989) pp. 191-197.

[19] R. Rivest, A. Shamir and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM, vol. 21, pp. 120-126, 1978.

Lossless Compression Algorithms for Smart Cards: A Progress Report*

J.-F. Dhem, J.-J. Quisquater and R. Lecat
Email:{Dhem,Quisquater}@dice.ucl.ac.be
URL: http://www.dice.ucl.ac.be/crypto/

Université catholique de Louvain,
UCL Crypto Group,
Microelectronics Laboratory, Place du Levant 3,
B-1348 Louvain-la-Neuve, Belgium.

Abstract. We first describe how data compression can be suitable for the smart card context. After looking at the different models of compressors, we choose the best suited for smart cards at the present time: the static lossless compressors mainly the Huffman and the Arithmetic ones. The static Huffman algorithm we implemented and the memory size needed for a smart card implementation are detailed. Finally, we present the first experimental results we obtain on some typical files that permit to conclude that our compressor is implementable in smart cards. This paper opens the way to a new approach of data compression algorithms for small files and using very small resources. This work is still in progress.

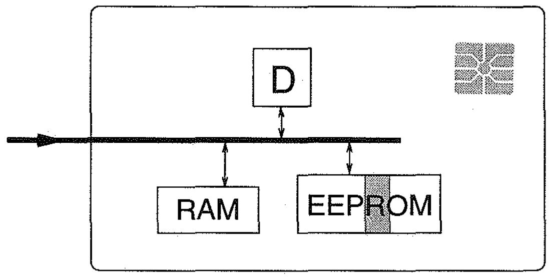
Keywords: Static Lossless Compression, Smart Cards, Huffman Coding, Arithmetic Coding.

1 Introduction

The data compression has never really been studied for some uses in the smart card context. The size of the memory and of the files to compress are so little that nearly all the existing compressors are inadequate. They regularly use huge tables with a lot of memory and are well suited for large files (typically greater than 10 Kbytes) with no memory constraints. Our goal was not to find a new compression algorithm with the same efficiency as the better ones used in other applications but to have a good one using small amount of memory and able to decompress small files. In a second step, we studied the ability to compress files in smart cards. The main part of the compressor/decompressor code is the conversion table that permits to change the data (character by character, or block of characters by block of characters) in relation to its statistics.
Several implementations in smart cards are possible depending on the needs in data compression:

* This work was performed at UCL in part under the framework of the CASCADE EP8670 project founded by the European Community.

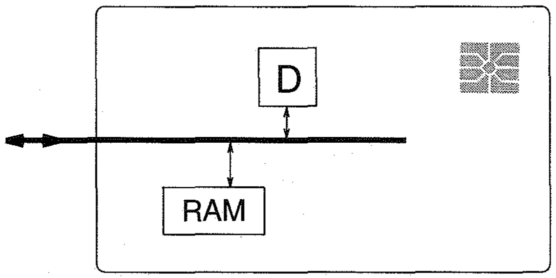
1.1 Decompressor alone, used EEPROM:



It is the basic and simplest case. The compression may be executed outside the card without any constraint on memory or speed. In the smart card context, two things may require the use of compression: the lack of memory in the smart card and the need to accelerate the slow communication between the card and the outside (often limited to 9600 bits/s).

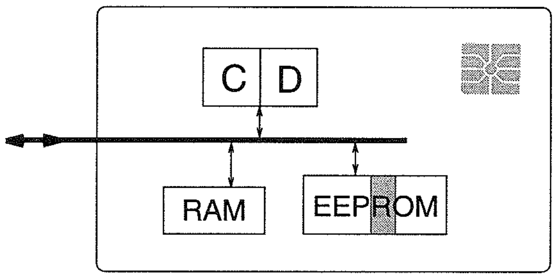
An important variant is sometimes useful. With some large files, we may want only to access to a small part of it without decompressing everything.

1.2 Decompressor alone, no used EEPROM:



This case is typical for the decompression on the fly. Either we want to use the smart card to decompress some external file (with also perhaps doing at the same time some computation on the file like cryptographic signature verification) and to resend the decompressed file directly outside or to decompress the file to be used directly by the smart card CPU without any need of recovering. The decompression must work quickly on the fly avoiding to put some intermediate results in EEPROM.

1.3 Compressor and decompressor with EEPROM:



This is the more flexible case but also the one using more memory as we also include the compressor.

2 Algorithms Options

Different options were possible:

2.1 Losing or lossless:

For the moment, we focus our attention on data like text, numerical data or assembly code. For such data, we need to exactly recover the initial values. We thus choose the lossless compressor.
Compression of data like images or sound will thus be not efficient. These data need algorithms like JPEG, quantization vector or more sophisticated lossless algorithms like JBIG. Future developments of smart card compression algorithms must nevertheless not forget this type of compression.

2.2 Dynamic or static:

A compressor uses a character or block of character's conversion table. The dynamic algorithms progressively build the table themselves when they read the data. On the other side, the static ones only need to have the table stored in memory before the compression.
The common dynamic algorithms are based on:

- *Ziv-Lempel:* Finds the same sufficiently repeated identical sequences in the text and codes them in a shorter way.
- *Run-length:* Computes the number of identical consecutive characters and only stores in memory the character and its associate number.
- *Dynamic Arithmetic and Huffman algorithms:* Dynamic version of the static algorithms is described further in this paper.

The dynamic algorithms have three drawbacks:

- The compressor must change the table at each read character. Thus, the table cannot be stored in EEPROM. We will see later that it is not possible to put the whole table in RAM.
- The algorithm's code source is bigger than with a static implementation.
- They cannot be used for decompression on the fly and for decompression of only one part of a compressed file.

Thereby, we choose to use static algorithms with a fixed known table before the beginning of the compression. This table (statistics) will be computed by a third algorithm, called "modeler". In any case, this algorithm will be placed outside the smart card. The well known simplest lossless static algorithms implementable for smart cards are the Arithmetic and the Huffman compressors described further.

2.3 Size of the conversion table:

Some algorithms use larger tables for the same amount of information. This argument may be here very important.
For one algorithm, the larger will be the table, the better will be the compression rate. In our case, the size of this table and the compromise between this size and the one of the compressed file are very critical. Furthermore several options are also possible:

- *One table linked to each file:* it allows a very good compression factor with a sufficiently large table but since each file has its own table, the size of each table must be small.
- *One table for several files:* the gain in space is evident but, to have a good compression factor, the table must be initially build taking into account all the compressed files with this table. A file with a very different statistics than the one of the table will be very badly compressed.
- *One table for several files with one very small one for each file:* this is an mixing of the two preceding ones. It is heavy to implement.
- *Table included in the compressed files:* the preceding solutions may be seen with the tables not separated of the compressed files but included in their core. It does not give significant improvements and it is also heavy to implement.

Our developments are based around the two first solutions.

2.4 Resistance to noise attack:

Some algorithms have a catastrophic behavior with regard to noise. This argument is irrelevant for us.

3 Types of compression

To determine whether a file is compressed or not, we choose to put a 24-bit word at the beginning of the stream with the 8 first bits determining the type of compression (with or without compression, algorithm 1 using table x, algorithm 1 using table y,...). The 16 last bits give the size of the file.

We shall only consider fixed size characters to variables for compression and the invert for decompression.

The recurrences in the files may be treated in different manners:

- Using only the statistics of each character in the file,
- Taking into account several linked identical characters (AAAA ... BBBB BBB),
- Taking into account several linked identical patterns (ABCABCABC),
- Taking into account identical patterns randomly placed in the file (...AB ......AB .. AB ....),
- a mixing of all the preceding methods.

As the code and the tables must be limited, we only focus our study on the first basic compression option (based only on the statistics), the other ones are more linked to dynamic algorithms.

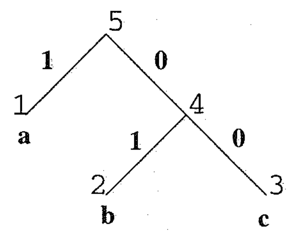
4 Huffman Algorithm

Proposed by Huffman in the early 50's [3], this algorithm uses a binary tree giving the binary representation of the characters used in the chosen alphabet to represent a compressed file. In what follows, n represents the number of characters in this alphabet.

Creation of the table by the modeler:

The first step is to build the tree using an external modeler. For that, we consider the list of the character's probabilities like a list of one node tree. The modeler takes the two smallest probabilities in the list, makes the corresponding nodes, generates an intermediate node (the parent) and labels the link from the father to the child having the smallest probability with a '0' and to the other child with a '1'. In the list of probabilities, the father now replaces its children. It is equal to the sum of the probabilities of its children. This process is repeated until having only one tree in the list. The table for the compressor/decompressor is simply a record of the tree. Practically, this table is composed of $2n-1$ lines representing the nodes of the tree. Each line contains three data: both children and the father. We can see that $2n$ lines contain only one significant data: the father. Indeed, the n leaves (corresponding to the initial characters) do not have any child. To facilitate the implementation of the compressor, the n first lines will be the n leaves.

Example: We take a three character alphabet 'a', 'b', 'c' with respectively the probabilities $p_a = 0.5$, $p_b = 0.25$, $p_c = 0.25$.
The modeler builds the tree:



and stores it using a representation as in the following table:

	Node	Branch '1'	Branch '0'	father
'a'	1	-	-	5
'b'	2	-	-	4
'c'	3	-	-	4
	4	2	3	5
	5	1	4	-

□

Compression:
To compress a character, we simply need to cover the tree from the leaf representing the character to the root. It gives the binary representation of the new character from its LSB to the MSB.

Rebuilding a character each time is a waste of time. Indeed, it will be better for speed and for the simplicity of the compressor itself to simply memorize a table giving directly the correspondence between the character to compress and its new representation. In fact, this point of view is very bad as the table to memorize may be far bigger than with our implementation or very difficult to handle. We must not forget that the new representation of a character may take up to n bits!

Example (cont.): Let us compress the text "abca". To compress 'b', for example, first we go to node 4. At this point we see that the first bit is a '1' as the branch '1' refers to the calling node 2. We then go to node 5 and there we see that the next bit is a '0' (branch '0' refers to node 4). We thus obtain the new character '01' for the 'b'. We may do that for all the text "abca" that is changed into 101001. □
The compressor's pseudo-code is then:

```
    do
1       node = read one character
        do
2           line = table(node,'father')
3           if node = table(line,Branch '1') then   output '1'
                else   output '0'
```

```
4           $node = line$
5       until  $line$ = 'root'
6   until  eof
```

Decompression:

To decompress, we simply cover the tree in the reverse side, from the root to a leaf.

Example (cont.): We always start from node 5. If the first bit is a '0' we go to node 4, then if we have again a zero we go to node 3. There we stop as we are at a leaf. The decoded character is a 'c' ('00'). □

The decompressor's pseudo-code is:

```
    do
1       $line$ = 'root'
        do
2           $bit$ = read one bit
3           $node$=line
4           if $bit = 0$ then   $line$ = table(line,Branch '0')
                else   $line$ = table(line,Branch '1')
5       until  $node \leq$ size of the alphabet
6       Output character
7   until  eof
```

As we can see, due to the fact that the tree is covered in one side for the compression and in the other side for the decompression, the generated bits come from the MSB to the LSB for the compression and in the other side for the decompression. For that and to avoid additional complexity and the use of more RAM memory for the compressor/decompressor we need to decompress a file from one end and we recover it from the other end. It is not constraining if we take it into account, for example sending a file to compress by the card in the reversing order.
An important property of the Huffman compressor is that we may start the decompression directly at an entry point everywhere in the compressed file (by an external pointer indicating for example that a special procedure is available at a known position in the compressed file). The decompressor simply begins to read the first bit at that position. This property is not directly available with the arithmetic compressor described below.

Memory Requirements:

It is immediately obvious that the ROM code needed to implement the compressor/decompressor will not be very big. For example on the ARM7M 32-bit RISC processor used for the future smart card developed in the European CASCADE EP8670 project, the sizes of the compressor/decompressor are both inferior to 300 bytes (compiled C code). When this implementation will be done in assembler it may be certainly better.

The RAM needed to perform the computations is almost null as all the computations may be done on the fly without retaining a lot of intermediate results. This is only true when considering the choice we have done for the implementation at the end of the preceding section concerning the compression/decompression in reversing order. On the ARM7M, the internal registers are sufficient.
The only additional needed memory is for the used table. Several options are possible. If all the data to be processed are of the same type, only one table may be put forever in (EEP)ROM or temporarily stored in EEPROM or in the future, when we will have sufficiently RAM, directly in it. In any case, some memory must be used for it during the compression (decompression). As suggested before, this size depends on the dimension of the alphabet used. For an n-bit alphabet we need 2^n values of n-bit to memorize the father of each leaf, $2 \times 2^n - 1$ values of $n+1$ bit (there are $2 \times 2^n - 1$ nodes in a tree) to memorize the children of each non leaf node and $2^n - 1$ values of n-bit to memorize the father of these nodes. Only n-bit values are needed to memorize the fathers as there are $2^n - 1$ fathers. With an 8-bit alphabet we thus need 1086 bytes of memory for the compression/decompression tables. If we want to implement only the decompression, the memorization of the fathers is not necessary and only $2 \times 2^n - 1$ values of $n+1$ bit are needed for the table. For an 8-bit alphabet it corresponds to 574 bytes.

4.1 Speed

The Huffman compressor's speed is very fast. As we can see, there is nearly no computation for the compression or decompression algorithms. The only critical points are the memory access of the table needed to generate the new characters. For the conversion of a character, this time directly depends on the depth of the tree.
Indeed, with a well suited table for a file to decompress, a character occurring frequently will have a very small path between its leaf to the root. The mean depth of the tree is $\lceil \log_2 n \rceil$ but it may be equal to n in the worst case and to 1 in the best case.
As a consequence, we may also say that a badly compressed file is also compressed/decompressed in a very poor time
Practically, on the ARM7M processor (@ 20 MHz), the speed for a compression or a decompression on files like ARM7M object code or English text files always lays between 50 Kbytes/s and 100 Kbytes/s.

5 Arithmetic Algorithm

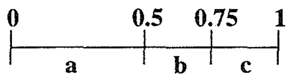
The arithmetic algorithm is described in detail in [7].

5.1 Description

This algorithm subdivides the [0,1] interval into the same amount of segments that there are characters in the alphabet. The length of these sub-segments is related to the character's probabilities. To compress, we read the character, take the corresponding subinterval and subdivide it again in the same way. After determining the short interval of the complete text, we take a number representing the compressed text inside this interval. To decompress, we take this number and look in which segment of [0,1] it lies and then we find the corresponding character. We multiply the actual value by a number corresponding to the expansion of the preceding interval to [0,1].

As for the Huffman algorithm, we need a modeler to build a table with each character and the corresponding cumulative statistics. Nevertheless, the table is here strictly identical for the compressor and the decompressor.

Example: With the same example as before, let us compress the text 'abca'. We have the initial interval:



As the first character is 'a', the interval becomes [0,0.5]. Next, 'b', the interval becomes [0.25,0.375] (taking the third quarter of [0,0.5]). Next, 'c', the interval becomes [0.34375,0.375]. Finally 'a' gives the final interval: [0.34375,0.359375]. The algorithm chooses finally the number 0.345 (or any other in the interval).

To decompress, we take the number 0.345. It lies between 0 and 0.5. Therefore the first character is 'a'. Next, we multiply this number by the factor readjusting the character's interval on the [0,1] interval, it gives $2*0.345 = 0.69$). So, we can conclude that the second character is 'b'. The number becomes then $4*(0.69-0.5) = 0.76$. The third character is 'c'. The number is now $4*(0.76-0.75) = 0.04$. And the last character is 'a'. □

We note at this point that if the decompressor do not know the length of the decoded text or do not use an "end-of-file" character, it can loop indefinitely.

The pseudo-code of the compressor is:

```
$L = 0, H = 1$
**do**
    $car$ = read input
    $(P_i, P_{i+1})$ = Interval of $car$
    $L = P_i * (H - L) + L$
    $H = P_{i+1} * (H - L) + L$
**until** eof
Choose number $T$ such that $L < T < H$
```

The pseudo-code of the decompressor:

```
1   Read T
    do
2       $(P_i, P_{i+1})$ = Interval of $T$
3       Output the character corresponding to $(P_i, P_{i+1})$
4       $T = (T - P_i)/(P_{i+1} - P_i)$
5   until eof
```

The "Interval of" function is the same for the compressor and the decompressor. As the table made by the modeler is the cumulative probabilities of the characters in growing order, the function simply inspects the table line after line to find the interval. The position in the table represents the character.

The decompressor algorithm described before has two important drawbacks:

- We work with all the compressed text ($= T$), and thus it must be entirely in memory before the beginning of the decompression. This problem may be solved by a scaling method allowing to work with an input stream and not directly with all the data [7].
- The use of a division may slow down the speed of the decompressor as the multiplication in the case of the compressor. In the future smart cards like CASCADE, the used processor is sufficiently powerful so that even if it is somewhat slower than the Huffman compressors, it remains usable.

The implementation for a smart card use is not fully carry out for the moment. However, on the CASCADE chip, the ROM code will be inferior to 1 Kbyte for the compressor and the decompressor. The optimization is in progress. As for Huffman, the needed RAM will certainly not be a problem. The big advantage is in the table size: only 512 bytes are needed (compressor and decompressor together) with an 8-bit alphabet and using the fact that we never compress large files ($>$ 65 Kbytes) in the smart card context. Furthermore, the compression level is usually better than with Huffman.

6 Remarks

All the results below are given for an 8-bit alphabet. This choice may be different depending on the data to treat. Furthermore, if the alphabet is smaller, then the used table shall also be smaller. In fact, we use compression algorithms based only on the probability occurring of the characters in a file. If the same alphabet is not used for the "creation" of the file (an English text file is normally coded with an 8-bit ASCII alphabet), and for the compression, the compression/decompression efficiency will be very bad. Indeed, the overlapping of the characters will be rare except if the size of the alphabet is a multiple of the other. On the other hand, if the alphabet is too small, the efficiency is also reduced. Thus to choose a 4-bit alphabet for the compression of a text file is not so good compare to the results with 8 bits.

Possible problems linked to the statistics:

- if the statistics are the same for each character and all the possible characters are used, the tree is perfectly balanced and there is no compression.
- If the statistics are the same but all the possible characters are not used, there is no compression but an expansion! This case is simply solved by reducing the alphabet or by using the original file in place of the "compressed" one.
- If the statistics are all different with a maximum distance between each others, the tree is totally unbalanced, the compression is very good for a file having the same (or nearly) statistics but may be very bad for another one having a totally other statistics.

7 Compression Efficiency

First, we link the compression rate of one algorithm to the optimal one described by the entropy theory. After we present our results on some typical files.

7.1 Definitions

Let us first define the following terms:

n = Number of characters in the alphabet.
p_i = Probability of character i.
q_i = True probability of character i in a sample.
k_i = Number of bits to represent the character i.
α = Compression efficiency.

We may write:

$$\begin{aligned}\alpha &= \frac{Length\ of\ compressed\ message}{Length\ of\ uncompressed\ message}\\ &= \frac{\sum_i^n q_i k_i}{\log_2 n}\\ &= \frac{\sum_i^n q_i k_i}{\sum_i^n p_i k_i} \cdot \frac{\sum_i^n p_i k_i}{\sum_i^n p_i \log_2(\frac{1}{p_i})} \cdot \frac{\sum_i^n p_i \log_2(\frac{1}{p_i})}{\log_2 n}\\ &= f_a \cdot f_b \cdot \alpha_{opt}\end{aligned}$$

Where:

- f_a is a function of the divergence between the samples.
- f_b is a function of the chosen algorithm and the p_i. An upper bound of this function is given by [4]: $1 + \frac{p_{max}+0.086}{\sum_i^n p_i \log_2(\frac{1}{p_i})}$

- α_{opt} is the lower bound of the compression rate: by the entropy theory [2], it is equal to $\frac{\sum_i^n p_i \log_2(\frac{1}{p_i})}{\log_2 n}$.

This formula separates the influence of the statistics α_{opt} and the influence of the algorithm f_b on the compression efficiency. In our case, we suppose to know the exact statistics before compression: f_a will be always equal to 1.

7.2 Obtained Results

It is very difficult to have a reference to compare compression algorithms. One way is to use a well known data base of files for this type of work. Some files are effectively often used for that but they are not numerous and not well suited for tests in the smart card scope. Nevertheless we present here some results on these files to give a comparison with some well known compression algorithms developed without constraints linked to the memory or the computing power. As this work was done in the framework of the CASCADE project, some of the test files are also ARM7 object code.

The used files here are:

- *Book2*: Standard English text file often used for compression rate comparison. Size: 610856 bytes.
  Available at: ftp.cpsc.ucalgary.ca in /pub/projects/text.compression.corpus/
- *progc*: Source file in C language also often used for comparison and available at the same place. Size: 39611 bytes.
- *gosip_v2.txt*: English text file. Size: 262134 bytes.
  Available at snad.ncsl.nist.gov in /pub/gosip/
- *cjpeg*: ARM7 object code. Size: 729796 bytes. Available after compilation with the ARM 2.0 Software Development Toolkit.

We may directly compute α_{opt} of these files by computing the character's probabilities.

For Huffman, when the table (tree) is built, each k_i is known and the "true" α is then directly computable. In the following table, we can see the good efficiency of the Huffman coding (α_{opt} is very near to α). This is always true only if we compress a file with the table build with this file!

	Book2	progc	gosip_v2.txt	cjpeg
α_{opt}	0.5991	0.6499	0.5187	0.7225
$\alpha_{Huffman}$	0.6029	0.6542	0.5269	0.7251
f_b	1.0063	1.0066	1.0158	1.0036

We conclude that for a compression only based on the probabilities of each character taken independently, the Huffman algorithm is nearly optimal. Nevertheless, as shown in the following table, the performances are poor compared to dynamic algorithms like Block coding or Gzip where not only the characters probabilities are taken into account. But they are not well suited for smart cards. The performances of the basic static arithmetic algorithm are very similar to (our) Huffman implementation. Indeed, despite its poor speed and higher complexity compared to Huffman, the table size and the compression performances remain attractive. The following table gives the winning in percent from the uncompressed text.

	Book2	progc	gosip_v2.txt	cjpeg
Our Huffman	39.7%	34.6%	47.3%	27.5%
Static Arithmetic	40.3%	34.6%	48.6%	30.6%
Dynamic Arithmetic				
by words	68.0%	61.0%	77.0%	61.4%
by bits	57.5%	53.6%	67.8%	53.7%
Block Coding	68.6%	67.7%	77.3%	78.0%
Gzip	66.2%	66.5%	74.8%	82.2%

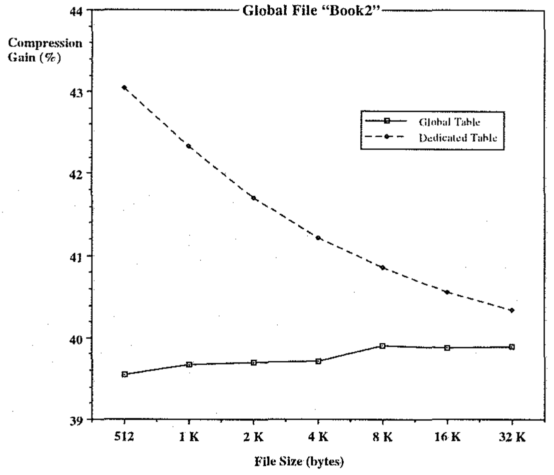
The algorithms in this table are public:

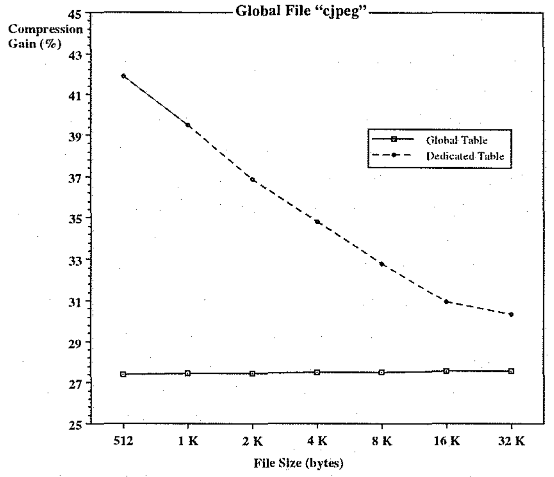
- Static arithmetic: Proposed in [7]. Available at ftp.cpsc.ucalgary.ca in /pub/projects/ar.cod/
- Dynamic arithmetic by words/bits: This is an improvement of the algorithms in [7]. Available at munnari.oz.au in /pub/arith_coder.
- Block Coding: For some description see [8].
  Available at ftp.cl.cam.ac.uk in /users/djw3/
- gzip : Well known GNU "zip" based on the Ziv-Lempel algorithm [9]. Namely available at ftp.switch.ch in /mirror/gnu/

Our Huffman implementation remains attractive even with tables not directly built separately for each file. One important thing for smart cards is the possibility to use a unique table for files to compress. Indeed, if several ones are close to 1 Kbyte, it is a non-sense to compress them as each table is near of 1/2 K if we want only to decompress and of 1 K if we want to compress/decompress.

For the following graphs, the files "book2" and "cjpeg" were cut in pieces of 512 bytes, then of 1 Kbyte, 2 K, 4 K, 8 K, 16 K and 32 K. For each set of files we computes the compression gains with a table built from the original file and with one particular for each small file. The graphs represent the gain's mean on each set of files of equal size. It will give naturally the same result if we really used the concatenation of several files of similar types.

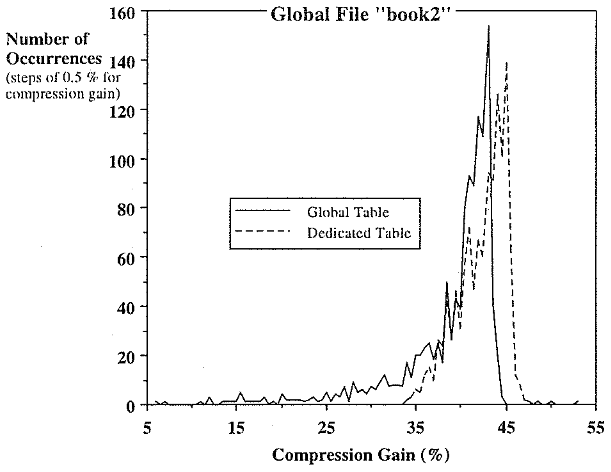
We see that the compression gain is always better for dedicated tables than for a global table. The compression gain always tends to the one of the global file. For some statistics like with the file "cjpeg", the compression gain is much more better for small files with dedicated tables than for larger files (this fact is also true for "book2" but less significant). It is due to the influence of the table on the compressed files that is bigger on small files (for a fixed alphabet, the table's size is identical for each file's size). It thus shows also some influence of the alphabet's size on the compressed files in relation to their sizes. For a global table, the compression gain is nearly constant whatever the file's size, even if it is better for files closer to the largest one.

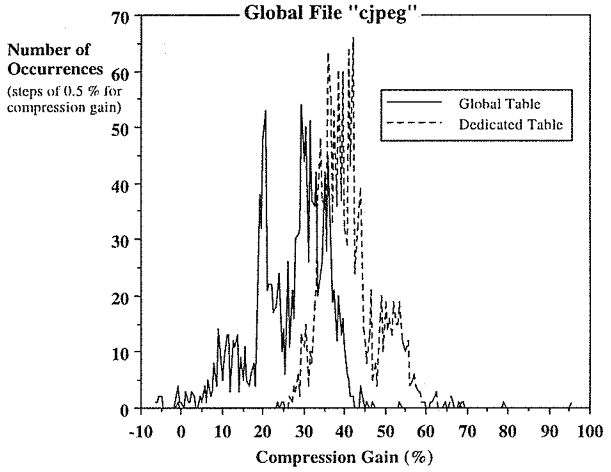




In fact the small files used to build the global table must be also "close" enough to be able to have the given results. It is evident that building a global table from the concatenation of text files, processor object code and picture files will give a very bad result on all the files! What we must try to do in this case is to have a general table for text files, another one for the object code,...
The next two graphs give for "book2" and "cjpeg", the distribution of only the 512-byte files in term of memory gain in percent. We can see that outside the mean compression factor, there are very few files having bad compression rates. The graph for the ARM object code is less clean (except for dedicated tables) because if we take very small pieces of code from a large file there is always some very particular zones doing very specific things. If we take larger files, this problem progressively disappears. We can see that there are also 9 files (out of 1425) having a negative compression factor. For these files, the algorithm must clearly choose to keep the original. We also see that with an individual table,

no such problem remains.





Even with more differentiated files like when using "cjpeg", we see that the compression remains useful even for smart cards. For real smart card applications its seems that a compression gain of minimum 30% is realistic with a global table. It implies that in this worst case, the advantage of using a compression algorithm begins with a total number of files to compress over 3 K, if we use only a decompressor (table of 574 bytes and code < 300 bytes), and over 5.4 K with both compressor and decompressor (table of 1086 bytes and code < 600 bytes). Viewing all the results together, it remains also very important that for such static compression algorithm we implemented, the more information we can have in advance on the files to compress, the better will be the compression (it is not so crucial for the usual dynamic compressors). In every case it will certainly be better to have several compression algorithms (or versions of the same algorithm) to be more suited for some specific data.

8 Conclusion

After surrounding the present needs for compression algorithms suitable for smart cards, we focus on two lossless static algorithms: Huffman, and the Arithmetic. We reduce the memory needed for the code and the table to adapt the Huffman algorithm in this context. We obtain a good compression efficiency in spite of the little code and table length and the restriction on the statistics. The use of a unique table for several files is also possible. We are on the way to find similar results for an arithmetic compressor. The evolution of the compression in smart cards will be likely to replace the static compressor by a dynamic compressor and when the RAM space will be sufficiently important to handle a table only in RAM. Other works on algorithms like JBIG or JPEG may perhaps also lead to similar results for other kinds of data.
We hope that this work will encourage people to continue on that fruitful way where a lot of things seems still to be found.

9 Acknowledgments

We thank the UCL Crypto Group and particularly Ir Sylvie Lacroix for her support and for the realization of the graphics showing the performances of the Huffman compressor.

References

[1] J. A. Storer, "Data Compression: Methods and Theory", Computer Science Press, 1988.

[2] C. E. Shannon, "A Mathematical Theory of Communication", Bell Syst. Tech. J 27, pp. 379-413, 1948.

[3] D. A. Huffman, "A method for the construction of minimum redundancy codes," Proc. IRE, vol 40, pp. 1098-1101, 1952.

[4] R. G. Gallager, "Variation on a Theme by Huffman," IEEE Transactions on Information Theory, vol 24, no 6, pp. 668-674, November 1978.

[5] J. S. Vitter, "Dynamic Huffman Coding" ACM Transactions on Mathematical Software, vol 15, No 2, pp. 158-167, June 1989.

[6] P. G. Howard, J.S. Vitter, "Arithmetic Coding for Data Compression," Proceedings of the IEEE, vol 82, no 6, pp. 857-864, June 1994.

[7] I. H. Witten, R. M. Neal, J. G. Cleary, "Arithmetic Coding for Data Compression" Communication of the ACM, vol 30, no 6, pp. 520-540, June 1987.

[8] D. Wheeler, "An Implementation of Block Coding", available at gatekeeper.dec.com in /pub/DEC/SRC/research-reports/SRC-124.ps.Z, October 1995.

[9] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, vol 23, no 3, pp. 337-343, May 1977.

This article was processed using the LaTeX macro package with LLNCS style

Structuring and Visualising an IC-card Security Standard

Hugh Glaser[1], Pieter H. Hartel[1,2], Eduard K. de Jong Frz[3]

[1] Department of Electronics and Computer Science, University of Southampton, UK, Email: hg,phh@ecs.soton.ac.uk.
[2] Faculty of Mathematics, Computer Science, Physics and Astronomy, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, Email: pieter@fwi.uva.nl.
[3] QC Technology, Zaandam, The Netherlands, Email: eduard@q2c.nl.

Abstract. The standard way of visualising protocols using pictures with boxes and arrows is insufficient to study the protocols in detail. The problem is that the structuring of the protocols relies on elements not explicit in the usual visual rendering. To solve the problem one should visualise not only the operations and the messages but also the state and the security. This paper presents a system which can be used to visualise a protocol, and is applied to some of the protocols in the load purse transaction of the CEN Inter-sector electronic purse draft standard as an example. The resulting conformant prototype provides abstract and concrete views on the system at all significant levels. The prototype supports animation of the standard, giving the protocol designer feedback on design decisions.

Keywords: protocol visualisation, protocol structure, standards, smart cards.

1 Introduction.

The security of IC-card systems depends as much on the cryptography as it does on the quality of the system implementation. Provably correct implementations of the relatively complex transaction protocols are as yet infeasible because of their high cost. The costs of provably correct implementations of complex software will come down [11], but presently one has to compromise. The only technique available for building secure systems is to structure the system carefully. Many small building blocks and lean interfaces make it difficult for errors to permeate the entire system. If furthermore the building blocks can be composed easily, a high level of confidence in the reliability of the system is possible.

When given a particular application area, it is possible to produce structures that give the required degree of modularity without unduly constraining the system performance. The work described here is directed at producing an appropriate structure to support the security architecture in an emerging standard. It is important to scrutinise standards for potential problems in the implementations. Much effort is put into drafting standards, and once accepted, the standard will persist for a long time.

Our efforts form part of a larger project to build prototypes for IC-card systems so as to study the properties of these systems. In previous work, we have taken a rather more formal approach to prototyping smartcard and network architectures using similar transaction protocols [7]. The focus of the present paper is on the structural aspects of the transactions, and more precisely on the clarity of structure rendering. This does not preclude the use of formal methods to make additional statements about the correctness of the system, but this is not our aim here.

The present paper emphasizes the structural aspects of transaction protocol implementation through the use of a visual, object oriented, data flow programming language Prograph [6]. Important visual cues inherent in the work with protocols are most naturally rendered by a visual programming language. A full implementation of a protocol makes it possible to run simulations of the protocol, and even more importantly to animate the protocol. The visual feedback to the protocol designer and prototype builder is vital to a full understanding of the design.

A structure has been developed for the transaction protocols in the European draft standard for an Inter-sector electronic purse [5]. This document will henceforth be referred to as 'the standard'. The standard contains several different transaction protocols, which fit the proposed structure. We believe that other transaction protocols will also be accommodated by the proposed structure.

The next section introduces our case study, the protocols from the standard. In Sect. 3 we discuss principles of structuring synchronous protocols. These are used to describe a particular protocol from the standard in Sect. 4. This is followed by a description of a prototype animation in Sect. 5. Related work and conclusions are presented in the last two sections.

2 Inter-Sector Electronic Purse.

The standard [5] describes a security architecture for electronic purse systems. In the electronic purse system three parties are defined. The purse provider (for example a bank) issues the electronic purse. The service provider (for example a shop) accepts electronic payment for services. The card holder charges the electronic purse with funds and uses the purse to pay for services. The system works in a similar way to travellers cheques. These are also prepaid by the user and accepted as payment by a service provider. The organisation issuing the travellers cheques guarantees the cheques to both the user and the service provider. The electronic purse is not a real purse in the sense that two purse holders cannot exchange funds.

The electronic purse system defines a number of protocols for the three parties to use when communicating for a specific purpose. The major protocols are:

- *Load* the purse with the electronic counter value of a certain amount of conventional currency. This transaction involves the card holder and the purse provider.

- *Purchase* a service using the electronic purse to effect payment. This transaction involves the card holder and the service provider.
- *Collect* the totals of the electronic transactions. This protocol credits the service provider with the conventional counter value of the electronic money (similar to a shop keeper paying travellers cheques into the bank account). This transaction involves the purse provider and the service provider.

A number of other protocols are defined for maintenance purposes, such as monitoring the status of the purse, conversion of currency etc.

In the standard, the transaction protocols are partly described using natural language, partly using a BASIC-like programming language. The standardisation committee has spent considerable effort in clarifying the important issues. However, the transactions are sufficiently complicated to make the descriptions verbose, difficult to interpret and error prone. The use of adequate structuring tools would have been useful to increase the clarity of the descriptions.

2.1 Structure in the Standard

The standard is structured as a four part document. Within each part various structuring tools are used:

- *Part 1* describes the basic business relationships of the purse provider, the service provider and the purse holder. The three parties conduct their business through a number of logical components that interact. The standard specifies the functionality and interconnections of these logical components. The standard distinguishes Secure Application Modules (SAM) from other modules that are not secure.
- *Part 2* describes the security architecture of the electronic purse system. The description consists of a global overview and a detailed description of each protocol. The global overview is cryptographically neutral. The detailed description of a particular protocol will be shown specialised towards a particular cryptographic system. For most protocols, both an RSA and a DES version exist.
- *Part 3* describes the data elements and interchanges.
- *Part 4* describes the devices.

In the structures listed above it is Part 2 that provides a handle on the structuring of the actual protocols; It gives global and a detailed view on the protocols. The remaining structuring tools offered by the standard are aimed either at the levels above the protocols, such as the flow of electronic value and payment, or at levels below the protocol. They are thus not relevant to our presentation, which deals with the protocols.

3 Structure in a Transaction Protocol.

The structure of a protocol from the standard is highlighted by separating the essential elements of the protocol from the details. This results in a two part description consisting of a one page overview of the main elements of the protocol (see Fig. 2 for an example) and a listing of the details running over several pages (for a fragment see Fig. 3, a digest of these figures is given in Sect. 4).

The structure of the one page overview follows the exchange of messages that arises when a particular transaction is made. This is typical for the description of cryptographic protocols, which mostly follow the synchronous pattern as shown in Fig. 1a. Here two parties 'Alice' and 'Bob' exchange a number of messages whilst each performs some actions A1 ... A6 and generate some responses R2 ... R5.

Alice		Bob
A1:		
	$\longrightarrow$	
		A2:
		R2:
	$\longleftarrow$	
A3:		
R3:		
	$\longrightarrow$	
		A4:
		R4:
	$\longleftarrow$	
A5:		
R5:		
	$\longrightarrow$	
		A6:

(a) Disregarding the state owned by the parties.

Alice		Bob
A1:		
	$\longrightarrow$	
		A2:
$\downarrow a_1$		$\downarrow b_1$
		R2:
	$\longleftarrow$	
A3:		
$\downarrow a_2$		$\downarrow b_2$
R3:		
	$\longrightarrow$	
		A4:
$\downarrow a_3$		$\downarrow b_3$
		R4:
	$\longleftarrow$	
A5:		
$\downarrow a_4$		$\downarrow b_4$
R5:		
	$\longrightarrow$	
		A6:

(b) Explicitly tracing the state owned by the parties.

Fig. 1. Two abstract renderings of a synchronous protocol where two parties exchange messages. The A_i represent actions of the parties. The R_i are the responses and the a_i and b_i represent the state owned by the two parties.

The protocol of Fig. 1a does not expose much detail but it does show that we are dealing with a synchronous protocol between two parties. To obtain more information about the protocol one would have to zoom in on the A_i and R_i. The standard does exactly this, and it does so at two levels of detail. The first level exposes some detail (the one page overview) and the second level exposes all detail. The standard is reasonably successful when it describes the essential elements of a protocol in the one page overview. The data elements that are manipulated by the partners in the transaction are exposed as well as the data elements that are transmitted in the messages. Some of the relevant operations on the data elements are also exposed, such as 'sign' and 'verify' (these operations apply to signatures).

The standard uses no other structuring methods. We have identified two additional structuring tools that can successfully and usefully be applied to a protocol description. These tools will be presented in the next two sections.

3.1 Ownership of Data Imposes Structure

The first additional structuring tool extends the idea that in a protocol data is owned by one of the two parties. Data that is not transferred in a message is not accessible to the other party. We will often call the data that is owned by a party the state of that party. Only data that is carried explicitly from one action to the next is accessible. Data that is not explicitly traced becomes inaccessible. Tracing the state in an abstract protocol is schematically rendered in Fig. 1b. For example, the arrow labelled a_1 carries the state of Alice from action A1 to action A3.

In a concrete protocol one might trace all data elements explicitly from its producer to its consumer, so that it is clear where data is produced, accessed, modified and destroyed, thus making it possible to reason about the (mis)use of data elements.

3.2 Action Details Impose Structure and Sub-structure

The second structuring tool that we wish to impose on a protocol description concerns the internal structuring of the actions taken by a party. The following three steps are essential within each action:

- *Operational aspects* describe what the protocol is for and how the operations it supports take place. For example, The electronic purse protocols serve to transfer funds. The structure should make it easy to find out when and how funds are transferred.
- *Security aspects* describe what security is used for and how it is achieved. Security in the electronic purse protocol serves to authenticate the parties and to guarantee non-repudiation. It must be possible to uniquely identify transactions and parties involved in transactions. Security is delivered by cryptographic means. The structure should identify the cryptographic data and operations.
- *Protocol aspects* describe which data are transmitted and when the data is transmitted. The remaining data are kept locally by the parties involved in a transaction.

The low level details of each of these aspects should be independent of the high level aspects. For example it should be possible to exchange one cryptographic system for another without affecting the functionality of the protocol.

The three steps within each action are sufficient to describe transaction protocol in the ideal world. Adding security aspects to the protocols requires each step to be subdivided into several sub-steps. We identify six sub-steps in each action:

- *Protocol aspects – capture vars* receives the message. The contents of the message are captured in the state.
- *Operational aspects – operational conditions* check the state for operational errors. After checking the operational conditions, the state is known to be consistent.
- *Security aspects – protocol security* generates and checks the data used for authentication and non-repudiation. This includes the provision of unique identifiers for the components as well as the transaction.
- *Security aspects – cryptographic security* performs the cryptographic functions, such as encryption, hashing and signing of relevant data.
- *Operational aspects – operations* modify the state to reflect the progress made in the current transaction.
- *Protocol aspects – form response* gathers the data necessary to generate the response to the incoming message.

The structure of an action is sufficiently general to capture the pattern of computation imposed by the transaction protocols of the standard.

3.3 Using the Data Flow Paradigm to Make Structure Concrete

Having identified two new ways of imposing structure on a transaction protocol, we will now show that the data flow paradigm [9, 14] is ideal to support this structure. There are two reasons for this. The first is that both data flow programs and protocols are naturally highly visual. The second reason is that the data flow paradigm requires data to be traced explicitly from its source to its destination.

At the lowest level, data flow is based on the manipulation of data by state-less operators. An operator can have several inputs and outputs. An operator fires as soon as all its inputs are available. To pass data from one operator to another requires an explicit connection between the operators. This makes it possible to trace explicitly the origin and destination of all data. The basic state-less operators have no memory; state needs to be manipulated explicitly as data. Operators can be connected to form a new, more powerful operator. These constructed operators may have state, which at the implementation level of the constructed operator would be clearly visible. When viewed as a separate entity, the state of a constructed operator would not be visible.

The protocol of Fig. 1b can be given a data flow interpretation. This requires interpreting the actions A_i as (constructed) operators. The arrows in the protocol form the data flow edges.

A data flow network is most naturally represented as a picture. This corresponds well to the usual rendering of protocols. The extra edges (cf. the difference between Fig. 1a and b) needed for a data flow diagram to thread the 'state' of each of the parties in the protocol gives the picture an extra visual cue. It shows clearly that the state is not threaded arbitrarily through the system, and that the state is not accessible to just anyone. Instead the state is confined to particular actions. The state must be confined, because it may contain secret information. Obvious secrets that must not leak are cryptographic keys. In many applications the operational data is also regarded as sensitive. In all cases it must be clear where the sensitive information resides, so that proper protective measures can be made.

The graphical nature of the data flow model gives leverage in the prototyping of a protocol for a security architecture. In an interactive, GUI based system such as Prograph, the visual aspect allows the designer to interact directly with the prototype. Animations of the system make it possible to trace the flow of data. The animation can be stopped to allow data to be inspected and altered. This gives the designer immediate and valuable feed back.

4 Load Transaction from the Draft Standard.

The standard describes a number of transaction protocols in the style of Fig. 2. It is sufficient for the present discussion to describe only the essential elements of one of the transactions: the load purse transaction. The details the transaction may be found in the standard.

Figure 2 shows three parties engaged in a the load purse transaction: The Inter-sector Electronic Purse (IEP), the Load Device Application (LDA) and the Purse Provider Secure Application Module (PPSAM). The LDA acts as an intermediary driving the transaction because it is the device operated by the purse holder. From the security point of view, the LDA is transparent. This implies that our abstract two party protocol is applicable to describing the protocol from the standard. An important aspect of the driving function of the LDA is that it always checks the completion codes of the IEP and the PPSAM. If the completion code indicates an error, the LDA informs the purse holder and aborts the entire transaction.
Steps A1–C2
The transaction starts when the purse holder inserts the IEP in the LDA device. Step A1 acquires the amount to be loaded M_{LDA} and currency $CURR_{LDA}$. Together these two values are referred to as $\overline{M}_{LDA}$. Step C2 sends a message to the PPSAM containing the command to 'initialise PPSAM for Load'. The parameter of the command is the amount and currency to be loaded ($\overline{M}_{LDA}$).
Steps A2–R2
The PPSAM at step A2 first checks that it is able to deal with the requested currency $CURR_{LDA}$. Step A2 then checks that the balance of the PPSAM, BAL_{PPSAM}, is sufficient for the load transaction. Then the PPSAM generates a random number R which will serve to uniquely identify the transaction. At step R2 the PPSAM forms the response message to the LDA. The response message contains as parameters the identity of the PPSAM and the random number.

	IEP			LDA			PPSAM
			A1:	Acquire M_{LDA}			
			C2:	Initialise PPSAM for Load parameters($\overline{M}_{LDA}$)	⟶		

						A2:	verify($CURR_{LDA}$)
							verify($BAL_{PPSAM} \geq M_{LDA}$)
							Generate random R

					⟵	R2:	response(PP_{PPSAM}, PPSAM, R)
		⟵	C3:	Initialise IEP for Load parameters(PP_{PPSAM}, PPSAM, R, $\overline{M}_{LDA}$)			

A3:	verify($CURR_{LDA}$)
	verify($BAL_{IEP} + M_{LDA} \leq BALmax_{IEP}$)
	NT_{IEP}:=NT_{IEP} + 1
	S_1:=sign(PP_{IEP}, IEP, $DEXP_{IEP}$, NT_{IEP}, $\overline{M}_{LDA}$, BAL_{IEP}, PPSAM, R)						

R3:	response($\overline{FD}_{IEP}$, NT_{IEP}, BAL_{IEP}, S_1)	⟶					
			C4:	Debit PPSAM parameters($\overline{FD}_{IEP}$, NT_{IEP}, BAL_{IEP}, S_1)	⟶		
						A4:	verify(has got/will get conventional payment) verify($\overline{FD}_{IEP}$) verify(S_1) S_2:=sign(PP_{PPSAM}, IEP, NT_{IEP}, $\overline{M}_{LDA}$) BAL_{PPSAM}:= $BAL_{PPSAM} - M_{LDA}$
					⟵	R4:	response($\overline{IK}_{PPSAM}$, S_2)
		⟵	C5:	Credit IEP parameters($\overline{IK}_{PPSAM}$, S_2)			
A5:	verify(S_2) S_3:=sign(PP_{IEP}, PPSAM, R, CC_{IEP}) BAL_{IEP}:=BAL_{IEP} + M_{LDA}						
R5:	response(S_3, CC_{IEP})	⟶					
			C6:	PPSAM Load acknowledgement parameters(S_3, CC_{IEP})	⟶		

						A6:	verify(S_3)
							verify(CC_{IEP})

					⟵	R6:	response(CC_{PPSAM})

Fig. 2. Load transaction without an LSAM from the standard.

The identity of the PPSAM consists of two elements: the identity of the device, PPSAM, and the identity of the purse provider who owns the device PP_{PPSAM}.

Step C3

The LDA sends the 'initialise IEP for load' command to the IEP, with the amount and currency to be loaded. The LDA also sends the identification information of the PPSAM to the IEP.

Steps A3–R3

The IEP first checks that it can deal with the requested currency. It then checks whether its balance BAL_{IEP} would not exceed a predetermined maximum $BALmax_{IEP}$. The IEP increments its transaction count NT_{IEP} to generate a unique number of its own. The first authentication related step is the creation of the signature S_1. It contains a number of relevant data elements including the expiration date of the IEP, $DEXP_{IEP}$, and the identity of the IEP, the identity of the PPSAM and the unique numbers generated by the two parties. The response message generated at R3 packages all relevant information for the LDA. The notation $\overline{FD}_{IEP}$ is an abbreviation of a list of various data elements including identities, expiration date and keys for the security system.

Step C3

The LDA continues the transaction if the IEP is able to deal with the currency and amount to be loaded. It sends the 'debit PPSAM' command with the relevant parameters to the PPSAM.

Steps A4–R4

The PPSAM checks that the counter value of the money to be loaded has been supplied by conventional means. This could be done by, for example, requiring the purse holder to insert some bank notes into the LDA device. The data contained in $\overline{FD}_{IEP}$ will be checked. This involves testing operational conditions, such as whether the PPSAM and the IEP are indeed able to interact with one another, whether the IEP has perhaps been cancelled etc. The second verification step authenticates the IEP to the PPSAM via the signature S_1. The signature S_2 is then computed, so that later in step A5 the IEP is able to authenticate the PPSAM. The PPSAM has now been satisfied that all the operational conditions have been checked. Furthermore it knows that it is communicating with a genuine IEP, so the balance of the PPSAM can be decremented. This is the first 'real operation'. The response $\overline{IK}_{PPSAM}$ as generated at step R4 contains a list of relevant data for the IEP.

Step C5

The LDA sends the 'credit IEP' command along with relevant parameters to the IEP.

Steps A5–R5

The IEP authenticates the PPSAM. The IEP then computes its second signature, S_3, which is necessary for the PPSAM at step A6 to make sure that the load has indeed taken place. If one were to remove the IEP, for example, whilst it is carrying out step A5, the PPSAM needs to know about this so that it can undo the debit operation on its balance. The second 'real operation' of the transaction is to increment the balance of the IEP, just before the response at step R5.

Step C6
The LDA sends the 'PPSAM load acknowledgement' command. The completion code CC_{IEP} is passed to the PPSAM so that it can check whether the IEP has been successful in incrementing its balance.
Steps A6–R6
The final step checks the completion of the IEP. Step R6 sends a final response to the LDA, which can be used to produce a receipt. This has not been made explicit in the given transaction protocol, as it is not interesting from the point of view of the security.

4.1 Analysis of the Transaction Structure

The transaction shown in Fig. 2 contains all of the aspects identified by the structuring tools of Sect. 3:

- Operational aspects:
  - Checking operational conditions at A2, A3 and A4.
  - Performing operations at A4 and A5.
- Security aspects:
  - Generation of signatures at A3, A4 and A5.
  - Verification of signatures at A4, A5 and A6.
- Protocol aspects:
  - Driving the transaction at C2 ... C6.
  - Generation of command messages at C2 ... C6 and the generation of result messages at R2 ... R6.
  - Decoding of command messages at A2 ... A6 and the decoding of result messages at C3, C4, C5 and C6.

The presentation of Fig. 2 focuses on the protocol aspects, as the messages are clearly identified. The other elements of the structure are not identified. We will first study some of the details of one of the actions in the protocol. Thereafter we present a description of the load transaction from the standard with all its structure clearly identified.

4.2 Details of Debit PPSAM

The analysis of the transaction applies to the level of detail as shown in Fig. 2. This ignores a number of important elements, such as which cryptographic system is used, and whether or not operations are logged. Such elements must play a role in a comprehensive analysis of the protocol. To illustrate these elements we will zoom in on the most interesting step A4–R4, which debits the PPSAM. This step is the only step which defines all of the operational, security and protocol aspects. Figure 3 shows step A4–R4 as it appears in the standard.

The first addition to the earlier, more abstract rendering of step A4–R4 is that we now see that a log is maintained of the actions taken by the PPSAM. The log contains relevant information necessary to trace all transactions.

	PPSAM	
A4:	verify(has got/will get conventional payment)	Operational conditions
	verify($\overline{\mathrm{FD}}_{\mathrm{IEP}}$)	.
	verify(S_1)	Protocol security
	S_2:=sign($\mathrm{PP}_{\mathrm{PPSAM}}$, IEP, $\mathrm{NT}_{\mathrm{IEP}}$, $\overline{\mathrm{M}}_{\mathrm{LDA}}$)	.
	$\mathrm{BAL}_{\mathrm{PPSAM}}$:= $\mathrm{BAL}_{\mathrm{PPSAM}} - \mathrm{M}_{\mathrm{LDA}}$	Operation
R4:	response($\overline{\mathrm{IK}}_{\mathrm{PPSAM}}$, S_2)	Protocol

Step	Action	Details	
A4:	Update the log	write(Load Log) [IEP, $\mathrm{NT}_{\mathrm{IEP}}$]	Operational conditions
	Is the conventional payment OK?	[Outside the scope of this standard]	
	Does the IEP belong to the PP?	IF $\mathrm{PP}_{\mathrm{IEP}} \notin \{\mathrm{PP}_{\mathrm{PPSAM}}\}$ THEN	.
		$\mathrm{CC}_{\mathrm{PPSAM}}$:=PP_MISMATCH	.
		write(Load Log) [$\mathrm{CC}_{\mathrm{PPSAM}}$]	.
		abort	.
	Is the IEP in the negative file?	IF ($\mathrm{PP}_{\mathrm{IEP}}$||IEP) $\in$ **NF** THEN	.
		$\mathrm{CC}_{\mathrm{PPSAM}}$:=IEP_OPPOSED	.
		write(Load Log) [$\mathrm{CC}_{\mathrm{PPSAM}}$]	.
		abort	.
	Has the IEP expired?	IF $\mathbf{DATE}_{\mathrm{PPSAM}} > \mathrm{DEXP}_{\mathrm{IEP}}$ THEN	.
		$\mathrm{CC}_{\mathrm{PPSAM}}$:=IEP_EXPIRED	.
		write(Load Log) [$\mathrm{CC}_{\mathrm{PPSAM}}$]	.
		abort	.
	Is the algorithm and key version supported?	IF ($\mathbf{ALG}_{\mathrm{IEP}}, \mathbf{VK}_{\mathrm{IEP}}$) $\notin$ $\{(\mathbf{ALG}_{\mathrm{PPSAM}}, \mathbf{VK}_{\mathrm{PPSAM}})\}$ THEN	Protocol security
		$\mathrm{CC}_{\mathrm{PPSAM}}$:=ALG_OR_KEY_MISMATCH	.
		write(Load Log) [$\mathrm{CC}_{\mathrm{PPSAM}}$]	.
		abort	.
	Select master key using key version	$\mathbf{KML}_{\mathrm{PPSAM}}$:=select(IEP, $\mathbf{VK}_{\mathrm{IEP}}$) [$\{\mathbf{KML}_{\mathrm{PPSAM}}\}$]	.
	Compute diversified key	$\mathbf{KD}_{\mathrm{PPSAM}}$:=encipher($\mathbf{KML}_{\mathrm{PPSAM}}$) [IEP]	.
	Compute session key	$\mathbf{KSES}_{\mathrm{PPSAM}}$:=encipher($\mathbf{KD}_{\mathrm{PPSAM}}$) [$\mathrm{DEXP}_{\mathrm{IEP}}$||$\mathrm{NT}_{\mathrm{IEP}}$]	.
	Verify IEP signature	IF $\mathrm{S}_1 \neq$ sign($\mathbf{KSES}_{\mathrm{PPSAM}}$)	.
		[$\mathrm{M}_{\mathrm{LDA}}$|| $\mathbf{CURR}_{\mathrm{IEP}}$|| $\mathrm{BAL}_{\mathrm{IEP}}$|| PPSAM|| R] THEN	.
		$\mathrm{CC}_{\mathrm{PPSAM}}$:=LOAD_S1_FAILED	.
		write(Load Log) [$\mathrm{CC}_{\mathrm{PPSAM}}$]	.
		abort	.
	Compute debit signature	S_2:=sign($\mathbf{KSES}_{\mathrm{PPSAM}}$) [$\mathrm{M}_{\mathrm{LDA}}$|| $\mathrm{CURR}_{\mathrm{LDA}}$]	.
	Decrease PPSAM balance	$\mathrm{BAL}_{\mathrm{PPSAM}}$:=$\mathrm{BAL}_{\mathrm{PPSAM}} - \mathrm{M}_{\mathrm{LDA}}$	Operations
R4:	Transmit PPSAM parameters and debit signature	response(S_2, $\mathrm{CC}_{\mathrm{PPSAM}}$)	Protocol

Fig. 3. Two levels of detail for the 'Debit PPSAM' step using DES from the Load transaction without an LSAM.

The first four IF statements show how the operational conditions are checked. The abort primitive causes the present step to be abandoned. It also generates a result message to the LDA containing only the completion code $\mathrm{CC}_{\mathrm{PPSAM}}$.

The identifiers in bold font are identifiers that have not yet been seen. They appear only at this level of detail. For example **NF** represents the negative file. This file contains the identities of the IEP's that are no longer valid, for example, because they have been reported lost.

The fifth IF statement shows how authentication works in the cryptographic protocol being used. A session key $\mathbf{KSES}_{\mathrm{PPSAM}}$ is computed, which is then used to encipher the data contained in the signature S_1. If the enciphered data fails to match the signature, the authentication fails.

The standard is in principle neutral with respect to the chosen cryptographic system. It provides diagrams such as Fig. 3 both for DES and for RSA.

This concludes the description of the standard load protocol. The main point of the description is that the structure of the protocol is easily obscured by a wealth of detail. We have indicated the structural elements by indicating to the right of the statements in Fig. 3 which parts of the structure they represent. In the following section we will look at a conformant prototype of the protocol structure from the standard, with the aim to clearly expose the structure.

5 A Prograph Prototype of the Protocol Structure.

It is desirable that the conformant prototype follows the standard as closely as possible, and so we expect to see the different aspects of the standard reflected in separate parts of the implementation. In this section we give the reader a flavour of the prototype, without becoming too involved in the details of the Prograph language.

We begin by looking at the first part of the standard, where the components are defined. The window that does this for the example we are using is shown in Fig. 4. There are five entities defined in the Prograph, which correspond to five of the entities from the standard. These are the components *PSAM*, *IEP* and *PPSAM* and the devices *LDA* and *PDA*. At this level, we have also chosen to expose the difference between the values that are part of the permanent data of components (the *State*), and the ephemeral or session data (the *Vars*), since the classification of data is important.

Prograph is an object-oriented language, and the lines between components indicate inheritance. The PSAMs, IEPs and PPSAMs are proper components since they inherit from *COMP State*. The LDAs and PDAs are not considered (by the standard) to have the same status, and therefore they do not inherit from the class *COMP*. When running the system, it is possible to open any of these icons and examine the attributes that instances of these entities have.

Most of the standard is concerned with the actual processing and flow of messages, and so it is important that the prototype accurately reflects the view of Fig. 2, which is the heart of the design. The Prograph method (procedure) which implements this is shown in Fig. 5. The dataflow in Prograph is down the

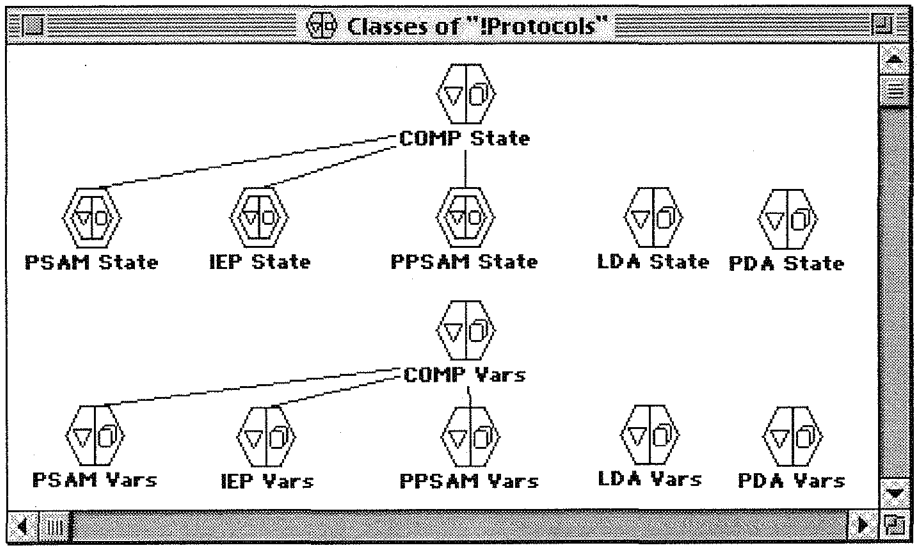


Fig. 4. The window defining the types of components of the standard.

page (data enters the top of operations and leaves from the bottom). We have placed the operations so that the whole method sends the right visual cues, with the IEP on the left, the LDA in the middle and the PPSAM on the right. This method is parameterised over the IEP and PPSAM, which come in as pairs of (State, Vars) at the input bar at the top, and return their new States via the output bar at the bottom.

Following the principle laid out in Sect. 3, we have introduced an extra level of representation here, for Fig. 2 includes more detail than Fig. 5. The latter is merely the Prograph equivalent of our general notion of a protocol as given in Fig. 1. We can now see that it is useful to be able to view the transaction without the details. Omitting the details also makes the animation of this method (when the system dynamically illustrates the operations being fired) clear.

Two aspects of the system have been made visible, which could not be seen in the standard. The first is concerned with aborting transactions. The notation in Prograph to indicate that an operation might fail, is to add an $\boxed{\text{X}}$ on the right. We can thus see that the LDA is in control of the transaction, since it is the component that can report failure (this is because it is the LDA that looks at the Completion Codes, and decides on the actions).

The second aspect that has been made visible in our rendering of the protocols, but not in the standard is the explicit threading of state from one step of the protocol to the next. The vertical lines connecting the steps to A3, from A3 to A5 and from A5 carry the permanent *State* and the ephemeral *Vars* of the IEP. Similarly for the LDA in the middle and the PPSAM on the right.

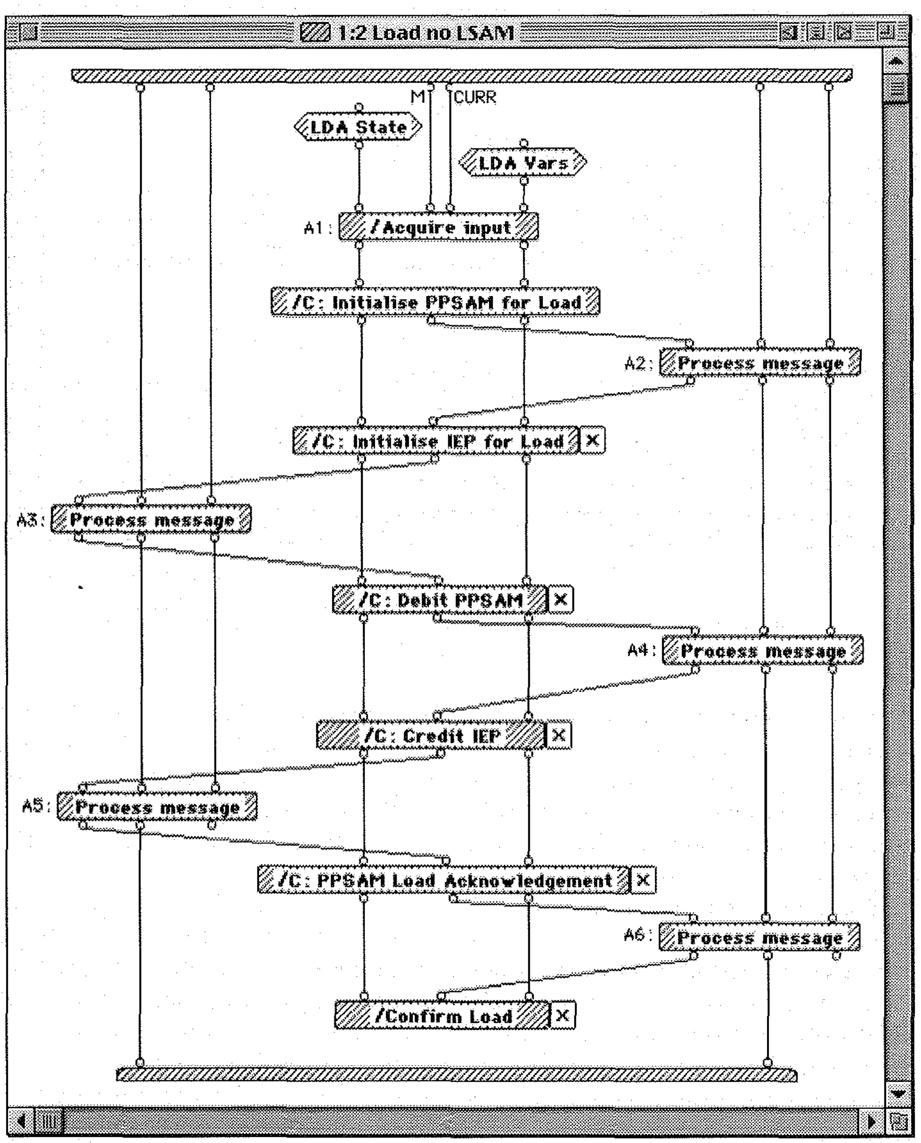


Fig. 5. The method which defines the Load no LSAM transaction protocol.

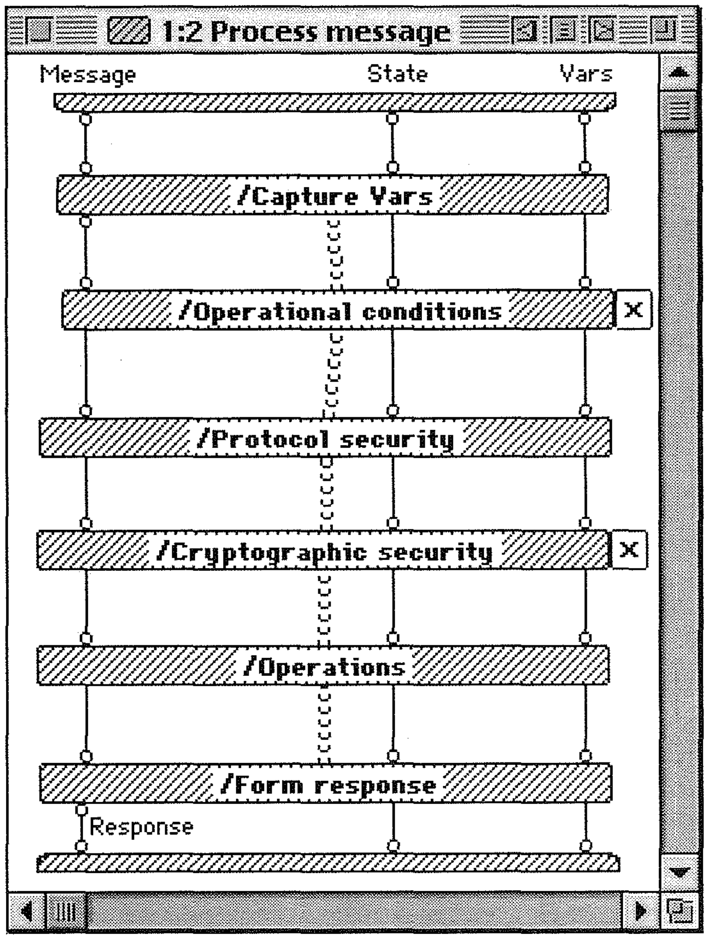


Fig. 6. Process message method conforming to the refined substep structure of each step in the protocol.

Having looked at the transaction at the level of the message and component state flow, we now wish to examine the processing of the individual messages. Looking inside the *Process message* operation of Fig. 6, we find that it follows exactly the structure of Sect. 3. We can also see clearly where the processing of a message might cause failure of the step. The data flow edges connecting the boxes indicate that in addition to the permanent *State* and ephemeral *Vars* the incoming message is also threaded from sub-step to sub-step. This is to tell each of the methods contained in the sub-steps to which message to respond.

We have used Prograph *synchros* (the connections made from the letter *u*) to ensure that we know the order of operation execution, in this case so that, for example, security is checked before any operations take place.

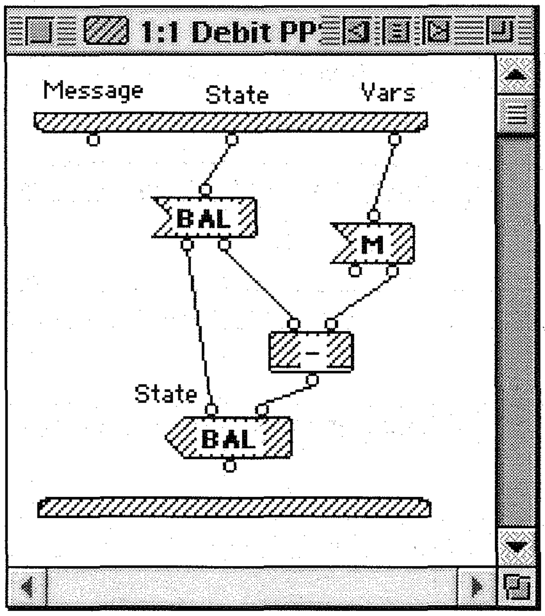


Fig. 7. The Operations method for the PPSAM in response to Debit PPSAM.

Continuing down through the structure, we now look at the details of processing a particular message, *A4: Debit PPSAM.* As a simple example, opening the *Operations* operation for this message, we see (Fig. 7) the actual operation BAL_{PPSAM}:=BAL_{PPSAM} − M_{LDA} from the one page transaction overview of Fig. 2. The specially-shaped operations are Prographs way of accessing components of structures. The two at the top of the method access the BAL_{PPSAM} and M_{LDA} components (which leave on the right hand outputs) and the one at the bottom sets the BAL_{PPSAM} attribute to the new value. The rectangular box labelled with the − sign performs the subtraction.

As another example of looking deeper into the structure, we show the *Cryptographic Security* operation (again of the PPSAM in response to Debit PPSAM). This operation deals with two signatures: it checks S_1 and creates S_2. The dataflow view on this is provided in Fig. 8. This shows that both operations need access to the ephemeral and permanent states, but not to the message. At this level we do not see exactly what data elements are required, which would be the purpose of the next level down, that is the methods *Check S1* and *Create S2.* Checking signature S_1 may fail. The creation of S_2 cannot fail, but instead its

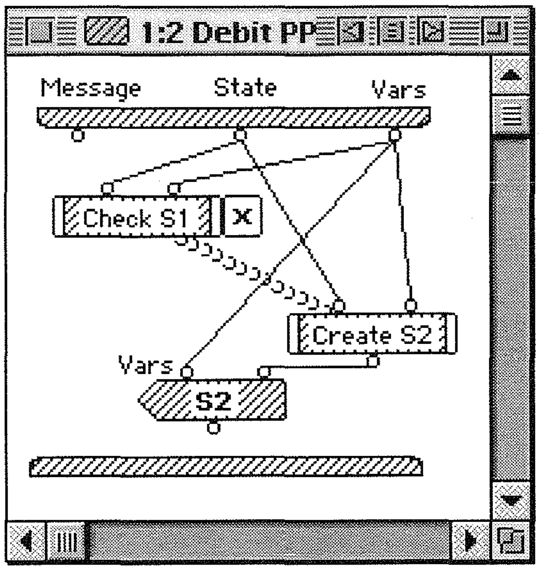


Fig. 8. The Cryptographic Security method for the PPSAM in response to Debit PP-SAM.

value must be recorded in the ephemeral state. Because of the use of locally defined operations (with white bars at the sides) in the dynamic system, it is easy to name the more complex operations. We can also see that it is independent of the cryptographic system being used.

Looking inside the *Create S2* local shown in Fig. 9, we find that it has been possible to separate the signature processing into two steps. The first is to gather all the values to be put into the signature, as well as any keys and algorithms from the component (these are in $\overline{\text{IK}}_{\text{PPSAM}}$). The second is then to apply the cryptographic routines of the system being used. In this way we have provided cryptographic neutrality by encapsulating the cryptographic details as tightly as possible.

The final method to shows the start of the cryptographic system specific part, which is where methods for dealing with each of the signatures must be provided for each cryptographic system. Thus Fig. 10 shows the *Make S2* details for DES. It is interesting to note that it is here that the decisions of the cryptographer are documented, which are frequently used to optimise some calculation. In this case there are two such decisions, which our Prograph rendering of the protocol brings to the fore.

The first is the choice to create the session key ($\mathbf{KSES}_{\text{PPSAM}}$) once and to use it twice: to check S_1 and to create S_2. Figure 10 shows that the session key $\mathbf{KSES}_{\text{PPSAM}}$ is simply picked up from the ephemeral state. The code for *Check*

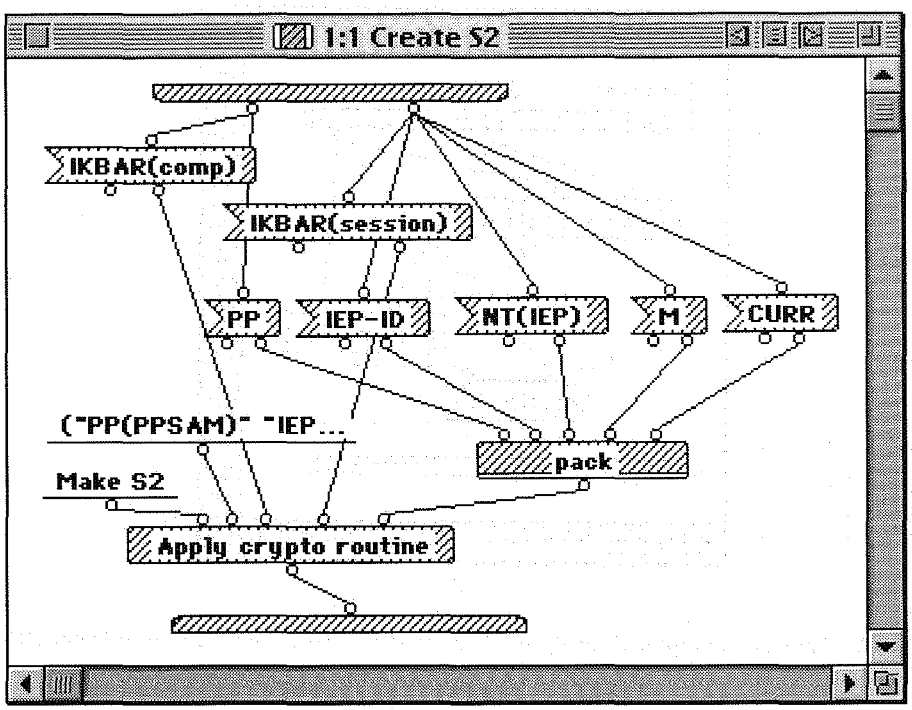


Fig. 9. The Create S2 local from the Cryptographic Security method for the PPSAM in response to Debit PPSAM.

S1 (not shown) puts the session key into the ephemeral state.

The second decision is to sign only M_{LDA} and $CURR_{LDA}$, instead of the data PP_{PPSAM}, IEP, NT_{IEP} and $\overline{M}_{LDA}$, as suggested at step A4 in Fig. 2. (Remember that the notation $\overline{M}_{LDA}$ stands for the pair M_{LDA} and $CURR_{LDA}$). Omitting the data is permitted, as they have been used in computing the session key. One would have to study the details of Fig. 3 quite carefully to see what is happening. In the Program code omitting the first three elements of the list is shown as an explicit discarding using *split-nth*.

The final point about building a prototype in an object oriented system is that certain mistakes are easily found. Consider the code labelled 'Verify IEP signature' in Fig. 3. We have shown in bold face the data that did not appear in the one page overview. As could be expected, all data to do with DES, such as $\mathbf{KSES}_{PPSAM}$ appear in bold face. Such data could not appear in the one page overview of Fig. 2, as that would have made crytographic neutrality impossible. The appearance of $\mathbf{CURR}_{IEP}$ is surprising, for it is a data element belonging to the IEP component. The code labelled A4 is part of the PPSAM. There are at least two possibilities. Firstly it could be that the standard contains a

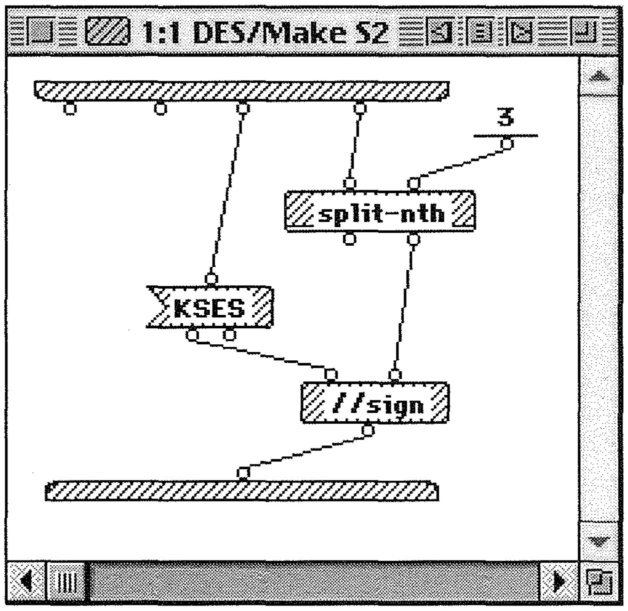


Fig. 10. The Make S2 method from DES Encryption system.

mistake. Secondly this could be an optimisation relying on the fact that at step A4, $\mathrm{CURR}_{\mathrm{LDA}}$ and $\mathbf{CURR}_{\mathrm{IEP}}$ represent the same value. The construction of the conformant prototype of the standard has revealed a number of such issues. Their description is beyond the scope of this paper, as it would require us to present more detail of the standard.

6 Related Work.

Protocol structure is generally discussed in terms of the language used to express the protocols. At one end of the spectrum we find general purpose programming languages. They are often used to specify and or implement protocols. This provides little opportunity for appropriately structuring the protocols. At the other end of the spectrum one finds model-based protocol specifications [2, 4]. Widely known formal languages such as LOTOS and Estelle provide a high level abstract view of protocols and impose structure on the protocols. It is interesting to note that development of both LOTOS and Estelle were part of the standardisation efforts of OSI.

Abbot and Peterson [1] argue the necessity to avoid both extremes. This has some elements in common with our work. Their object oriented language Morpheus is designed for protocol specification. The key idea of Morpheus is to provide a fixed number of so-called shapes, as building blocks for protocols.

The shapes are similar to our components. A shape is a sub class of an abstract super class. Code and data reuse of the abstract class and of the shape make it easy to create specific instances of shapes. These instances are then used as building blocks for a protocol. Morpheus allows only a number of predetermined shapes (a multiplexor, a worker and a router). Abbott and Peterson argue that this provides their Morpheus compiler with more scope for optimisation. Our conformant prototyping system does not need to deliver high performance and offers more flexibility by using a general purpose programming language. Our protocols are represented using only a fixed set of classes, which are used in a disciplined fashion. Abbott and Peterson note that protocols from standards are generally difficult to represent because of their concern for efficiency. In fact they represent the functionality of a standard protocol without necessarily adhering to its exact syntax. Our rendering of the transaction protocols from the standard is exact in that sense. We have prototyped every detail of the load purse transaction as specified in the standard. Admittedly, we are not aiming to specify arbitrary protocols, but a more restricted family of protocols.

Many tools have been developed to study complex protocol behaviour. The Interrogator is an automatic protocol security analysis tool [10]; the language Argos [8] supports automatic protocol validation using a mix of depth-first and breadth-first search for particular scenarios; the Protean tool [3] is designed to detect deadlock in complicated protocols specified as petri nets. The emphasis in all these systems is on understanding and reasoning about the dynamics of complex protocols. The understanding is achieved by animations and simulations of the protocols. The Proteon system, for example, produces animated displays of the history of a protocol, offering the user the opportunity to view a message flowing on an arc, and even to change the message. Such facilities are also offered by our system, because they are provided by Prograph. Our prototype thus offers sophisticated functionality at no extra effort.

Many authors who work with protocols note that the lack of structure in the protocols causes them problems. Billington et al [3] and Bochmann et al [13] discuss the possibility of separating normal operations from exceptional conditions, and thus enriching the structure of protocols. Tel [12], in the context of proving properties of distributed algorithms in general and protocols in particular, suggests a number of simple rules that enhance the structure of a protocol. Tel also suggests that properties of protocol skeletons may carry over to fully-populated protocols, provided certain precautions are taken.

Our restricted setting of the authentication protocols has made it possible to separate out exceptional from normal operations, and also to separate out the security operations. This gives an extra level of structure in our protocols. To our knowledge this has not been achieved elsewhere.

7 Conclusions.

The standard way of visualising protocols using pictures with boxes (representing operations) and arrows (representing messages) is appropriate for a global study of a protocol. The standard visualisation method is insufficient to study the protocols in detail, the problem being that the structuring of the protocols relies on elements not explicit in the standard visual rendering. Such elements include the security and the state of the interacting components. To render the structure properly one should visualise not only the operations and the messages but also the state and the security.

Using the data flow model makes it possible to explicitly render the state manipulations implied by the protocols. Using an object oriented design methodology makes it possible to be selective about the state elements that are accessed and updated at various points in the protocols. A system combining both models is thus appropriate to deal with the manipulation of state in the protocols.

Security in an abstract sense relies on cryptographic systems. In a concrete sense it is the encapsulation of the cryptographic data and operations that determines whether the system is robust or fragile. We have shown how the separation of steps in the protocol into a number of judiciously chosen sub-steps separates out the important protocol, security and operational aspects. The design method that we have used supports this subdivision of larger into smaller steps. It has pointed out where the standard may be unclear. This indicates that building a highly structured prototype and using an object oriented data flow model which supports the encapsulation of data and operations gives a high level of confidence in the robustness of the system.

The combination of data flow, object orientedness and visual programming is provided by the Prograph language that we have been using to build a prototype of some of the protocols in the load purse transaction of the CEN Inter-sector electronic purse draft standard. The Prograph prototype makes it possible to animate the protocols and to perform simulations and experiments with the protocols. This feedback of the system to the designer is a powerful design tool.

8 Acknowledgements.

We thank Hugh McEvoy and the referees for their comments on a draft version of the paper. Jon Hallet provided assistance with the rendering of the screen shots.

References

1. M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. In *Communications architectures & Protocols (SIGCOMM)*, pages 27–38, Baltimore, Maryland, Oct 1992. ACM Computer communication review, 22(4).
2. M. S. Atkins. Experiments in SR with different upcall program structures. *ACM transactions on computer systems*, 6(4):365–392, Nov 1988.
3. J. Billington, G. R. Wheeler, and M. C. Wilbur-Ham. PROTEAN: A High-Level petri net tool for the specification and verification of communication protocols. *IEEE transactions on software engineering*, SE-14(3):301–316, Mar 1988.

4. A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. BEST: A network simulation and prototyping testbed. *CACM*, 33(10):64–74, Oct 1990.
5. CEN European Committee for Standardization. Identification card systems – inter–sector electronic purse. Draft standard prEN 1546, European Committee for Standardization, Brussels, Nov 1995.
6. F. R. Giles, P. T. Cox, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Workshop on Visual Languages*, pages 150–156, Rome, Italy, Oct 1989. IEEE Computer society press, Washington.
7. P. H. Hartel and E. K. de Jong Frz. Towards testability in smart card operating system design. In V. Cordonnier and J-J. Quisquater, editors, *1st Smart card research and advanced application conference (CARDIS 94)*, pages 73–88, Lille France, Oct 1994. Univ. de Lille, France.
8. G. J. Holzmann. Automated protocol verification in *argos* : Assertion proving and scatter searching. *IEEE transactions on software engineering*, SE-13(6):683–696, Jun 1987.
9. J. R. McGraw. The VAL language: Description and analysis. *ACM transactions on programming languages and systems*, 4(1):44–82, Jan 1982.
10. J. K. Millen, S. C. Clark, and S. B. Freeman. The Interrogator: Protocol security analysis. *IEEE transactions on software engineering*, SE-13(2):274–288, Feb 1987.
11. S. Stepney. *High integrity compilation: A case study.* Prentice Hall, Hemel Hempstead, England, 1993.
12. G. Tel. *Topics in distributed algorithms.* Cambridge Univ. Press, Cambridge, England, 1991.
13. G. v. Bochmann and J. P. Verjus. Some comments on "Transition-Oriented" versus "structured' specification of distributed algorithms and protocols. *IEEE transactions on software engineering*, SE-13(4):501–505, Apr 1987.
14. A. H. Veen. Dataflow machine architecture. *ACM computing surveys*, 18(4):365–396, Dec 1986.

This article was processed using the LaTeX macro package with LLNCS style

Using Formal Methods to Cultivate Trust in Smart Card Operating Systems

Marjan I. Alberda[1,2], Pieter H. Hartel[2,1], Eduard K. de Jong Frz[3]

[1] Fac. of Math., Computer Science, Physics and Astronomy, Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, Email: alberda@fwi.uva.nl
[2] Dept. of Electronics and Computer Science, Univ. of Southampton, S017 1BJ Southampton, UK, Email: phh@ecs.soton.ac.uk (corresponding author)
[3] Integrity Arts Inc., San Mateo, USA, Email: edejong@integrityarts.com

Abstract. To be widely accepted, smart cards must contain completely trustworthy software. Because smart cards contain relatively simple computers, and are used only for a specific class of applications, it is feasible to make the language used to program the software components focussed and tiny. Formal methods can be used to precisely specify this language and to reason about properties of the language, which results in more trustworthy software. We explore this process by specifying the core of a proprietary systems programming language for smart card operating systems. We show how the specification obtained is used in proofs, and in the development of tool support.

Keywords. smart card operating systems, structural operational semantics, software components, proofs, tool support.

1 Introduction

The ITSEC requirements [8] state that formal methods must be used in the construction of IT systems that are to be endowed with maximal trust. The term Formal Methods describes a mode of operation in the design, specification and implementation of a system which is characterised by the application of formalised methods of reasoning. Formal methods:

- Help to clarify the exact requirements of a system.
- Help to detect mistakes and errors early.
- Make it possible to prove properties about a system.
- Make it possible to prove the correctness of an implementation.

We present a formal specification of the core of one of the components of the Tosca smart card system as it is currently being developed by Integrity Arts Inc. It is our aim to demonstrate in this paper the benefits of using a formal approach in a setting that is accessible to an audience that is not specialised in Formal Methods.

We give a detailed specification of the core of the proprietary systems programming language Clasp (Compact Language for Abstract Secure Processors) that plays a key role in the Tosca system. The Clasp language is designed for secure execution and to generate a dense object code. The security aspects of the language will be highlighted in this paper, space precludes us from exploring its density properties.

Formal specifications of languages are not novel. An excellent introduction to the subject may be found in Nielson and Nielson [10]. To our knowledge formal specifications as described in this book have not been used to help design and reason about smart card systems. This is the first contribution of this paper. The second contribution is that we indicate how a formal specification may be used to help develop support tools for the development process of a smart card system. We identify tools such as emulators, simulators and animators and give a code fragment from our simulator. The key role of a formal specification in producing these support tools is explained.

In the next section, we discuss the general design of the Tosca system as a layered software architecture. In Section 3, we present an informal description of the core of Clasp, which is then followed by a formal specification. Section 4 describes how properties of the formal specification can be formulated and proved. It also describes how tools can be built to create the structural parts of simulators, emulators, animators and the actual implementation of components of a smart card system. This section also includes future work. Section 5 describes related work and the final section presents our conclusions.

2 A Layered Software Architecture

Programs in smart cards are usually stored in ROM. This ensures that the programs cannot easily be tampered with. Unfortunately, this also makes smart cards inflexible, because the ROM cannot be changed after fabrication. To overcome this limitation, smart card operating systems need to support the downloading of executable code. To maintain the operating system integrity, the loadable executable code should be contained in recognizable sections of application code [11, 1], which are usually referred to as 'applets'. In Tosca, the applet code is executed by a carefully crafted interpreter that makes sure that the applet code behaves properly. Later, we make more precise what 'proper behaviour' means. The provision of a downloadable applet facility makes smart cards flexible, without hampering the security.

The same idea has been used to build flexible Web browsers using Java [6]. Java enables applets to be downloaded from the server and to be executed by the client browser. Java code is compiled by the server into a byte code, which is subsequently interpreted. The Java interpreter can check that the downloaded applets behave properly.

A smart card is not a Web browser, so that although in principle similar approaches can be followed, the practice is different. For one thing, smart cards are small. Web browsers may assume that there are megabytes of memory available. The specially designed Tosca smart card application language is to smart cards what Java is to the Web browser: application code in Tosca is compiled into loadable applets.

To cope with the limitations of smart cards, the Tosca interpreter is itself written in an interpreted code, called Clasp. This increases the code density and improves the level of trust beyond what is possible with a single level of interpretation. The time penalty incurred by using multiple levels of interpretation is not a problem for smart cards. A typical smart card transaction lasts for about one second and the only compute intensive aspect is associated with the cryptography. By coding the compute intensive parts of cryptographic operations at the lowest level (directly in machine code), and by making these operations available as instructions at a higher level of interpretation, smart card systems can be efficient and secure.

Tosca programs	compile into	Tosca-object code for the Tosca-object interpreter (written in Clasp)	Level 2
Clasp programs	compile into	Clasp-object code for the Clasp-object interpreter (written in Assembler)	Level 1
Assembler	generate	machine code for the smart card CPU (built in hardware)	Level 0

Fig. 1. Layered smart card software architecture.

Multiple levels of interpretation are standard practice in computer architecture [14]. A typical CISC machine would have a micro code program that interprets machine instructions. The operating system would interpret system calls, and a user program (such as the command language interpreter) might interpret a further set of instructions. The point here is that the problems that smart card software architects are facing can be solved using tried and tested ideas such as the multi level abstract machine and client/server approaches.

Tosca's multi-layered software architecture is schematically shown in Figure 1. The Tosca development environment allows for the creation of trustable smart card applications on the basis of the Tosca operating system, standard applets and customer applets. The Tosca language is used to create the applets. Tosca applets are compiled into Tosca-object, which is then interpreted by the Tosca-object interpreter.

The Tosca-object interpreter is written in Clasp, as are the standard library modules that are supplied with this interpreter. The Clasp language is a threaded code, interpreted language. It has been designed specifically for the provision of trustable code. It can be compiled into a dense and efficient interpreted code called Clasp-object.

Clasp-object interpreters are small assembler programs, written specifically for typical IC card processors, such as the 8051 and the 6805.

In the rest of this paper, we will focus on particular aspects of the Clasp language.

2.1 The Compact and Secure Clasp Language

The Clasp programming model is that of a stack machine. It works in the same way as FORTH in that it shares with FORTH the ability to extend the language. However, unlike FORTH, Clasp provides security and dense code, but does not support compiled routines and run-time dictionary lookup.

A Clasp program consists of a number of declarations, of which we shall only discuss the procedure declarations. Each procedure has a name and a body, there are no explicit parameters to the procedures. Arguments are passed implicitly via the stack. Return values are left on the stack. This makes programming in Clasp a bit awkward, but Clasp extensions have been developed to help overcome this limitation. A future extended definition of the language is under construction to provide syntactic support in this aspect. We shall not discuss these extensions here, and we shall not even look at procedure calls. Instead, we will focus on how the body of a procedure is executed. A body consists of a number of control statements and stack operators. Constructs are separated by semi-colons as in Pascal. The control statements and the stack operators communicate exclusively via the stack. The stack is kept in the RAM of the smart card, which therefore limits the amount of space that can be used for the stack.

Let us concentrate on the basic mode of operation of the Clasp interpreter: it executes one statement or stack operator at a time, manipulating the stack as appropriate. Some important types of Clasp statements and operators are:

- The statement: n
  pushes the value n onto the stack. Values are bytes so they must be in the range $0 \ldots 255$.
- The statement: **test** *true part* **else** *false part* **end**
  tests the value that resides on the top of the stack. If this value is 1 the statements and operators of the *true part* are executed. The *false part* is executed if the value is 0.
- The operator: *pickn*
  duplicates a particular element of the stack.
- The operator: *dropn*
  removes a certain number of elements from the stack.
- The operator: *rotn*
  circularly rotates a particular portion of the stack.
- The operators: $-$, $<$
  replace the top two elements of the stack by a newly computed result.

This summary description shows two things. First, the Clasp language may seem rather a poor language. We should point out that we have not shown the complete language. For instance, I/O and memory operations and in particular the loop construct and procedures have not been described. We do not need those for the example that we will be discussing in detail. Furthermore, the description of these concepts is not difficult to add. Adding loops and procedures does not increase the number of different language constructs significantly. The complete language is kept small and focussed which makes it easier to build language processors and to reason about programs written in the language.

The second point to make is that the summary description above is also vague in many ways. This is typical for informal language descriptions. It may be a source of considerable frustration both to the compiler writer and to the programmer, as neither knows exactly what the language designer intended, and problems created by such vagueness are in general difficult to address. In a trusted environment this vagueness cannot be tolerated, and a formal description of the programming language is called for.

2.2 The Syntax of a Representative Subset of Clasp

The syntax of a programming language is usually described with a formal language, e.g. based on BNF, or the pictorial form of BNF: syntax diagrams. By using appropriately chosen names for the describing rules, the syntax may indicate what particular language constructs mean. It does not enforce that meaning, the names serve only as a suggestion. The formal semantics of the language, defined in addition to the syntax, is to provide that. Working with a semantics may seem difficult and mathematical at first, because of the use of special symbols. Once adapted to the particular notation, using formal methods feels like programming.

$$
\begin{array}{rcll}
n & ::= & 0|\ldots|255 & (\text{ numbers }) \\
s & ::= & n & (\text{ push number onto stack }) \\
 & | & \textbf{skip} & (\text{ null statement }) \\
 & | & \textbf{test}\ s_t\ \textbf{else}\ s_f\ \textbf{end} & (\text{ test value on top of stack }) \\
 & | & s_1; s_2 & (\text{ statement composition }) \\
 & | & \mathit{pickn}|\mathit{dropn}|\mathit{rotn} & (\text{ stack operators }) \\
 & | & -|< & (\text{ arithmetic operators })
\end{array}
$$

Fig. 2. Abstract syntax of a subset of Clasp.

Figure 2 shows the abstract syntax of the subset of Clasp as described earlier. The syntactic category n represents numbers and s (and also s_1, s_2, s_t and s_f) represent statements and stack operators.

It is customary to use an abstract rather than a concrete syntax for reasons of brevity. The difference between the two forms of syntax is that in an abstract syntax parentheses indicate how constructs are parsed. In a concrete syntax this information is encoded in the rules, which are therefore slightly more complicated. This rule encoded parsing information is not relevant for the description of the meaning of the constructs, and can be safely ignored.

Here is an example of a Clasp code fragment, with parentheses to enforce a correct parse:

$$s = 5; (3; (2; (\mathit{pickn}; (2; (\mathit{pickn}; (<; \textbf{test skip else } 2; \mathit{rotn} \textbf{ end}))))) \quad (1)$$

It is possible to check that the statement above satisfies the syntactic requirements of Clasp. The meaning of the code fragment can only be guessed, which is not satisfactory. The next section introduces the formal aspects of describing the meaning of a code fragment such as that of (1).

3 Towards a Formal Description of the Clasp Subset.

In this section, the execution of the statement sequence in Clasp is presented and explained using English and diagrams (section 3.1). Semantic rules are introduced (section 3.2) and the derivation sequence of the Clasp example is presented (section 3.3). The rules are modified and extended with the statement composition rules (section 3.4). Finally, tags are added to distinguish between correct and incorrect behaviour (section 3.5). Tables 2 and 3 summarise the final rules.

The application of formal rules makes an execution trace more readable, reduces ambiguity inherent to vagueness and makes a language description compact.

3.1 Executing a Clasp Statement Sequence.

This section presents the execution of a Clasp statement sequence using English and diagrams. It will show how elaborate an explanation can become when using these techniques instead of formal methods used later on.

Consider the statement sequence (1) again:

$$5; 3; 2; \mathit{pickn}; 2; \mathit{pickn}; <; \textbf{test skip else } 2; \mathit{rotn} \textbf{ end} \quad (2)$$

The sequence consists of 8 statements separated by the operator ; . Brackets are omitted, for we assume the sequence is interpreted from left to right. All successive stack configurations in the computation are shown in figure 3. The stack grows downwards. Indexes of the elements are given to the right of the stack.

The first statement in the sequence pushes the number 5 onto the stack (b). The execution of the next two statements results in the numbers 3 (c) and 2 respectively pushed onto the stack (d). After these steps, the sequence of statements is reduced to

$$\mathit{pickn};\ 2;\ \mathit{pickn}; <; \textbf{test skip else } 2; \mathit{rotn} \textbf{ end} \quad (3)$$

The statement *pickn* pops the stack, updates the index (e), and pushes a copy of the element with the index given by the popped element, that is, index 2, value 5 (f). The execution continues by pushing a 2 onto the stack (g). Again, the *pickn* statement pops the stack and updates the index (h). It pushes a copy of the element with index 2 (the popped element) onto the stack, i.e. the number 3 (i). The next statement, $<$, compares the top two elements of the stack. It pushes a 0 (representing false) onto the stack if the top element (with index 1) is greater than or equal to the element with index 2, otherwise it pushes a 1 (representing true). In this case $<$ pushes a 0 for $5 \not< 3$ (j). The sequence of statements is now reduced to

$$\textbf{test skip else } 2; rotn \textbf{ end} \tag{4}$$

The **test** statement resembles an *if...then...else* construct, but it has no boolean expression. Instead, the boolean on which the decision is based as to which branch to take, must have been pushed onto the stack before a **test** statement is encountered. The $<$ statement in the sequence can be considered as the boolean expression in an *if...then...else* construct. The **test** statement inspects the top of the stack, finds a 0, pops it and subsequently executes the statements after the **else**. It results in stack configuration (k) and the sequence (2;*rotn*) to be

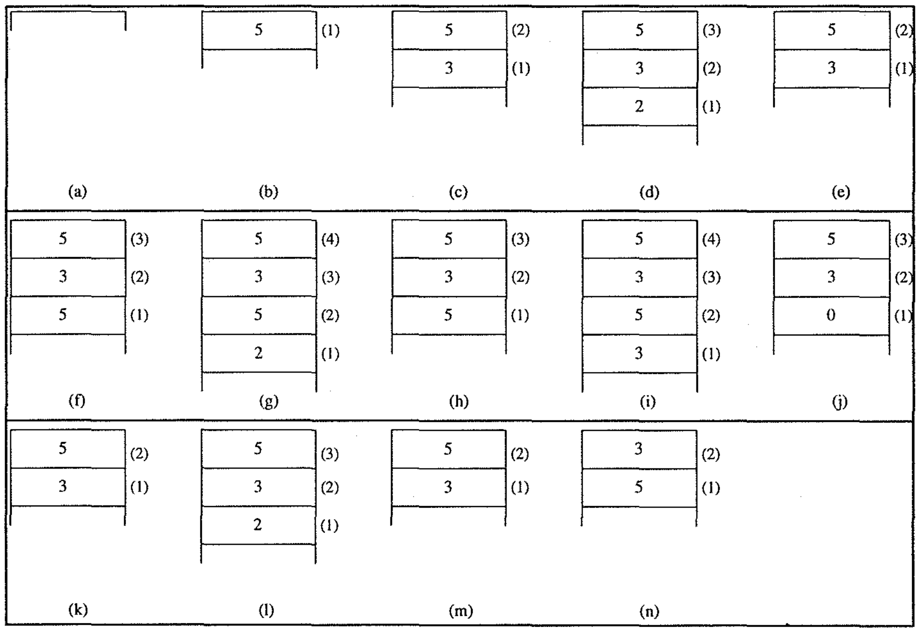


Fig. 3. Successive configurations of the stack in the execution of (2).

executed. A 2 is pushed onto the stack (l). Like *pickn*, *rotn* pops the stack and updates the index (m). Then, it swaps value 5 (index 2) and value 3 (index 1) (n).

Execution of the statement sequence results in stack configuration (n). If the initial statements of the sequence had been 3;5 instead of 5;3, the $<$ would have pushed a 1 instead of a 0. The **test** statement would therefore have resulted in the execution of **skip**. This would have delivered the same final stack configuration (n), for **skip** does not change the stack.

The execution of the sequence

$$2; \mathit{pickn}; 2; \mathit{pickn}; <; \textbf{test} \;\; \textbf{skip} \;\; \textbf{else} \;\; 2; \mathit{rotn} \;\; \textbf{end} \tag{5}$$

will result in a stack with its top element larger than or equal to the element with index 2.

3.2 Introducing the Formal Rules.

Instead of using pictures to illustrate the configuration of the stack and using English to explain what happens, formal rules can be used to specify the meaning of a sequence of statements more succinctly.

A rule has a standard formal notation. It consists of

- a tuple $\langle \mathit{statement}; \mathit{statements}, \mathit{stack}\rangle$, which represents a state in the execution,
- an arrow $\Rightarrow$ to indicate a transition from this state to a second tuple,
- $\langle \mathit{statements}, \mathit{stack}'\rangle$, the next state in the execution.

The concatenation ($\mathit{statement}; \mathit{statements}$) defines a sequence of statements, in which *statement* is the first statement. It is followed by the sequence *statements*. Just as n is used to denote any number and s to denote any sequence of statements, u (user stack) is used to denote any stack configuration (including the empty stack). The stack grows from right to left, and stack elements are concatenated using the operator : . The symbol $\emptyset$ explicitly indicates the empty stack. The notation ϵ specifies the empty statement sequence.

Consider the rule for the statement n:

$$\langle n; s, u\rangle \;\Rightarrow\; \langle s, n : u\rangle \tag{6}$$

The rule states that the execution of the statement n followed by the statements s, with stack configuration u, results in a new state in which statements s are to be executed with the new stack configuration $n : u$. Here n is the new top. As an example consider the execution of the first three statements of sequence (2) in section 3.1:

$$\langle 5; 3; 2, \emptyset\rangle \Rightarrow \langle 3; 2, 5 : \emptyset\rangle \Rightarrow \langle 2, 3 : 5 : \emptyset\rangle \Rightarrow \langle \epsilon, 2 : 3 : 5 : \emptyset\rangle \tag{7}$$

This derivation sequence shows three successive applications of rule (6). The numbers 5, 3 and 2 are pushed onto the stack respectively, resulting in the final stack configuration $2 : 3 : 5 : \emptyset$, in which 2 is the top of the stack and has index

1. All statements are executed. Note that a statement can always be followed by the empty statement (sequence) ϵ, i.e. *statement* $\equiv$ *statement*; ϵ, in which $\equiv$ means semantically equivalent. This implies that ϵ can be omitted. In (7), for instance, $5;3;2 \equiv 5;3;2;\epsilon$. There is no particular reason for omitting ϵ, other than saving space.

A rule's left-hand side is viewed as a template to which the statement sequence to be executed is matched. The semantics is thus syntax directed. The starting state in (7), $\langle 5;3;2,\emptyset\rangle$, and the left-hand side of the rule (6), $\langle n;s,u\rangle$, can be matched by considering $n = 5$, $s = 3;2$, and $u = \emptyset$. Applying the rule results in the state $\langle 3;2,5:\emptyset\rangle$. Note that $2 \equiv 2;\epsilon$ is used to match 2 in $\langle 2,3:5:\emptyset\rangle$ of (7) with the statement sequence $n;s$ in rule (6).

The rule for the statement *pickn* is given by

$$\langle \mathit{pickn}; s, n:u\rangle \Rightarrow \langle s, u[n]:u\rangle \tag{8}$$

The top of the stack on the left-hand side of the rule, n, is used as an index in the stack u. $u[n]$ is the element in the stack with index n and replaces n as the top of the stack.

Consider the following example of using *pickn*:

$$\langle \mathit{pickn}, 1:3:5\rangle \Rightarrow \langle \epsilon, 3:3:5\rangle \tag{9}$$

By considering $n = 1$, $u = 3:5$, $s = \epsilon$ and $\mathit{pickn} \equiv \mathit{pickn};\epsilon$, applying the rule results in the state $\langle\epsilon, 3:3:5\rangle$, as, after reindexing, 3 is the top of the stack u in $n:u$. It has index 1 and so $u[n] = u[1] = 3$. Writing the stack as $n:u$ on the left-hand side of the rule (8) forces the stack to consist of at least one element.

The rule for the statement *rotn* is given by

$$\langle \mathit{rotn}; s, n:u\rangle \Rightarrow \langle s, u[u[n]/1, u[1]/2, \ldots, u[n-1]/n]\rangle \tag{10}$$

This rule introduces a new notation for stack manipulations. A stack element at index i to be substituted by a value v is denoted by $u[v/i]$; a number of simultaneous substitutions may be denoted by including them as a comma separated list inside the square brackets. Whenever the value v is defined as an element of u, it is extracted from u before any of the substitutions has taken place. The resulting stack in (10), denoted by the outer $u[\ldots]$, is obtained by simultaneously substituting each element with index $1 \leq i \leq n$, by its index-wise predecessor in the initial stack, denoted by the inner $u[i-1]$, and the top by $u[n]$. In effect, all elements with index $i < n$ are rotated one place downwards in the stack and the element with index n is the top of the new stack.

When $n = 2$, the rule defines a swap function, as was shown in section 3.1. The rule precisely defines the behaviour of the statement *rotn* for all n (i.e. for all 'correct' n, for stack underflow and overflow are not taken into account, yet). As an example, consider:

$$\langle \mathit{rotn}, 3:1:2:3:4\rangle \Rightarrow \langle\epsilon, 3:1:2:4\rangle \tag{11}$$

After reindexing, $n = 3$, $u[3] = 3$, $u[1] = 1$, and $u[2] = 2$, and $u[u[3]/1, u[1]/2, u[2]/3]$ results in the final stack configuration.

The **test** statement has two possible executions. If a 0 resides on top of the current stack, the *false part* after the keyword **else** is executed. If the current top has value 1, **test** continues by executing the *true part*. Two rules define the behaviour of the statement **test**:

$$\frac{b \;=\; 1}{\langle \textbf{test}\ s_t\ \textbf{else}\ s_f\ \textbf{end}; s, b : u\rangle \;\Rightarrow\; \langle s_t; s, u\rangle} \qquad (\textbf{test}^{True})$$

$$\frac{b \;=\; 0}{\langle \textbf{test}\ s_t\ \textbf{else}\ s_f\ \textbf{end}; s, b : u\rangle \;\Rightarrow\; \langle s_f; s, u\rangle} \qquad (\textbf{test}^{False})$$

The rules introduce the notion of a precondition, or premise. A premise is a condition on the transition, i.e. the transition can only take place if the premise is true. The premise is written above the horizontal line and the actual transition from state to state, or derivation step, is written below it. By writing $b : u$ as the stack configuration on the rules' left-hand sides, the top of the stack, b, can be used in the premise. If $b = 1$, the rule named ($\textbf{test}^{True}$) matches, execution continues by executing $s_t; s$. If $b = 0$, rule ($\textbf{test}^{False}$) is applied, and the sequence $s_f; s$ is to be executed. The rules are mutually exclusive but not complete, as the behaviour in case $b > 1$ is not defined. This will be amended in section 3.5, but for now, it is assumed that the boolean b is either 0 or 1.

The statement $<$ also has two possible execution paths. Depending on the outcome of the comparison of the top two elements of the stack, either a 0 or a 1 is pushed onto the stack. Having already introduced the notion of a premise, the rules for the statement $<$ are included (see table 1) without further explanation.

The rule for the statement **skip** is given by

$$\langle \textbf{skip}; s, u\rangle \;\Rightarrow\; \langle s, u\rangle \qquad (12)$$

The execution of **skip** does not affect the stack. If a **test** statement was to be defined with only one execution path, for instance, **skip** can be used to constitute the empty statement sequence as the other path.

Table 1 summarises the named rules introduced in this section.

3.3 Execution of the Clasp Statement Sequence Using the Formal Rules.

Consider again the statement sequence (2). Its execution trace is derived by the successive application of the formal rules (table 1) to its constituents. This derivation sequence is given below. Each step is annotated with the rule that

Table 1. The simplified formal rules of the subset of Clasp.

$\langle n; s, u\rangle \Rightarrow \langle s, n : u\rangle$	$(number)$
$\langle \mathbf{skip}; s, u\rangle \Rightarrow \langle s, u\rangle$	$(\mathbf{skip})$
$\dfrac{b \; = \; 1}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}; s, b : uk\rangle \Rightarrow \langle s_t; s, u\rangle}$	$(\mathbf{test}^{True})$
$\dfrac{b \; = \; 0}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}; s, b : u\rangle \Rightarrow \langle s_f; s, u\rangle}$	$(\mathbf{test}^{False})$
$\dfrac{n_2 \; < \; n_1}{\langle <; s, n_1 : n_2 : u\rangle \Rightarrow \langle s, 1 : u\rangle}$	$(<^{True})$
$\dfrac{n_2 \; \geq \; n_1}{\langle <; s, n_1 : n_2 : u\rangle \Rightarrow \langle s, 0 : u\rangle}$	$(<^{False})$
$\langle pickn; s, n : u\rangle \Rightarrow \langle s, u[n] : u\rangle$	$(pickn)$
$\langle rotn; s, n : u\rangle \Rightarrow \langle s, u[u[n]/1, u[1]/2, \ldots, u[n-1]/n]\rangle$	$(rotn)$

was applied.

$$
\begin{array}{rl}
\langle 5; 3; 2; pickn; 2; pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ \emptyset\rangle & \Rightarrow_{(number)} \\
\langle 3; 2; pickn; 2; pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 5 : \emptyset\rangle & \Rightarrow_{(number)} \\
\langle 2; pickn; 2; pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 3 : 5 : \emptyset\rangle & \Rightarrow_{(number)} \\
\langle pickn; 2; pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 2 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(pickn)} \\
\langle 2; pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 5 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(number)} \\
\langle pickn; <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 2 : 5 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(pickn)} \\
\langle <; \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 3 : 5 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(<^{False})} \\
\langle \mathbf{test}\ \mathbf{skip}\ \mathbf{else}\ 2; rotn\ \mathbf{end},\ 0 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(\mathbf{test}^{False})} \\
\langle 2; rotn,\ 3 : 5 : \emptyset\rangle & \Rightarrow_{(number)} \\
\langle rotn,\ 2 : 3 : 5 : \emptyset\rangle & \Rightarrow_{(rotn)} \\
\langle \epsilon,\ 5 : 3 : \emptyset\rangle &
\end{array}
$$

The derivation sequence contains all semantically relevant information in only 11 formal lines. It shows each configuration of the stack and what statements are left to be executed. The derivation sequence shows at least two of the advantages of using formal rules instead of English and pictures, namely compactness and clarity. Compactness and clarity help to make precise the behaviour of an execution or system, and therefore help to detect mistakes and errors.

3.4 Adding the Composition Rules.

In section 3.2, the general form of a rule was defined as follows: $\langle \textit{statement}; \textit{statements}, \textit{stack}\rangle \Rightarrow \langle \textit{statements}, \textit{stack}'\rangle$, i.e. after executing *statement*, and perhaps changing the stack, *statements* are left to be executed. The first statement is of interest, but also the fact that *statements* follow it (though not their precise contents). A different approach from the one used in section 3.2 is to only concentrate on the individual statements. This allows for separation of concerns, bearing in mind the principle "if you can separate then separate". Sequencing and the actual computation are the two concerns here. Separation of the two will lead to shorter derivations and it is more extensible.

The rule for the statement n has been given by $\langle n; s, u\rangle \Rightarrow \langle s, n : u\rangle$. It is now defined as: $\langle n, u\rangle \Rightarrow \langle \epsilon, n : u\rangle$. Hence, we are no longer concerned with the statements that follow n. The final table of section 3 summarises the adapted rules (with tags to be introduced in section 3.5).

A mechanism is needed for the newly defined rules to be applied to a sequence of statements as before. For this purpose, two composition rules are introduced:

$$\frac{\langle s_1, u\rangle \Rightarrow \langle \epsilon, u'\rangle}{\langle s_1;\ s_2, u\rangle \Rightarrow \langle s_2, u'\rangle} \qquad (;)$$

$$\frac{\langle s_1, u\rangle \Rightarrow \langle s_1', u'\rangle \ \wedge\ s_1' \neq \epsilon}{\langle s_1;\ s_2, u\rangle \Rightarrow \langle s_1';\ s_2, u'\rangle} \qquad (;^{step})$$

The first rule states that if the execution of s_1 will lead to a state in which there are no more statements to be executed (ϵ) and in which the new stack is u' (this includes the possibility that u and u' are the same, as would be the case after the execution of **skip**), the statement sequence $\langle s_1; s_2, u\rangle$ continues to $\langle s_2, u'\rangle$, i.e. to a state in which s_2 is to be executed with the new stack configuration u'. The second rule differs from the first rule in that s_1 is now a compound statement and the one step execution does not lead to ϵ, but rather to the statements s_1'. Here, the new stack configuration is also denoted by u' for which same reasoning holds as for u' in the first rule. The statement sequence $\langle s_1; s_2, u\rangle$ now continues to $\langle s_1'; s_2, u'\rangle$, i.e. to a state in which it will continue executing the remaining statements of s_1, named s_1'.

In the simplified rules of section 3.2, *statements* could be empty (ϵ). This was a necessity, for $\langle \textit{statement}; \textit{statements}\rangle$ would otherwise always be an endless statement sequence. However, if $s_1' = \epsilon$ in the rule $(;^{step})$, the premises of the two composition rules are exactly the same and one state, namely $\langle s_1; s_2, u\rangle$, has two possible transitions! The new form $\langle \textit{statement}, \textit{stack}\rangle$ allows us to go back to our original syntax definition in section 2.2 which excludes the possibility of *statement* being empty. Now, ϵ is the only possible empty statement sequence.

To summarise, rules are now of the form $\langle \textit{statement}, \textit{stack}\rangle \Rightarrow \langle \epsilon, \textit{stack}'\rangle$, and the composition rules were added to be able to reason about a sequence of statements. We have separated sequencing from doing the stack operations.

3.5 Adding tags.

So far, it was assumed that statements that affect the stack can always be executed. In an implementation, this cannot be assumed. When using a stack, two kinds of errors may occur: stack overflow and stack underflow. The first occurs when a statement pushes data onto a full stack, or when the stack is addressed with an index that is larger than its size. Here, (*number*) is the only rule that pushes data onto the stack. Several other rules may use an 'incorrect' index. A mechanism is needed to trap this error. The other error, stack underflow, occurs when an empty stack is popped, or when the stack is addressed with an index less than 1. Several rules may cause this error by using an 'incorrect' index. The rule (<) compares the top two elements of the stack, but what happens when the stack is only one element large or even empty? A mechanism must be present to handle these errors, for rules must always be precise.

For this purpose, the notion of a tag is introduced, and each state is extended with a tag. Consequently, the general form of the rules is: $\langle \mathit{statement}, \mathit{stack}, \mathit{ok}\rangle \Rightarrow \langle \epsilon, \mathit{stack}', \mathit{tag}\rangle$, in which *tag* is either *ok* or *nok* (*n*ot *ok*). Note that the tag on the left-hand side is always *ok*. If $\mathit{tag} = \mathit{nok}$ in the current state, no rule should match, for, in that case, an error has occurred and execution should be stopped (an exception handling mechanism is provided in Clasp, but limitation of space precludes us from explaining that here).

The execution of statement n is now defined by two rules: one defining its behaviour when no stack overflow will occur, and one defining it when it will:

$$\frac{|u| \ < \ \mathit{max}}{\langle n, u, \mathit{ok}\rangle \ \Rightarrow \ \langle \epsilon, n : u, \mathit{ok}\rangle} \qquad (\mathit{number}^{ok})$$

$$\frac{|u| \ \geq \ \mathit{max}}{\langle n, u, \mathit{ok}\rangle \ \Rightarrow \ \langle \epsilon, u, \mathit{nok}_{\mathit{full}}\rangle} \qquad (\mathit{number}^{\mathit{full}})$$

Here, $|u|$ is the number of elements in the stack. If this number is greater or equal to a certain predefined number *max* which denotes the maximum stack size, rule ($\mathit{number}^{\mathit{full}}$) applies, and the tag is set to *nok*. The subscript *full* is used to distinguish between different kinds of errors. If $|u| < \mathit{max}$, the statement is executed as before, i.e. n is pushed, and the tag is not changed. Note that the two rules are complete, which is a requirement when proving properties (section 4.1).

In section 3.2, the value b used in the premise of the two **test** rules was assumed to be either 0 or 1. The two rules are mutually exclusive, but not complete. We now have a mechanism with which it is possible to add another rule to replace the ad-hoc assumption, and that is to be used when $b > 1$. This new rule is given by:

$$\frac{b \ > \ 1}{\langle \textbf{test } s_t \textbf{ else } s_f \textbf{ end}, b : u, \mathit{ok}\rangle \ \Rightarrow \ \langle \epsilon, u, \mathit{nok}_{\mathit{errb}}\rangle} \qquad (\textbf{test}^{\mathit{errb}})$$

In addition, a rule is needed here to deal with the error of stack underflow. If the number of elements in the stack is less than 1, the statement cannot be executed.

Execution is therefore stopped, and the tag is set to *nok*:

$$\frac{|u| \ < \ 1}{\langle \textbf{test} \ s_t \ \textbf{else} \ s_f \ \textbf{end}, u, ok \rangle \ \Rightarrow \ \langle \epsilon, u, nok_{empty} \rangle} \qquad (\textbf{test}^{empty})$$

The other two rules, ($\textbf{test}^{True}$) and ($\textbf{test}^{False}$), are as before, maintaining the *ok*-status, and are included in the two final tables, tables 2 and 3. Note that $|u| \geq 1$ is not included in the premises of these two rules. The construction $b : u$ explicitly denotes that there is at least one element in the stack.

In the *pickn* statement, n is used as an index in the stack. The simplified rule (*pickn*) is now extended with a premise to only match when the index n is 'correct'. Two rules are added to define the transitions to error states (see tables 2 and 3).

The form of the rules introduced in section 3.4 makes the rules more extensible than the form of the simplified rules introduced in section 3.2. If we had extended the simplified rules with the tags, tables 2 and 3 would be twice as big. We would have had to define the behaviour of *statements* in $\langle statement; statements \rangle$ for every possible execution of *statement*.

Tables 2 and 3 summarise all the rules with the tags and the composition rules added. Note that the tag nok_x in the premise of the composition rule ($;^{nok}$) is used to pass the tag *nok* with the message *empty*, *full* or *errb* (hence the x) to the transition below the horizontal line.

4 Using the Formal Description.

We have discussed the formal description of the subset of the Clasp language. The rules precisely describe, in a clear and compact way, what happens in a normal situation and in case some error occurs (proper behaviour). In this section, this formal description is used. First, it will be shown how it can be used to prove properties of the language. In the second part, a more general view is given on using the formal description in building support tools for the software development process.

4.1 Proving Properties.

Security is an important aspect of a smart card system. Runtime errors like stack overflow, stack underflow and non-determinism of code impinge on the security. Trust can be gained by proving that these errors will not occur, or that if they do, they will be dealt with. The formal rules allow for such proofs. Since the rules are precisely the formal specification of the language Clasp, properties proven using the rules are properties proven for Clasp programs.

All proofs follow the same strategy [10] (this still holds when the subset is extended with loop constructs): prove that the property holds for all derivation sequences of length 0. Then prove the property holds for all other derivation sequences: assume that the property holds for all derivation sequences of length

Table 2. Part 1 of the formal rules of the subset of Clasp with the composition rules and tags added.

$\dfrac{\lvert u\rvert \ \geq \ max}{\langle n, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u, nok_{full}\rangle}$	$(number^{full})$
$\dfrac{\lvert u\rvert \ < \ max}{\langle n, u, ok\rangle \ \Rightarrow \ \langle \epsilon, n : u, ok\rangle}$	$(number^{ok})$
$\langle \mathbf{skip}, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u, ok\rangle$	$(\mathbf{skip})$
$\dfrac{\lvert u\rvert \ < \ 1}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u, nok_{empty}\rangle}$	$(\mathbf{test}^{empty})$
$\dfrac{b \ > \ 1}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}, b : u, ok\rangle \ \Rightarrow \ \langle \epsilon, u, nok_{errb}\rangle}$	$(\mathbf{test}^{errb})$
$\dfrac{b \ = \ 1}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}, b : u, ok\rangle \ \Rightarrow \ \langle s_t, u, ok\rangle}$	$(\mathbf{test}^{True})$
$\dfrac{b \ = \ 0}{\langle \mathbf{test}\ s_t\ \mathbf{else}\ s_f\ \mathbf{end}, b : u, ok\rangle \ \Rightarrow \ \langle s_f, u, ok\rangle}$	$(\mathbf{test}^{False})$
$\dfrac{\langle s_1, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u', ok\rangle}{\langle s_1;\ s_2, u, ok\rangle \ \Rightarrow \ \langle s_2, u', ok\rangle}$	$(;^{ok})$
$\dfrac{\langle s_1, u, ok\rangle \ \Rightarrow \ \langle s_1', u', ok\rangle \ \wedge \ s_1' \ \neq \ \epsilon}{\langle s_1;\ s_2, u, ok\rangle \ \Rightarrow \ \langle s_1';\ s_2, u', ok\rangle}$	$(;^{step})$
$\dfrac{\langle s_1, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u', nok_x\rangle}{\langle s_1;\ s_2, u, ok\rangle \ \Rightarrow \ \langle \epsilon, u', nok_x\rangle}$	$(;^{nok})$

at most k (this is the *induction hypothesis*) and show that it holds for derivation sequences of length $k + 1$.

Now, it is proven that stack overflow will not occur, i.e. execution continues if execution of the current statement will not cause the stack to overflow, and execution is stopped with the tag set to *nok* if it does.
Assumption: execution is started with $\lvert stack\rvert \leq max$. Throughout the proof *stack* is used to denote any stack, and, as before, *max* is used to denote the maximum stack size.
case $k = 0$: The property $\lvert stack\rvert \leq max$ still holds, for if no statements are executed, $\lvert stack\rvert \leq max$ holds by the initial assumption. □
case $k + 1$: Every possible statement to be executed as the $(k + 1)$-th statement

Table 3. Part 2 of the formal rules of the subset of Clasp with the composition rules and tags added.

$\dfrac{\lvert u\rvert < 1}{\langle pickn, u, ok\rangle \Rightarrow \langle \epsilon, u, nok_{empty}\rangle}$	$(pickn^{empty})$
$\dfrac{n < 1 \lor n > \lvert u\rvert}{\langle pickn, n : u, ok\rangle \Rightarrow \langle \epsilon, u, nok_{pickn}\rangle}$	$(pickn^{nok})$
$\dfrac{n \geq 1 \land n \leq \lvert u\rvert}{\langle pickn, n : u, ok\rangle \Rightarrow \langle \epsilon, u[n] : u, ok\rangle}$	$(pickn^{ok})$
$\dfrac{\lvert u\rvert < 1}{\langle rotn, u, ok\rangle \Rightarrow \langle \epsilon, u, nok_{empty}\rangle}$	$(rotn^{empty})$
$\dfrac{n < 2 \lor n > \lvert u\rvert}{\langle rotn, n : u, ok\rangle \Rightarrow \langle \epsilon, u, nok_{rotn}\rangle}$	$(rotn^{nok})$
$\dfrac{n \geq 1 \land n \leq \lvert u\rvert}{\langle rotn, n : u, ok\rangle \Rightarrow \langle \epsilon, u[u[n]/1, u[1]/2, \ldots, u[n-1]/n], ok\rangle}$	$(rotn^{ok})$
$\dfrac{\lvert u\rvert < 2}{\langle <, u, ok\rangle \Rightarrow \langle \epsilon, u, nok_{empty}\rangle}$	$(<^{empty})$
$\dfrac{n_2 < n_1}{\langle <, n_1 : n_2 : u, ok\rangle \Rightarrow \langle \epsilon, 1 : u, ok\rangle}$	$(<^{True})$
$\dfrac{n_2 \geq n_1}{\langle <, n_1 : n_2 : u, ok\rangle \Rightarrow \langle \epsilon, 0 : u, ok\rangle}$	$(<^{False})$

must be looked at:
subcase n: When n is encountered, $|stack| \leq max$ holds by induction. Now, suppose there is stack overflow danger, i.e. $|stack| = max$. Then, rule $(number^{full})$ applies and deals with this in the proper way, i.e. execution is stopped and the tag is set to *nok*. There is no other rule that matches in this situation. The other possibility is that $|stack| < max$. Now, $(number^{ok})$ applies, and n is pushed. Consequently, the stack will grow, but $|stack| \leq max$ still holds. □
subcase skip: Again, by induction this statement will be encountered with $|stack| \leq max$. **skip** does not change the stack, so, $|stack| \leq max$ still holds. □
subcase test: By induction, $|stack| \leq max$ holds when encountering statement **test**. Here, $stack = b : u$, and in all the **test** rules except $(\textbf{test}^{empty})$, one element is taken off the stack, namely b. In effect, $|stack| \leq max$ still holds.

(**test**^{empty}) does not change the stack, so, because $|stack| \leq max$ holds before this rule is applied, it still holds. □
subcase ;: In all ; rules, the single step in the premise is the last step in the induction, i.e. it is the step from $k-1$ to k. So, after execution of the premise $|stack| \leq max$ holds. Now, none of the ; rules thereafter change the stack. Therefore, $|stack| \leq max$ still holds. □
subcase *pickn:* By induction, $|stack| \leq max$ holds when encountering the statement *pickn.* ($pickn^{empty}$) does not change the stack, and, ($pickn^{nok}$) pops an element, so, in both cases, $|stack| \leq max$ still holds after execution. ($pickn^{ok}$) replaces the top element by another element, so, again, if by induction $|stack| \leq max$, it still holds now. □
subcase *rotn*: Similar to *pickn*, for it also replaces (just more) elements in the stack, and does not change the size of the stack. So, $|stack| \leq max$ still holds after executing the statement. □
subcase $<$: ($<^{empty}$) does not change the size of the stack. The other two rules replace the top two elements by one element and therefore do not increase the number of elements in the stack (in fact, they are decreasing it). So, in each case, $|stack| \leq max$ still holds after execution. □
The property is now proven for all possible statements to be executed next. This finishes the proof for **case** $k+1$. Because k can be any natural number, the property is proven for any sequence length. □

The above proof constitutes an informal proof, in the sense that it does not use any formal notation as is used in a formal proof. It is just as thorough though.

It was proven that the error stack overflow will be dealt with. A similar proof can be given for the error stack underflow. Instead of ensuring that $|stack| \leq max$ holds all the time, it must be ensured that $|stack| \geq 0$ holds all the time. Execution is started with $|stack| \geq 0$ (assumption). As an example, consider the subcase for statement *rotn*. Using the induction hypothesis, $|stack| \geq 0$ holds when encountering a statement *rotn*. Suppose that $|stack| = 0$, and popping an element will cause stack underflow. This situation is intercepted by rule ($rotn^{empty}$). Apart from ensuring that there is a rule that matches, it also needs to be ensured that there is no other rule that matches. Indeed this is the case, for it can immediately be seen that the other rules all write their starting stack configuration as $n : u$, i.e. at least one element on the stack. The other possibility to cause stack underflow is when n is 'not right', i.e. $n < 2$, for at least two elements will be needed in order to do a rotation. Here, rule ($rotn^{nok}$) matches, which also traps the possibility of $n > |stack|$. Note that the two premises of ($rotn^{nok}$) do not have to be included in the rule ($rotn^{empty}$). The only other possibility left is that n is 'right'. In this situation, only rule ($rotn^{ok}$) matches. Note also that the premise $|u| \geq 1$ need not be added. If $|u| \geq 1$ then n would have to be either 0 or 1, because $n \leq |u|$ is also part of the premise. The premise $n \geq 2$ would not be true, and using the boolean construct the premise as a whole would not be true.

In a similar way, other properties that are important to smart card software can be proven. Determinism of the language is such a property. A formal proof is not given here. Informally, one should prove that every possible execution for every possible statement has one, and only one, matching rule that defines its behaviour. This is not difficult to prove for the subset of the Clasp language, i.e. the rules in tables 2 and 3. For instance, the rules for the statement n define the only two possible executions, and they are mutually exclusive. Again, induction can be used to prove the property for compound statements.

New rules or new restrictions in the language can also be defined. Consider, for instance, the possibility of byte overflow. The only statement that pushes a number onto the stack, namely n, can not cause overflow, for $0 \leq n \leq 255$ by definition. The statement $<$ also deals with bytes, for it compares two bytes. The rules, however, define its behaviour: either a 0 or a 1 is pushed onto the stack. So, $<$ will not cause any problems. Of course, the inequality operators use arithmetic operators, and these operators should also be 'checked'. Our subset of Clasp has $-$ as its only arithmetic operator, so, this is the only operator that needs to be looked at. We now do the reverse of what we did before: we design a rule for this arithmetic operator, in such a way, that it will behave correctly. The rules can be defined as:

$$\frac{|u| \; < \; 2}{\langle -, u, ok \rangle \; \Rightarrow \; \langle \epsilon, u, nok_{empty} \rangle} \qquad (-^{empty})$$

$$\langle -, n_1 : n_2 : u, ok \rangle \; \Rightarrow \; \langle \epsilon, (n_2 - n_1) \; mod \; 256 : u, ok \rangle \qquad (-^{ok})$$

Rule $(-^{ok})$ simply ignores byte overflow by using the *mod* function. An alternative approach is to generate an exception (Clasp includes exception handling, but it is not shown here). The point here is that any rule can be defined, and a choice must be made when designing a language.

4.2 Tool Support

A formal specification can play an important role in the software engineering process. Figure 4 identifies the major 'components' in the process. We use the term component here as a general term for such diverse items as the concepts on which a system is designed, the specification of the system, and also the documentation, implementation and 'derived components'. The latter include various tools, such as simulators of the system. Figure 4 shows only some of the many relationships between the components. For example the connection between concept and specification represents an iterative process of design and redesign. The connection between specification and derived components represents program generation tools and also programming and design efforts.

A formal description of the subset of the Clasp language has been given and its use demonstrated. This is just one component of the software for smart cards. Any system is as weak as its weakest component so there is a need for formally specifying the complete Clasp language, and also Tosca, the compilers, the interpreters and the assembler. Then we can begin to reason about the system as a whole. We have formally specified a substantial part of the Clasp language, its compiler and interpreter. Currently we are working on the specification of Tosca and on completing the Clasp specifications.

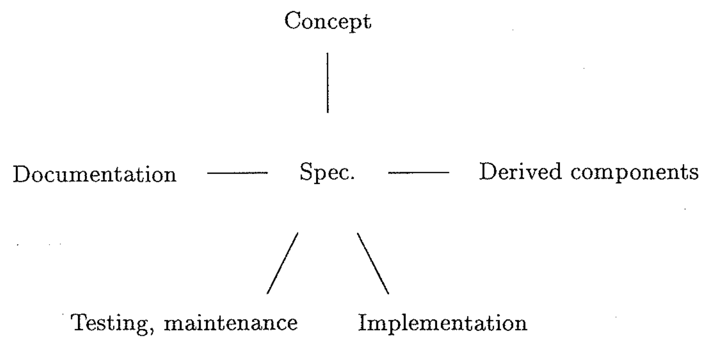


Fig. 4. Relationships between specification, components and derived components in the software engineering process.

Work on the components should be supported by adequate tools, which basically take as input the specification and produce as output a *derived component.* The following are useful derived components, which implement all or part of the functional behaviour of the component. The non-functional behaviour (that is how much resources are being used) of the derived components is generally different from the non-functional behaviour of the component itself.

- An *animator* implements the functional behaviour as specified, and supports interaction. The key feature of an animator is that it allows the user to inspect and even change data whilst the data is being processed.
- A *simulator* implements the functional behaviour as specified, or an abstraction of the behaviour, and supports statistics gathering like timing.
- An *emulator* implements precisely the functional behaviour as specified, and is often used as a prototype.

These definitions are not static, and vary considerably across different fields of computer science.

Tool support is necessary to keep the development of these derived components, the specification, the documentation and the testing and verification procedures, and the real component in step. Figure 4 shows the relationships between all the relevant parts in the development process.

A formal specification in the style that we have presented in the previous section is abstract in the sense that it only deals with the functional behaviour. At the same time it is detailed because it describes precisely what should happen in a normal situation and also in case some error has occurred. Our formal specification can thus serve as a starting point for building the derived components as well as the real implementation. The programming activity involved should be mainly concerned with choosing the right programming language and with creating the non-functional behaviours desired for the derived and real components. The functional behaviour is always the same.

We have built a simple tool to generate the major part of the animator from our Clasp specification. The animator is written in the functional programming language Miranda [4] [15]. We are now in the process of building a generator for simulators. Here is a fragment of such a simulator, written in C:

```
void test( statement * s, stack * u, tag * o ) {
  if( o->ok_nok_enum == Ok ) {
    if( stack_size( u ) < 1 ) {
      s->statement_enum = Empty ;
      tag_set_nok( o, Nok, "stack_empty" ) ;
    } else {
      int b = stack_pop( u ) ;
      if( b > 1 ) {
        s->statement_enum = Empty ;
        tag_set_nok( o, Nok, "errb" ) ;
      } else if( b == 1 ) {
        * s = * s->statement_union.test.test_true ;
        tag_set_ok( o, Ok ) ;
      } else if( b == 0 ) {
        * s = * s->statement_union.test.test_false ;
        tag_set_ok( o, Ok ) ;
      }
    }
  }
}
```

The structures `statement`, `stack` and `tag` correspond to the statements s, the stack u and the tag ok/nok of the specification of tables 2 and 3. The function `test` corresponds to the four rules ($\mathbf{test}^{empty}$), ($\mathbf{test}^{errb}$), ($\mathbf{test}^{True}$) and ($\mathbf{test}^{False}$).

Without going into the detail of the C fragment, we can see that the structure of the function reflects that of the four rules from the specification. The first conditional makes sure that the **test** statement is only executed if the tag is ok. The next conditional, checks whether the stack contains at least one element. If this is the case, the final conditionals distinguish between the case that the boolean value b on the top of the stack represents a non-boolean value, true or false.

The result of executing the **test** statement is recorded by modifying the structures that are passed to the function `test`. This is the only major difference

[4] Miranda is a trademark of Research Software Ltd.

between the specification and the simulator. It represents our efforts to make the simulator run fast, whilst maintaining the appropriate functional behaviour.

5 Related work

An approach similar to ours in the domain of building a high integrity compiler is described by Stepney [12]. Her compiler is for a small general purpose language, which by coincidence is also called Tosca.

Much work has been done to develop formal methods specifically for reasoning about protocols. The specification formalism LOTOS is widely used in this area. The protocols that are used in smart card systems could also be modelled using formalisms such as LOTOS but we do not know whether that has been done. The work at GMD [3, 9] uses a Petri net based method to perform model checking on the protocols used in the STARCOS and STARMOD systems [13].

In our own previous work, we have looked at the modelling of a smart card protocol with a view to proving liveness of that protocol [7]. We have recently built an animation system [5] for the protocols that are defined in the CEN European standard Inter Sector electronic purse [2]. This work explores the use of the visual programming language Prograph [4] as an animation support system. The connection between a formal specification of the protocols and the production of the animation is still to be made.

6 Conclusions

Using a formal approach to building trustworthy software is not widely practiced in the smart card community. This is to some extent surprising: the reason why smart cards are used is because they rely on cryptographic protocols. Cryptology is a branch of mathematics that is devoted to proving properties of cryptographic algorithms and protocols. In practice, when it comes to implementing these protocols in smart cards, the attention shifts to more technical issues, such as getting the implementation to work and making it work with limited resources, leaving correctness assurance aside.

We have shown that formal methods can profitably be used to build trustworthy components for a smart card system. Our main argument is that smart cards are tiny, and so the languages used to program the software components should be highly focussed and small. This makes it feasible to specify such languages formally. Programs written in such languages have a well-defined meaning. They are therefore amenable to formal reasoning. We have shown how properties can be proved, when given a small subset of the language that plays an important rôle in our smart card system. However, the methods used can be applied to any language or system. We have discussed the necessity of tool support in smart card software engineering. By giving a sample component of a simulator we have shown how formal rules can be used to help build suitable tools.

Acknowledgements

We thank Hugh McEvoy, Michael Butler, Andy Gravell, Jon Mountjoy and the referees for their comments on a draft version of the paper.

References

1. E. K. de Jong Frz. Objects in smart cards. In B. Struif, editor, *5th GMD-Smart card workshop*, pages 12.1–12.6, Darmstadt, Germany, Jan 1995. GMD, Darmstadt.
2. CEN European Committee for Standardization. Identification card systems – inter–sector electronic purse. Draft standard prEN 1546, European Committee for Standardization, Brussels, Nov 1995.
3. H. Giehl. Verifikation von Smartcard-Anwendungen mittels producktnetzen. GMD-Studien 225, GMD, D-53754 Sankt Augustin, Gemany, Nov 1993.
4. F. R. Giles, P. T. Cox, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Workshop on Visual Languages*, pages 150–156, Rome, Italy, Oct 1989. IEEE Computer society press, Washington.
5. H. Glaser, P. H. Hartel, and E. K. de Jong Frz. Visualising the structure of an IC-card security architecture. In *Smart Card 1996 convention proceedings – Technology and markets conference*, pages 22–34. Quality marketing services Ltd, Peterborough, UK, Feb 1996.
6. J. Gosling and H. McGinton. *The Java Language Environment: A White Paper.* http://java.sun.com/whitePaper/java-whitepaper-1.html, 1995.
7. P. H. Hartel and E. K. de Jong Frz. Towards testability in smart card operating system design. In V. Cordonnier and J-J. Quisquater, editors, *1st Smart card research and advanced application conference (CARDIS 94)*, pages 73–88, Lille France, Oct 1994. Univ. de Lille, France.
8. ITSEC. *Evaluation criteria for IT security – part 3: Assurance of IT systems.* INFOSEC central office, Brussels, Belgium, version 1.2 edition, 1993.
9. M. Nebel. Ein produktnetz zur verifikation von Smartcard-Anwendungen in der STARCOS-Umgebung. GMD-Studien 234, GMD, D-53754 Sankt Augustin, Gemany, Jul 1994.
10. H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, England, 1991.
11. P. Peyret. Application-enabling card systems with plug-and-play applets. In *Smart Card 1996 convention proceedings – Technology and markets conference*, pages 51–72. Quality marketing services Ltd, Peterborough, UK, Feb 1996.
12. S. Stepney. *High integrity compilation: A case study.* Prentice Hall, Hemel Hempstead, England, 1993.
13. B. Struif. Das smartcard anwendungspakket STARCOS. *Der GMD Spiegel*, 22(1):29–34, Mar 1992.
14. A. S. Tanenbaum. *Structured computer organisation.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1984.
15. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

This article was processed using the LaTeX macro package with LLNCS style

Protection of Software Algorithms Executed on Secure Microprocessors

H.D.L. Hollmann[1], J.P.M.G. Linnartz[1], J.H. v. Lint[2], C.P.M.J. Baggen[1], and L.M.G. Tolhuizen[1]

[1] Philips Research Laboratories, Prof. Holstlaan 4, WY 8, 5656 AA Eindhoven, The Netherlands. Email: (hollmann, linnartz, baggen, tolhuizn)@natlab.research.philips.com

[2] Eindhoven University of Technology, Eindhoven, The Netherlands.

Abstract. Loop structures in software code may reveal essential information about implemented algorithms and their parameters, even if the observer has no knowledge about which instructions are executed. As Secure Processors typically read their (encrypted) instructions from an external memory, the address bus between processor and instruction memory is a known weakness of such a system. This paper addresses the use of dummy instruction fetches inserted whenever the processor is busy executing previous instructions.

We show that for a class of dummy insertion strategies, a Viterbi Decoder can fairly reliably distinguish dummy fetches from real instruction fetches.

In the second part of this paper, we study strategies to choose dummy fetches from a more general model. For certain situations, the optimum protection strategy appears to be deterministic (as opposed to random). Moreover we show that in such case, it is fundamentally not possible to enhance the security of the Secure Processor by keeping the strategy for generating dummy fetches secret to the attacker.

Keywords: Software protection, secure processor, Viterbi decoder, dummy instructions.

1 Introduction

Most investigations into the strength of cryptographic algorithms assume the encryption or decryption algorithm to behave as a black box with three well-defined ports to the external world: one input and one output for the user data and an input for the cryptographic key. A classical situation is the chosen plain-text attack in which an attacker can send input data of his own choice to the circuit and can observe the encrypted output (ciphertext). This scenario is often used to evaluate attacks on smart cards with the objective to find the hidden keys buried in the silicon circuit on the card.

Practical implementations, however, may be less ideal than such a black box model, because significant amounts of information about the key may leak through necessary actions of the processor during execution of the cryptographic algorithm. Execution times, currents drawn through power supply pins, radiated electromagnetic energy and accesses to an external memory for fetching instructions can all be exploited by an attacker.

Recently, it has been realized (e.g. [1]) that observed computation times used by an RSA public-key crypto system [2] might be exploited to find the private key used by the processor, although to our best knowledge no attacks have yet been shown in practice. In RSA, a plain text message m is raised to some power e, where e is a cryptographic key. Typically, the operation $c = m^e \bmod N$ is implemented on a binary machine through successive squaring of an intermediate result $\hat{m}$. The algorithm recursively reads all key bits. For a bit in the key e that is set ("1"), the processor executes at least one multiplication instruction more than for a "0" bit. If the attacker is able to determine when such instructions are executed, the secret key e can be found.

This example appears to be a special case of a larger class of possible attacks to reconstruct embedded algorithms in electronic circuits. Therefore, in this paper we widen our scope, including not only smart cards but also security modules that consist of several chips, connected with busses that can be probed.

Since conventional processors provide almost no possibilities to keep the algorithm confidential, "Secure Processors" [3], [4], [5] have been developed that allow execution of algorithms without revealing to external observers the instruction code or parameters used. The Central Processor Unit (CPU) of a Secure Processor accesses an external memory to fetch encrypted instructions. Decryption occurs on the CPU chip. This scenario has the advantage that the manufacturer or its service organization can easily replace the (cheap) ROM memory to update the features of the system, without having to replace other, more expensive modules.

Secure processors are for instance used in the ignition, steering and braking control of automobiles. The automotive industry desires to keep their proprietary algorithms confidential to their competition, and does not want dangerous situations to evolve if hobbyists "fine tune" the performance of their car by experimenting with the algorithms.

Even if the attacker of the Secure Processor is not able to break the encryption/decryption algorithm, he may get substantial information about the algorithm performed and its parameters from the loop structure of the algorithm. These can be obtained through periodic patterns in the addresses of instructions fetched from memory. Particularly if the above exponentiation algorithm is executed, the trace of addresses reveals which bits in the key have been set to "1". Attempts to obscure the loop structure by a secret permutation on the location of instructions on the memory chip slows down the memory access, and by itself it can not obscure repetitive patterns in addresses.

Information leakage can be avoided if one allows that the processor always reads all instructions in the external memory sequentially, but only uses the instructions needed. Such a solution severely limits the performance of the implementation, so mostly this is not realistic or desirable. This paper reports new results on minimizing the amount of information leaked by Secure Processors.

The organization is as follows: Section 2 models the process of fetching instructions from memory as a Markov Process. This suggests to use a Viterbi Decoder to trace the most likely path, which is worked out in Section 3. A more generic case is presented in Section 4 and evaluated in Sections 5 and 6. Section 5 derives a lower bound for the probability by which the attacker can guess the program sequence correctly. Section 6 shows that a particular strategy can be chosen to ensure optimum security. In this case, the probability that the attacker can guess the correct program sequence is minimum. As an example, Section 7 addresses the case of an n round sequential execution of an algorithm having $n-k$ rounds of relatively short duration and k rounds of long duration.

2 Dummy reads

As illustrated in Figure 1, a Secure Processor consists of a single chip with a microprocessor and a decryption circuit for incoming instructions. The decryption key depends on the address of the instruction. Typically a (secret) random number is generated, using the address as a seed.

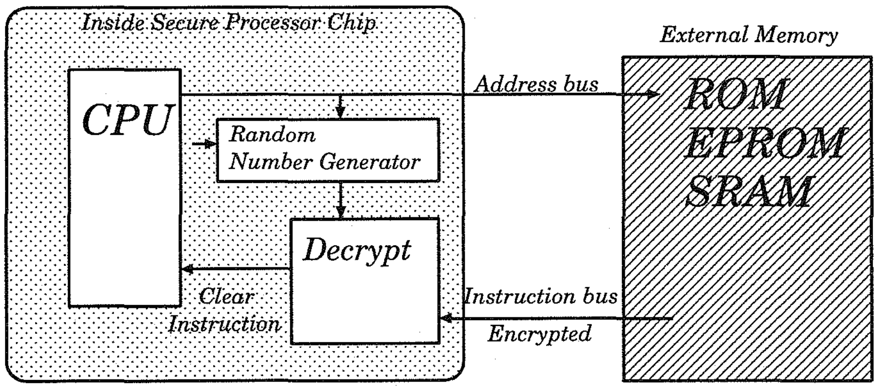


Fig. 1. Basic Elements of Secure MicroProcessor. The CPU and decryption are implemented on the same chip, using a secure silicon technology.

It has been proposed to insert dummy fetches (only) when the processor is busy executing previously fetched instructions or instructions that are already in the processor core (or its cache memory).

In order to investigate such dummy reads, it appears reasonable to model the CPU as a finite state machine, which asks for the next instruction depending on its current state. In Sections 2 and 3 we will make the simplification that the sequence $X^v = X_0, X_1, .., X_v$ of addresses of instructions is a time-homogeneous Markov Process. That is, the probability that $X_{v+1} = j$ depends only on X_v,

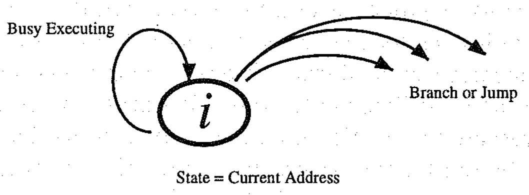


Fig. 2. Markovian model for instruction addresses in software execution

but is further independent of the (discrete) time v. In Section 4 we broaden this model. The unit of time can for instance be the length of a clock cycle on the bus connecting the CPU chip with its memory.

Now let $p_{i,j}$ denote the transition probability of $X_{v+1} = j$ given $X_v = i$. Whenever $i = j$, the processor is busy executing instruction X_v during cycles v and $v + 1$, and X_{v+1} may on the external address bus be replaced by an arbitrary dummy read Y_{v+1}. Otherwise, if $i \neq j$ then $Y_{v+1} = X_{v+1}$. We use $Y^v = Y_0, Y_1, .., Y_v$ to denote the sequence of (dummy and real) addresses that can be observed on the address bus.

Figure 2 describes a postulated model for instruction addresses X^v. The parameters have been inspired by rules of thumb used in processor design, such as "about one in five instructions is a jump", i.e., for all i, $p_{i,i+1} \approx 4 \sum_j p_{i,j}$, with $j \neq i, i+1$. and "75 percent of time the processor is busy executing previously loaded instructions" ($p_{i,i} \approx 0.75$).

3 Estimation of Hidden Markov Process

An attacker must estimate the original sequence X^v from the observed Y^v. Under the condition that the $Y^v|X^v$ is a sequence of independent random variables, that is, the choice of the dummy read Y_v depends only on X_v but not on X^{v-1}, the problem reduces to that of estimating a hidden Markov process. In particular, the most likely path can be found using a Viterbi Decoder (see e.g. [6] for a review), as

$$\hat{X}^v = \arg\max_{X^v} Prob(X^v|Y^v)$$

that is, it finds the argument X^v that has maximum a posteriori probability, given the observed Y^v. Using Bayes' rule, this expression is rewritten as

$$\hat{X}^v = \arg\max_{X^v} \frac{Prob(Y^v|X^v)Prob(X^v)}{Prob(Y^v)},$$

where we may omit $Prob(Y^v)$ because it does not influence the choice of X^v that maximizes the expression. Moreover, using the Markovian property of the instruction sequence X^v and the independence of $Y_v|X_v, X_{v-1}$ on X^{v-2}, and after taking a logarithm, we arrive at

$$\hat{X}^v = \arg\max_{X^v} \sum_{w=0}^{v} [\log Prob(Y_w|X_w, X_{w-1}) + \log Prob(X_w|X_{w-1})],$$

where we define $Prob(X_0|X_{-1})$ as the initial probability $Prob(X_0)$. The above expression has been the basis to the architecture of the Viterbi Decoder, which gives an efficient way to find the sequence $\hat{X}^v$ that has maximum probability. In Figure 3, we use transition probabilities from our postulated model on the a priori behaviour $p_{i,j}$ of software algorithms, and we use $Prob(Y_v|X_v, X_{v-1})$ from the assumption that dummy reads are randomly chosen whenever $X_v = X_{v-1}$, with equal (uniform) probability for all memory addresses.

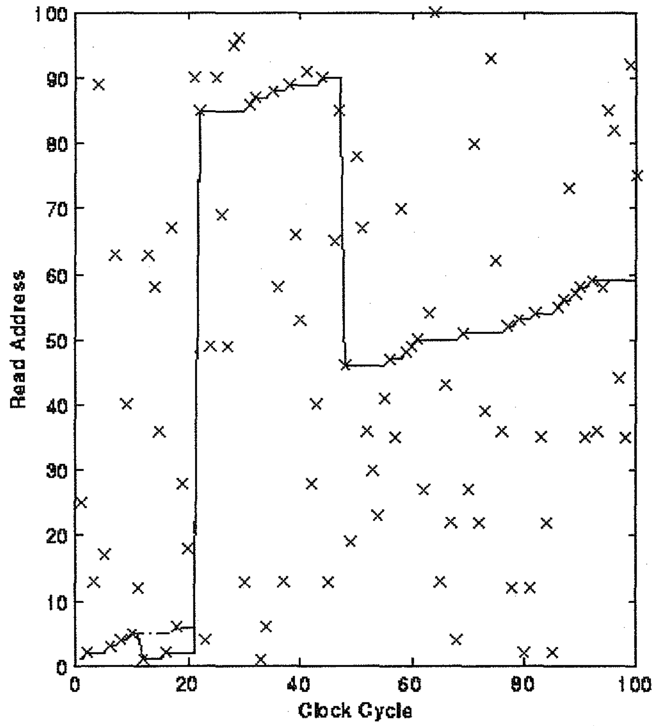


Fig. 3. Instruction address as a function of time. Sample path of the address (Solid line) seen by CPU, instruction fetches on bus, including both real and dummy reads (crosses), and the most likely path as estimated by a Viterbi Decoder (.-.-).

In Figure 3, the program proceeds with the next instruction (or with the next cache line) with probability $p_{i,i+1} = 0.2$. With probability 0.05, a jump to an instruction with a uniformly random location occurs. The probability that the

CPU does not fetch a new instruction is 0.75, which is equal to the fraction of instruction fetches on the external bus that are dummy reads. We take $X_0 = 0$ with probability 1.
In this example we see that the decoder only makes errors between clock cycles 10 and 20. In the rest of the trace, the estimated path (.-) coincides with the path that the CPU actually followed through the instructions.

Speaking in terms of hidden Markov processes, the software developer wishes to minimize the mutual information $I(X^v; Y^v)$. Explicit computation of the conditional entropy $H(X^v|Y^v)$ or $I(X^v; Y^v) = H(Y^v) - H(Y^v|X^v)$ is known to be tedious for hidden Markov processes. Moreover, the designer may select more intricate strategies for choosing Y^v. In such case the Viterbi Algorithm may not be applicable. In the next section, we broaden the class of dummy generation strategies. It turns out that we can quantify the uncertainty in X^v given the observation Y^v for some special cases.

4 Generic Model

In what follows, we will refer to X^v as a *program sequence* of length v. To encrypt such a sequence, entries at locations n for which $X_n = X_{n-1}$ may be replaced by another, according to some method, the *encryption strategy*, that is specified beforehand. We will call a location n with $X_n = X_{n-1}$ a "star", as we may replace the entry X_n by a "wild-card" $*$. (By abuse of notation, we will denote both the original program sequence and the sequence obtained after these replacements by X^v.) Regardless of whether this is desirable from a security point of view, at this point we will allow this encryption method to be *non-deterministic*, that is, the same sequence may be encrypted differently, at different moments or by different realizations of the processor. (Later, we will see examples where the best encryption strategy turns out to be deterministic after all.) We can thus describe an encryption strategy by specifying, for each program sequence $X^v = s$ and each possible encryption $Y^v = t$ of s, the probability that s will be encrypted as t. The task of the interceptor is to obtain information about the original program sequence from the encrypted version.

We can formalize the above idea as follows. Let S be the collection of all possible program ("source") sequences, and let T denote the collection of all possible images (encrypted sequences). Write $|S|$ to denote the cardinality of S.

An *encryption strategy* is described by an $|S| \times |T|$-matrix Q, where $Q_{s,t} = p(t|s)$ is the probability that a given program sequence s will be encrypted as the sequence t. Obviously, the attacker cannot do better than following a Maximum A Posteriori (MAP) probability decoding method: given an encryption t, the original program sequence is estimated as

$$\hat{s} = \arg\max_{s \in S} p(s|t).$$

(Note that, in general, the attacker would have to know the encryption strategy Q in order to do so.) In this case, the optimal (MAP) probability p of correct decoding, averaged over all t, is

$$p = \sum_{t \in T} p(t) \max_{s \in S} p(s|t) = \sum_{t \in T} \max_{s \in S} p(t|s)p(s),$$

where $p(s)$ denotes the probability that the program sequence to be encrypted equals s. Our aim is to choose Q such that p is as small as possible, since the attacker on average has the uncertainty of at least $1/p$ program sequences that could have resulted in the observed $Y^v = t$. In the special case of equiprobable program sequences, i.e., if $p(s)$ is independent of s, we have $p(s) = 1/|S|$ and the detection principle reduces to Maximum Likelihood detection, with

$$p = |S|^{-1} \sum_{t \in T} \max_{s \in S} p(t|s).$$

Note that this p is an *average* probability, that is, some sequences may be easier to decrypt than others.

5 A method to obtain a lower bound

We will now show that p cannot be made arbitrarily small by choosing a suitable Q. To this end, let $A \subset S$ be a collection of source sequences such that any two distinct words in A will always be encrypted into distinct codewords. (Note that this is precisely the case when we can find for each two distinct sequences in A a (non-star) coordinate where these sequences have different non-star symbols.)

Now a possible approach by the attacker is to decrypt when possible an observed sequence into the corresponding program sequence a from A. (Note that by our assumption on A there can be at most one such a sequence.) Using this approach, the probability of correct decoding at least equals the probability that the program sequence is contained in A. So we have proved that

$$p \geq \sum_{a \in A} p(a).$$

In the case of equiprobable program sequences, this bound reduces to

$$p \geq |A|/|S|.$$

6 A case of equality

We will now discuss a situation where the lower bound in the previous section holds with equality. For the remainder of this section, we will assume that the program sequences are equiprobable. Let A be a collection as in the previous section. To each $a \in A$ we will assign a *fixed* encryption $t_a \in T$. Now suppose that this assignment can be done in such a way that the collection $B = \{t_a \mid a \in A\}$ has the property that *each* program sequence can be encrypted into some member of B. Then, as a consequence of these assumptions we can assign to each program sequence $s \in S$ an encryption $f(s)$ in B, with $f(a) = t_a$, $a \in A$. Note that, due to our assumptions on A, the sequences t_a, $a \in A$, are all distinct, that is, $|B| = |A|$. We will consider the map f as describing a *deterministic* encryption rule.

The corresponding strategy matrix Q is given by

$$p(t|s) = \begin{cases} 1, \text{ if } t = f(s); \\ 0, \text{ otherwise.} \end{cases}$$

We now claim that under these assumptions, we have that

$$p = |A|/|S|,$$

with optimal encryption strategy described by Q (or by f). Indeed, we have that

$$\begin{aligned} p &= \sum_{t \in T} \max_{s \in S} p(t|s)/|S| \\ &= \sum_{b \in B} 1/|S| \\ &= |B|/|S| \\ &= |A|/|S|. \end{aligned}$$

We conclude that the optimal encoding strategy can be deterministic. The result also shows that choosing a statistical strategy does not always make the task of the attacker more difficult.

7 Example

As a special case, consider the sequential execution of n rounds of an algorithm out of which some, say k steps require a longer execution time. A program is represented by a sequence X^v of length $v = n + k$ obtained from the sequence

$$123 \cdots n$$

by the insertion of precisely k stars. Moreover, the first and last symbols in the sequence are non-stars, and the sequence does not contain two consecutive stars.

Let S be the collection of all source sequences with n symbols and k stars, and let T denote the collection of all possible images of source sequences in S. Write $|S|$ or $|S|(n,k)$ to denote the cardinality of $S(n,k)$. Note that $|S|(n,k) = \binom{n-1}{k}$.

For a word s in S, let $*(s)$ denote the collection of positions where the source sequence s has a *. For example, if

$$s = 12 * 3 * 45,$$

then

$$*(s) = \{3, 5\}.$$

To simplify the following arguments, we imagine that the collection of positions $I = \{1, 2, ..., n+k\}$ is partitioned into pairs

$$I_1 = \{1,2\}, \qquad I_2 = \{3,4\}, ...,$$

and possibly (if $n+k$ is odd) the singleton set $I_0 = n+k$. (Note that, if $n+k$ is odd, then all words s in S will have the symbol n in position $n+k$, hence all stars in words of S occur within the sets I_j, $j > 0$.) We take the set A as the collection of all a in S for which $*(A)$ consists of even numbers only. Note that we have

$$|A| = \binom{[(n+k-1)/2]}{k}.$$

It can be reasoned that any two members of this A will always produce distinct codewords. The encoding map f is described in terms of the coordinate pairs I_j as follows. For all sequences s in S, replace a star in position $i \in I_j$, say, by the symbol on the other position of I_j. The resulting t sequence could also have been generated from a sequence with stars only in even positions. It is now evident that

$$f(s) = f(a(s))$$

holds for every word s in S, where $a(s)$ denotes the special sequence in A that has the same image as s. So we have proved that the probability of correct decoding

$$p = \binom{[(n+k-1)/2]}{k} \Big/ \binom{n-1}{k} = \frac{[(n+k-1)/2]!(n-k-1)!}{[(n-k-1)/2]!(n-1)!}$$

holds for each n and k, with the optimal deterministic encoding strategy given by the map f above.

7.1 Asymptotic behavior

To study the asymptotic behavior for large programs ($n \to \infty$) , we define the ratio $\alpha = k/n$. The Appendix shows that

$$p \longrightarrow 2^{n(1-\alpha-h(\frac{1}{2}+\frac{\alpha}{2}))}$$

The optimum ratio α which asymptotically minimizes the value of the probability of correct decoding p with a given n and variable $n+k$ appears to be $\alpha = k/n = 3/5$. The optimum density of stars in very long programs equals $\frac{3}{8}$. This leads to

$$\lim_{n\to\infty} \frac{1}{n} \log p = \frac{4}{5} h\left(\frac{3}{4}\right) - h\left(\frac{3}{5}\right) \approx -0.322,$$

where $h(q)$ is the binary entropy function. Hence, the probability of correct decoding tends to $p \to 2^{-0.322n}$.

8 Concluding Discussion

The address bus between a Secure Processor and its memory is a weak point in the security of the system. Substantial information can be obtained from observing the loop structure of the program executed by the Secure Processor.

The insertion of dummy instructions helps to obscure the exact structure of the program. However, methods exist to separate dummy reads from real instruction reads. We tested the use of a Viterbi Decoder on simulated address traces and concluded that dummy reads may not simply be choosen uniformly randomly.

A frame work for the analysis of dummy reads has been proposed. We used this framework to find results for a special case of software code. For our considered situation, optimum strategies are deterministic. In such case, given the attacker information about the strategy for generating dummy reads does not compromise the security of the Secure Processor.

Although our results only are a preliminary step towards a full understanding of these issues, the results are believed to be relevant to evaluate the strength of Secure Processors, to develop stronger or more efficient strategies for dummy fetches and to better understand the potential of recently introduced timing attacks (in which cryptanalysts exploit measurements of instruction execution times).

9 Acknowledgement

The authors wish to thank P. Lenoir for fruitful discussions.

References

1. P.C. Kocher, "Cryptanalysis of Diffie-Hellman, RSA, DSS and other Systems using timing Attacks", extended abstract, World Wide Web, December 7, 1995, submitted to Crypto 96.
2. R.L. Rivest, A. Shamir, and L.M. Adleman, "A method for obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21, No. 2, Feb. 1978, pp. 120–126.
3. R.M. Best, "Preventing Software Piracy with Crypto-microprocessors", Proc. Compcon 1980 (spring), pp. 466–469, IEEE-CS Press, Los Alamitos, CA.
4. R.M. Best, U.S. Patent 4,168,396, "Microprocessor for executing enciphered programs", September 18, 1979.
5. U.S. Patent 5,386,469, "Firmware encryption for microprocessor/microcomputer", January 31, 1995.
6. G.D. Forney, Jr., "The Viterbi Algorithm", Proc. IEEE, Vol. 61, March 1973, pp. 268–278.

10 Appendix: Asymptotic Star Density

To study the asymptotic behavior for large programs ($n \to \infty$), we use the binary entropy function, defined as

$$h(q) = -[q \log q + (1-q)\log(1-q)]. \tag{1}$$

Its derivative is

$$h'(q) = \log \frac{1-q}{q}, \tag{2}$$

for any base of the logarithm. In this paper, we assume all logarithms and entropy functions to the base two.
Using Stirling's approximation for factorials

$$\sqrt{2\pi n} n^n e^{-n} e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n} n^n e^{-n} e^{\frac{1}{12n}}, \tag{3}$$

we obtain for binomial coefficients

$$\binom{n}{k} = 2^{n[h(\frac{k}{n}) + O(\frac{\log n}{n})]}. \tag{4}$$

Inserting this in the expression for p, we obtain

$$\lim_{n\to\infty} \frac{1}{n} \log p = \frac{1+\alpha}{2} h\left(\frac{2\alpha}{1+\alpha}\right) - h(\alpha) \tag{5}$$

$$= 1 - \alpha - h(\frac{1}{2} + \frac{\alpha}{2}). \tag{6}$$

We are interested in the value of k which asymptotically minimizes the value of the probability of correct decoding p. In order to find the optimum (for a given n and variable $n+k$), we set the derivative with respect to α equal to zero:

$$-1 = \frac{1}{2} h'\left(\frac{1}{2} + \frac{\alpha}{2}\right) = \frac{1}{2} \log\left(\frac{1-\alpha}{1+\alpha}\right).$$

We find that the optimum α equals $\alpha = \frac{3}{5}$. Note that the optimum density of stars in very long programs equals $\frac{3}{8}$, since the total length of a string including the stars equals $(1+\alpha)n$. Substituting this result in (6), we obtain

$$\lim_{n\to\infty} \frac{1}{n} \log p = \frac{4}{5} h\left(\frac{3}{4}\right) - h\left(\frac{3}{5}\right). \tag{7}$$

This article was processed using the LaTeX macro package with LLNCS style

Multi-Application Smart Cards and Encrypted Data Processing

Josep Domingo i Ferrer *

Universitat Rovira i Virgili, Escola Tècnica Superior d'Enginyeria, Estadística i Investigació Operativa, Autovia de Salou s/n., E-43006 Tarragona, Catalonia, e-mail jdomingo@etse.urv.es

Abstract. Some existing approaches to multi-application smart card design rely on the card containing data and importing the code of functions (methods) to be performed on data. A complementary solution is proposed in this paper to relax the requirement —or rather bottleneck— that all confidential data and processing be supported by the card. Our approach is based on running some applications outside the card using encrypted data processing, specifically privacy homomorphisms. Examples of privacy homomorphisms are given, one of which is very recent and allows full arithmetic on encrypted data while remaining secure against known-cleartext attacks.
Keywords: Cryptographic protocols for IC cards, IC architecture and techniques.

1 Introduction

Future smart cards will no longer be dedicated devices implementing a single issuer-dependent application. Instead, the trend is toward designing multi-application cards with an own operating system that can perform a variety of functions [Guil92]; in this way, an individual bearing a single card will be able to interact with several service providers (operating on the data stored in the card). In order for the card to be able to cope with several applications, it has been suggested that the card should contain the holder's data and that the code of functions (methods) to be performed on data should be imported by the card operating system from an outside server [Gama94]. In this paper, we propose a complementary solution to relax the requirement that all confidential data and processing be supported by the card. The idea is that *some* applications could run outside the card *on homomorphically encrypted data.* For these applications, the card does not exactly behave as a passive device, because the card operating system remains responsible for data encryption and decryption, and also for controlling access of outside applications to data (see section 3). Running some applications entirely outside the card has several advantages, for example

* This work is partly supported by the Spanish CICYT under grant no. TIC95-0903-C02-02.

- It amounts to having a multiprocessor environment formed by the card's processor and one or more external processors. Thus, true parallelism is possible, although there is some asymmetry when an external processor tries to access data in the card: external processors must interact with the card's processor following the protocol shown in section 3.
- Especially resource-demanding applications can make use of external storage and external processors more powerful than the one on the card. Any application dealing with data stored in the card can benefit from the approach presented here. This includes service provider applications as well as card-initiated applications such as biometric verification (recognition of voice, fingerprints or handwriting) which usually requires a substantial amount of storage and a large number of relatively simple operations.

In section 2, previous work is reviewed. In section 3, the basic idea is outlined. Section 4 contains some background on privacy homomorphisms, which are a tool for computing with encrypted data; examples of privacy homomorphisms are given, one of which has been recently proposed by this author and allows full arithmetic while remaining secure against known-cleartext attacks. Section 5 discusses integration of privacy homomorphisms into an object-oriented architecture. Section 6 is a conclusion.

2 Previous work

In [Gama94], the object-oriented concepts set forth in [Bire94] are used to sketch the operation for a multi-application smart card that can accomodate external service providers operating on the card's data. Given such a card, each service provider allocates a *card object* into the card. The methods in this card object can be invoked by the provider and also by host programs; however, the card object only contains interfaces for its methods, but not the actual code body of these. The program invoking a method is called *client* and the program containing the *card object* is the *card operating system*. In order to use a card object, the following protocol, illustrated on figure 1, is followed

Protocol 1 (On-card computation)

1. *The client program points to a local object (also called* proxy *or* surrogate *object).*
2. *The methods in the client proxy object perform procedure calls to methods in the card object. Upon making a procedure call to a card method, the proxy object provides the card with the certified code corresponding to the invoked method.*
3. *After integrity checks against the method certificate, the card operating system runs the code of the method on the card object data.*
4. *The card operating system returns the response to the client program.*

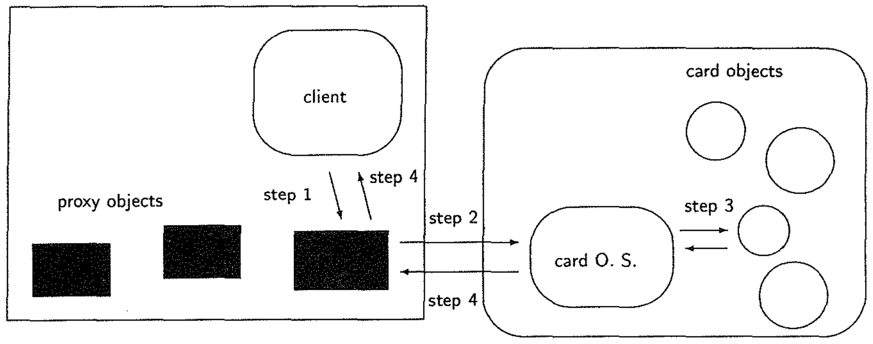


Fig. 1. Protocol 1: on-card computation

In order to speed up this basic protocol, the card operating system may temporarily store inside the card the code of an imported method to make it available to other object calls during the current session. The authors describe an alternative approach, which is based on agents. The only variation is that the client and the card operating system do not interact directly, but through a network agent.

3 Off-card computation

The problem with the approaches described in section 2 (object-oriented or agent-based) is that they rely on the card ultimately supporting all confidential data and processing. Thus, the card's limited storage and processing capacity can be a bottleneck when a very resource-demanding client application is to be serviced or simply, when several client applications are to be run in parallel.

For such applications, we propose the following protocol, illustrated on figure 2, which assumes that card object data can be processed outside the card in an encrypted form

Protocol 2 (Off-card computation)

1. *The client program points to a local object.*
2. *The methods in the client local object request from the card an encrypted version of the card object data. The card delivers the required data after proper security control.*

3. The methods in the client local object compute on encrypted data, find the desired (encrypted) result and send it to the card operating system.
4. The card operating system runs a method in the card object that decrypts the result received from the client; the clear response is then returned to the client program.

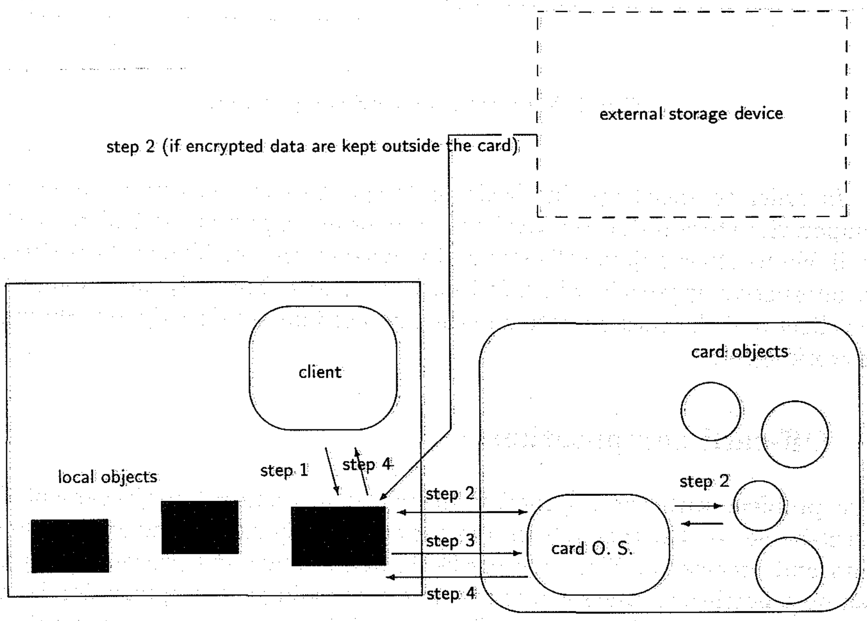


Fig. 2. Protocol 2: off-card computation

Next, the main differences between protocol 1 and protocol 2 are pointed out

- In the latter protocol, step 3 is done by the client program. Computationally, this eliminates a bottleneck and is an advantage, although a trade-off with security is obvious: it is usually *more secure* to confine data inside the card than to have them sent outside the card in an encrypted form.
- Card objects used by applications following protocol 2 contain methods (with their code) for data encryption and for data decryption. Other methods may

be included to control access of client applications to data. In addition, such card objects may also contain method interfaces as required by protocol 1.
- Local objects get data from card objects instead of supplying the latter with the code of methods.
- Due to the differences listed so far, local objects in protocol 2 are not subtypes of card objects and do not behave as proxy or surrogate objects in the sense of protocol 1 and [Bire95].

The implementation of protocol 2 can be refined in several ways

a) If card speed is critical but card storage is not, data need not be interactively decrypted by the card operating system. Instead, encryption can be done upon data acquisition, so that card object data are stored as ciphertext. However, ciphertext often requires more storage than cleartext (see section 4). Remark that encrypted data fields in card objects can also be used by protocol 1 provided that operations on encrypted data are *coded* as additional methods in the proxy objects; then, if card object methods are polymorphically defined, different codes for, say, multiplying data will be automatically provided at step 2 of protocol 1 depending on data being cleartext or being encrypted under one of several possible algorithms (refer to discussion in section 5).
b) If card storage is critical but card speed is not, then card object data can be stored as cleartext and be encrypted when requested by protocol 2.
c) If both card speed and storage are critical, then perhaps we might think of keeping data in encrypted form outside the card. This approach uses the fact that many client applications merely need read-only access (*e. g.* applications querying medical records, etc.). External encrypted data should be properly mirrored to maintain consistency as cleartext data stored in the card change. For speed to be really high, the external storage device should be reachable over the network and allow a high I/O throughput (comparable to the throughput of the card I/O interface).

4 The tool: privacy homomorphisms

Computation on encrypted data does not make sense unless the encryption transformation being used has some homomorphic properties. Privacy homomorphisms (PHs from now on) were formally introduced in [Rive78b] as a tool for processing encrypted data. Basically, they are encryption functions $E_k : T \longrightarrow T'$ which allow to perform a set F' of operations on encrypted data without knowledge of the decryption function D_k. Knowledge of D_k allows to recover the outcome of the corresponding set F of operations on clear data. The security gain is especially apparent in a multilevel security environment such as the ones described in sections 2 and 3. Data can be encrypted by the smart card, be processed by the client program, and the result be decrypted by the smart card.

Next follow some well-known results about PHs. If a PH preserves order, then it is insecure against a ciphertext-only attack. If a PH has addition among its ciphertext-domain operations, then it is insecure under chosen-ciphertext attack ([Ahit87]). With the exception of the RSA algorithm —which preserves only multiplication—, all of the examples proposed in [Rive78b] were subsequently shown to be breakable by a ciphertext-only attack or, at most, a known-cleartext attack (see [Bric88]); the authors of [Bric88] introduced R-additive PHs which remain secure at the cost of putting a restriction on the number of ciphertexts that can be added together. Lacking secure PHs that preserve more than one operation, most successful attempts at encrypted data processing have traditionally relied on *ad-hoc* procedures ([Ahit87], [Trou91]). In [Domi96a], we presented a new PH that preserves addition and multiplication and has the remarkable property of seeming able to withstand known-cleartext attacks.

For illustration purposes, we give two examples of privacy homomorphisms, each having interesting properties in its own right

Example 1. An exponential cipher such as RSA [Rive78a] is a PH. Let $m = pq$, where p and q are two large secret primes (about 100 decimal digits each). In this case,

$$T = T' = \mathbf{Z}_m$$

$$E_k(a) = a^e \bmod m$$

$$D_k(a') = (a')^d \bmod m$$

where $\mathbf{Z}_m$ is the set of integers modulo m, d is secret and $ed \bmod \phi(m) = 1$, with $\phi(m) = (p-1)(q-1)$ being Euler's totient function. Clearly,

$$D_k(E_k(a)) = a^{ed} \bmod m = a^{1+t\phi(m)} \bmod m = a$$

where Euler's theorem is used in the last step. Now, let $F = F' = \{\star\}$, where $\star$ denotes the modular multiplication over $\mathbf{Z}_m$. The following homomorphic property holds

$$E_k(a) \star E_k(b) = (a^e \bmod m)(b^e \bmod m) \bmod m = (a \star b)^e \bmod m = E_k(a \star b)$$

This homomorphism allows only one operation, but appears to be very secure. Finding D_k from E_k, *i. e.* finding d from e, seems to be equivalent to factoring a large modulus m —no polynomial-time algorithm for factoring has been published up-to-date—. An additional interesting property relates to the preservation of the equality predicate, because it holds that

$$E_k(a) = E_k(b) \text{ if and only if } a = b$$

To summarize, the RSA PH has the following features

- The only operation that can be carried out on encrypted data by the client program is multiplication.
- The equality predicate is preserved, and thus comparisons for equality can be done by the client program based on encrypted data.

– Security even against chosen-cleartext attacks seems to be guaranteed.
– Cleartext and ciphertext lengths are about the same, so there is no storage penalty for keeping data in encrypted form on the smart card. □

Example 2. The PH in this example is similar to the one in [Domi96a], but can be proven secure against a known-cleartext attack ([Domi96b]). The public parameters are a positive integer d and a highly composite large integer m ($\approx 10^{200}$) having at least one large prime factor ($\approx 10^{100}$). The secret parameters are $r \in \mathbf{Z}_m$ such that r^{-1} mod m exists and a small divisor m^* of m. In this case the set of cleartext is $T = \mathbf{Z}_{m^*}$. The set of ciphertext is $T' = (\mathbf{Z}_m)^d$. The set F of cleartext operations is formed by addition, subtraction and multiplication in T. The set F' of ciphertext operations contains the corresponding componentwise operations in T'. The PH transformations can be described as

Encryption Randomly split $a \in \mathbf{Z}_{m^*}$ into secret $a_{.1}, \cdots, a_{.d}$ such that $a = \sum_{j=1}^{d} a_{.j} \bmod m^*$ and $a_{.j} \in \mathbf{Z}_m$. Compute

$$E_k(a) = (a_{.1}r \bmod m, a_{.2}r^2 \bmod m, \cdots, a_{.d}r^d \bmod m)$$

Decryption Compute the scalar product of the j-th coordinate by r^{-j} mod m to retrieve $a_{.j}$ mod m. Compute $\sum_{j=1}^{d} a_{.j} \bmod m^*$ to get a.

As encrypted values are computed over $(\mathbf{Z}_m)^d$ by the client program, the use of r requires that the terms of the encrypted value having different r-degree be handled separately —the r-degree of a term is the exponent of the power of r contained in the term—. This is necessary for the smart card to be able to multiply each term by r^{-1} the right number of times, before adding all terms up over $\mathbf{Z}_{m^*}$.

The set F' of ciphertext operations consists of

Addition and subtraction They are done componentwise, *i. e.* between terms with the same degree.

Multiplication It works like in the case of polynomials: all terms are cross-multiplied in $\mathbf{Z}_m$, with an d_1-th degree term by a d_2-th degree term yielding a $d_1 + d_2$-th degree term; finally, terms having the same degree are added up.

Division Cannot be carried out in general because the polynomials are a ring, but not a field. A good solution is to leave divisions in rational format by considering the field of rational functions: the encrypted version of a/b is $\frac{E_k(a)}{E_k(b)}$.

Two remarks about fraction handling

1. When addition or subtraction are performed on fractions with different denominators, numerators cannot be added or subtracted directly. The rules for ordinary fractions should be followed

$$\frac{E_k(a)}{E_k(b)} \pm \frac{E_k(c)}{E_k(d)} = \frac{E_k(a)E_k(d) \pm E_k(b)E_k(c)}{E_k(b)E_k(d)}$$

2. If noninteger initial data are dealt with as fractions, then every result received from the client program level is a fraction; the numerator of the exact result must be decrypted and thereafter divided over the real numbers by the decrypted denominator (be it a power of 10 or not), in order to get the right number of decimal positions.

To summarize, this PH has the following features

- Addition, subtraction, multiplication and division can be carried out on encrypted data by the client program.
- In [Domi96b], it is proven that the PH is secure against known-cleartext attacks provided that $d > 1$ and the number n of *random* known cleartext-ciphertext pairs is such that $n \leq \log_{m^*}(\phi(m)/2)$. This means that the set of ciphertext must be much larger than the set of cleartext. Pairs that are derived from random pairs using the homomorphic properties do not compromise the security of the PH.
- The equality predicate is not preserved, and thus comparisons for equality cannot be done by the client program based on encrypted data. Remark that a given cleartext can have many ciphertext versions for two reasons: A) random splitting during encryption; B) the client program computes over $(\mathbf{Z}_m)^d$ and only the smart card can perform a reduction to $\mathbf{Z}_{m^*}$ during decryption.
- Encryption and decryption transformations can be implemented efficiently, because they only require modular multiplications. Note that no exponentiation is needed, because the powers of r can be precomputed.
- A ciphertext is about $d\frac{\log m}{\log m^*}$ times longer than the corresponding cleartext. Even if this is a storage penalty, a choice of $d = 2$ for encryption should be affordable while remaining secure. □

5 Integrating privacy homomorphisms into an object-oriented architecture

Protocol 2 is complementary to protocol 1. Therefore, the new proposal should have a rather slight impact on the card life cycle as understood in protocol 1. If a card object o is to be used both in protocols 1 and 2, then it has the following structure

$$o = (\{d_i\}, \{I_i\}, \{[E^i, D^i]\}, \{A_i\})$$

where $\{d_i\}$ are data fields (clear or encrypted, depending on the implementation, see section 3 and below), $\{I_i\}$ is a suite of method interfaces as used in protocol 1, $\{[E^i, D^i]\}$ is a suite of PH encryption/decryption transformations available for this object, and $\{A_i\}$ is a suite of access control methods to be used at steps 2 and 4 of protocol 2 (see below). Encryption, decryption and access control methods are full methods, that is, they contain their code bodies. Of course, if o is to be used only with protocol 1, just $\{d_i\}$ and $\{I_i\}$ are required. Conversely, if o is to be used only with protocol 2, then $\{I_i\}$ is not needed.

Example 3. Here is a sketch of a pseudo-C++ implementation of a card object allocated by a healthcare service provider

```
class MedicalCardObject {
private:
// Data fields
        MedicalStruct medical_record;
public:
// Suite of protocol 1 method interfaces (without body)
        void UpdateVaccination(Date vaccination_date);
        int GetNumberOfSurgicalOperations();
        ...
// Suite of privacy homomorphisms (with body)
        CipherData E1(Id client, String data_field_name) {
           // Invoke some access control method for the client,
           // encrypt a data field of the medical
           // record following PH no. 1,
           // and return encrypted data (step 2 of protocol 2).
           ... }
        ClearData D1(Id client, CipherData encrypted_result) {
           // Invoke some access control method for the client,
           // decrypt a result computed on encrypted data,
           // and return clear result (step 4 of protocol 2).
           ... }
        CipherData E2(Id client, String data_field_name) {
           // Same as E1, but for PH no. 2.
           ... }
        ClearData D2(Id client, CipherData encrypted_result) {
           // Same as D1, but for PH no. 2.
           ... }
        ...
// Suite of access control methods (with body)
        Boolean CheckACL(Id client, String data_field_name) {
           // Check whether the client belongs to the
           // access control list for the specified data field.
           ... }
        Boolean Authenticate(Id client) {
           // Run an authentication protocol to check
           // the client's identity.
           ... }
}
```

□

Typically, the suite of available privacy homomorphisms for a given object depends on the type of the data fields. For example,

- For *qualitative* (non-numerical) data fields, we might be interested in a PH preserving the equality predicate, such as the one of example 1; this would allow the client program to make comparisons and compute aggregate data.
- For *quantitative* (numerical) data fields, a flexible PH allowing to perform several arithmetical operations on encrypted data would probably be preferred. Thus, the PH of example 2 would be a good choice.

Note 1 Card objects with several PHs. In a general setting where a data field can be possibly encrypted under *more than one PH*, such a field is stored as cleartext, to avoid maintaining a copy encrypted under each PH. A particular PH is selected when the data field is requested by the client local object at step 2 of protocol 2; such a selection depends on the intended subsequent computation and must be authorized by the card operating system after running one access control method in the $\{A_i\}$ suite. The selected PH will be re-used at step 4 to retrieve the clear result of the computation. □

5.1 Functional limitations

One obvious functional limitation of the described approach is that the client program can only use certain operations on encrypted data. Whereas this may be good for security, it is a functional hindrance. Another drawback is that no single known *secure* PH preserves all usual logical *and* arithmetical operations. So, it is very likely that a local object method needs to be split into different portions each running protocol 2 with a different PH selection.

Finally, a mirage limitation could be alleged because, as it was shown in example 2, usual arithmetical operations on clear data are mapped to unusual operations on encrypted data. This would be problematic if client applications were coded in a procedural programming environment. If object-oriented technology is used, polymorphism allows the code of the client local object to be independent from the PH used by the card object, provided that the proper dynamic libraries implementing PH operations are available to the client.

5.2 Security limitations

First of all, the security of card object data is no longer exclusively tied to the card's physical tamper-proofness. It depends also on the security properties of the particular PH in use. Moreover, if the *same* clear data are encrypted (and exported) by the card under *several* PHs, this might cause unknown weaknesses against known-cleartext or even ciphertext-only attacks. However, this possibility depends on the particular combination of PHs and cannot be generalized.

Another limitation is related to the amount of control that can be exerted on computations on card object data. In protocol 1, the card operating system can check the code integrity of a method before its execution, because the card receives the certified body of every client-invoked method [Gama94]. In protocol 2, execution of the method code is left to the client program. Thus, the card does not see the method code and cannot check whether the code is good and corresponds to a "lawful" method. The card operating system can only exert several kinds of *indirect* control on the computations done by the client program:

- Before delivering encrypted data at step 2, the card operating system can run an access control or authentication method A_i to establish the client's identity. An access control method can be as simple as an access control list checker. An authentication method can be obtained by adapting one the protocols described in [Guil92] for authenticating the card itself.
- The PHs available for a given card object clearly limit the kind of operations that can be performed by the client program.
- The card can check to some extent whether the results received from the client at step 3 of protocol 2 are its own or have at least been computed on its data. To achieve this, the card could encrypt some redundancy with the clear data at step 2. The redundancy scheme must be such that it is preserved by the PH used (for example, multiplying initial data by a secret constant works for the PHs in section 4). Another possibility is for the card's processor to verify with a certain (low) probability each result received from the client.
- At step 4 of protocol 2 the card operating system can eventually decide not to send the decrypted result to the client program. Such a decision could be made by a method A_i implementing a set of rules trying to detect illegal leakage of card object data.
- If computations by the client program are of statistical nature, then the card operating system can use some kind of disclosure control procedure when returning (at step 4) cleartext statistics computed on object data. A statistical disclosure control can be implemented as a supplementary card object method A_i and is designed to thwart inference of individual data from statistics computed on these data; common disclosure control techniques rely on perturbing or partially suppressing the output statistics (see [Euro93],[Denn82]).

6 Conclusion

An approach for increasing the multi-application capacity of smart cards has been described. The goal is to bypass the computational bottleneck that occurs if confidential data and computation corresponding to different applications must all be supported inside the card. Our proposal has also inherent limitations, which make it a (good) complement to other design approaches rather than an alternative. Thus, general card objects of the type assumed in protocol 2 and described in section 5 may coexist with card objects of the type assumed in protocol 1. In practice, this means that client programs will follow either protocol depending on the nature of the object.

Acknowledgment

Special thanks go to Anna Enrich for helpful talks and discussions.

References

[Ahit87] N. Ahituv, Y. Lapid and S. Neumann, "Processing encrypted data", *Communications of the ACM* **20** (1987) 777-780.

[Bire94] A. Birell, G. Nelson, S. Owicki and E. Wobber, *Network Objects* (DEC SRC Report no. 115, Feb. 1994). Also in: *Proceedings of the 14th ACM Symposium on Operating System Principles* (Asheville NC, Dec. 1993).

[Bire95] A. Birell, G. Nelson, S. Owicki and E. Wobber, *Network Objects* (DEC SRC Report no. 115, revised Dec. 1995). Also in: *Software - Practice & Experience* (to appear).

[Bric88] E. Brickell and Y. Yacobi, "On privacy homomorphisms", in: D. Chaum and W. L. Price, eds., *Advances in Cryptology-Eurocrypt'87* (Springer, Berlin, 1988) 117-125.

[Denn82] D. E. Denning, *Cryptography and Data Security* (Addison-Wesley, Reading, 1982).

[Domi96a] J. Domingo-Ferrer, "A new privacy homomorphism and applications", *Information Processing Letters* (to appear).

[Domi96b] J. Domingo-Ferrer, "A provably secure additive and multiplicative privacy homomorphism" (working paper, 1996).

[Euro93] *Manual on Disclosure Control Methods* (Eurostat, Luxembourg, 1993).

[Gama94] A. Gamache, P. Paradimas and J.-J. Vandewalle, "Worldwide smart card services", in: V. Cordonnier and J.-J. Quisquater (eds.), *Proceedings of CARDIS'94* (Lille, Oct. 1994) 141-148.

[Guil92] L. C. Guillou, M. Ugon and J.-J. Quisquater, "The smart card: a standardized security device", in: G. J. Simmons (ed.), *Contemporary Cryptology: The Science of Information Integrity* (IEEE Press, New York, 1992) 561-613.

[Rive78a] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM* **21** (1978) 120-126.

[Rive78b] R. L. Rivest, L. Adleman and M. L. Dertouzos, "On data banks and privacy homomorphisms", in: R. A. DeMillo *et al.*, eds., *Foundations of Secure Computation* (Academic Press, New York, 1978) 169-179.

[Trou91] G. Trouessin, *Traitements Fiables de Données Confidentielles par Fragmentation-Rédondance-Dissémination* (Ph. D. Thesis, Univ. Paul Sabatier, Toulouse, 1991).

This article was processed using the LaTeX macro package with LLNCS style

Smart card use to manage user's mobility

David Carlier[1], Sylvain Lecomte[1] and Patrick Trane[2]

[1] RD2P - Recherche et Développement Dossier Portable
CHR Calmette - rue du Pr. J. Leclercq
59037 Lille Cédex - France
Tel: +33 20 44 60 46 - Fax: +33 20 44 60 45
e-mail: *{david,sylvain}@rd2p.lifl.fr*

[2] Tokyo Institute of Technology
Graduate School of Information Sciences
Ookayama 2-12-1 Meguro-ku Tokyo 152 Japan
Tel: +81 3 5499 7001 - Fax: +81 3 5734 2817
e-mail: *patrick@cs.titech.ac.jp*

Abstract. With computing now essential in companies, administrations, and even in the home, and with the information with the exchange spreading and transport costs decreasing, people are becoming more and more reliant on computers for their job and spare time activities. Based on this, we will define a system capable of managing not only the mobility of the user's computer but more generally the user's mobility. That is to say, communication with the outside whether it be by mobile equipment or from different stations. This last point is increasingly important due to the omnipresence of computers. This system introduces the agent technology. An agent associated to a user is created on the network to assist the user. The smart card solves problems induced from communication between the user and his agent.

Keywords

Mobile computing, smart card, fault-tolerance, transaction

1 Presentation of problems

The system presented in this paper targets mobile users using different machines such as fixed or mobile computers. We will first present the characteristics of mobile computing, then examine the consequences of the usage of the environment.

1.1 Mobile environment features

Physical constraints. The main problems of mobile computing are essentially hardware constraints such as weight and size, wireless communications, and energy autonomy. As a matter of fact, people will soon be able to carry their communicating nomadic object in their pocket. These features imply many consequences as described in [For94] and [Mar93].

Weight and size, both restricted, will always make mobile computers much less powerful and autonomous than fixed computers. This last observation is highly important, as the small battery size necessitates a reduction of energy consumption, *ie* switching to idle mode (slowing clock frequency, managing disk rotation, powering screen off,...), frequent disconnecting of the mobile unit, etc.

A mobile unit is not physically linked to the wired network but communicates with the external world via a hertzian link. The flow of communication is thus not as fast and reliable as with a fixed host using a wired network [Cac94]. Since a mobile user moves, messages designated to a mobile computer are not always sent at the same location. Mobile unit address has consequently to be dynamic and has to change according to its holder location.

The following table summarizes mobility characteristics by comparing mobile and fixed computer features.

Mobile Unit	**Fixed Station**
low powerful CPU	very powerful CPU
low energy autonomy	energy always available
weak communication flow	high rate of communication flow
few available resources	lots of available resources
dynamic address	static address

Table 1. Feature comparison between a mobile and a fixed host

Problems to solve.

1. *Task delegation :*

   It is often more interesting to delegate tasks to a compute server than to carry them out on the mobile unit. It brings additional available computation strength. Furthermore, while the mobile equipment is waiting for the results of a computation, it can either be doing other tasks or be switched to an idle mode to decrease power consumption.
2. *Communication strategies :*

   Communication strategies represent a very important topic to mobility management. A mobile station can only communicate via a wireless link. Thus, transferring data increases latency and power consumption.
   Firstly, the wireless link must not be used as often as the wired network. Task delegation seems to be a good solution. As a matter of fact, delegation enables to adapt task results to another host according to the hardware and the user's preference before their return to the station. It consequently

allows appropriate results to be sent back without any superfluous information. A good way of delegation from a fixed host is also to exchange data with various otherfixed stations. For example, if obtaining a result requires communication with various servers, establishing communication from a fixed host directly linked to a wired network will imply a decrease of the use of the link between the mobile equipment and the wired network.
Secondly, taking advantage of the communication cost asymmetry is judicious [For94], [Imi94]. A mobile station consumes more energy when it is emitting rather than when it is receiving. A nomadic object wishing to update a file on another host can, under certain conditions, request the original document and just send back the differences. In this condition, the station emits fewer data and consumes less energy.

3. *Asynchronism* :

   Mobile stations are often entirely disconnected from the network. In this case, they are unable to perform any task. However, this state does not have to be considered as a failure. Thus, before disconnection, a mobile unit could run a process to prepare itself for a new state and to inform the network about it.
   The disconnection time represents an important part of the mobile station life. There are therefore various asynchronism problems. How can requests to a nomadic object be managed if this object is not connected and how can the results be obtained? Some anwers are described in [Gad94].

1.2 Use of heterogeneous environments

The use of different stations by the same person appears to be quite unusual at present, however it is expected to become more and more common in the future[Gad94].

For example, if an individual has a workstation linked to the Internet in his office, a PC equipped with a modem at home, and a portable computer with a cellular network modem when he is travelling, then it is possible for him to communicate often with the external world by using of his computers. Mobile computing has become more and more vital in the research, developments and commercial domains. Mobile products will however not replace current products such as PCs or UNIX workstations. A person will use any of them according to the circumstances (in his office, on a trip,...).

Using several environments results in many problems. The user communicates using various kinds of links : Internet (relatively high flow), phone link (medium flow), and wireless link (low flow). Moreover, hardware constraints vary according to the machine: energy consumption, man-machine interface, available resources,... Receiving data coming from outside and transmitted in suitable form to the available hardware appears to be very useful. Thus, we observe that only relevant data according to both physical and application constraints has to be sent to a mobile unit in order to improve autonomy and to reduce the wireless link use. All data is consequently available from a station linked to a high flow network.

2 Representation agent based system

2.1 Representation agent use

Agent definition : An agent is a piece of software characterized by its ability to negotiate with other agents in order to achieve specific goals. An agent attempts to accomplish its task even with respect to the changes of its environmment.

Nowadays various software exist able to create agents such as Telescript of General Magic [Whi94] or Obliq [Car94],... Thanks to different tools, an agent can be created from a script and sent to a target host. When it is executed, it can return results. These agents are able to migrate from one station to another. Moreover, after any migration, the execution parameters are preserved, that is, the execution variables travel with it and remain constant. The execution then continues from the instruction just after the migration instruction.

There are other languages which create agents such as Java [Java95], but they are quite different. They are capble of only sending a script to a target host, which is then charged to execute it. No agent migration is considered.

We will especially focus on agents able to migrate from one site to another.

Representation agent features : The object we call a representation agent is an agent dedicated to a unique user, able to act on his behalf and having the following features.

The agents are performed on fixed hosts dedicated to provide them with computing power, memory and so on. We will call these kinds of stations representation agent acceptor sites or, in brief, acceptor sites. They must ensure agent security. That is why, in order to protect these sites, no operation such as *rlogin* or *telnet* may be authorized.

The use of a representation agent brings four main advantages.

1. The agent can be viewed as a fixed intermediary to the external world. Consider a mobile computer user wishing to access data from a fixed server. He sends a request to his dedicated agent which forwards it to the server. The server will then send the requested data to the agent. Thus, it does not have to manage either the user moving or the kind of terminal used. The agent is in charge of redirecting data to the computer used. Since the representation agent is the compulsory intermediary between the user and the outside world, the use of different environments and the moving of the user are concealed by the agent.
2. The agent sends data in a suitable form according to the users preference and hardware.
   For example, suppose a user is logged in on a mobile computer with a small black-and-white screen. When his agent receives a 16-million-color picture

addressed to the user, it will convert this image into a black-and-white picture before forwarding it. Consequently, only vital information is transmitted through the wireless link.

3. A user can delegate tasks to the agent by scripts.
   Sending a script (small program consisting on a set of instructions able to be interpreted) to the agent enables a mobile computer to gain more important processing power. Moreover, a task having to establish several communications or having to send regularly data to the user represents a good use of delegation. For example, if an individual subscribes to a newsgroup and regularly consults new items, it will be useful to create a small script to fetch and forward each item without request from the user.
4. The representation agent can act on the user's behalf even when the user is logged out.
   The agent is always active. So it can receive data or request results from a server even when the user is logged out. In this case, it stores them and waits for the user's connection. For instance, a script may be in process, without any user's connection.

The representation agent features described above correspond exactly to the current problems related to mobile computing and the use of heterogeneous environments (cf. former section). The representation agent brings to the mobile user a new partner assigned to himself.

Representation agent migration. The representation agent is a compulsory intermediary between the user and the external world. Some situations can involve efficiency problems such as latency increase due to user's mobility. For instance, if a person living in Europe travels to Japan, he will communicate with Japanese servers to get local information. If his agent stays in Europe, all data will transmit from Japan to Japan but via Europe. Such extra use of the network will imply a lower flow of communication between the agent and user's parties. That is why, agent migration is necessary when the user is far enough from the agent. Thus, in the previous example, the agent would follow the user to get closer to him and would be located on a Japanese acceptor site. However, two conditions must be satisfied to perform an agent migration. Firstly, no communication between the user and external partners may be on progress so that the agent remains fixed with patners during a communication. Secondly, the user must be logged in so that the agent can know the user's location to get closer to him.

Taking representation agent security into account. The use of representation agents on acceptor sites induces a number of security problems. Some requirements must be taken into account so that firstly an agent cannot voluntarily or otherwise damage an acceptor site, and secondly an agent is not able to read or modify data of another agent located on the same site. Some constraints are therefore applied on the agent processing domain. The idea is to predefine the code of the agent. If a user wants to get an agent on an acceptor site, he requests it to create one from a predefined code. This code can be written in a native code, so the execution speed is improved in comparison with interpreted code. The features and the behavior are, however, predefined: that means the agent cannot damage an acceptor site and cannot use another agent's content.

Opposite to most of agents such as Telescript or Java, the use of predefined code is realistic, because the role of agents is the same for all of them : they must manage communication from/to the associated user and be its script acceptor.

Divide the agent into (1) its code part or agent code and (2) its individual part or personal data *i.e.* viewing the agent as an object will prove very efficient and very secure. The code mainly enables the agent to communicate with the user and the external world, but it is also used to administrate data and scripts sent by the user. Data and scripts allow the user to define its own behavior. As the agent code is the same for all agents, only agent data and scripts are affected by the migration phase.

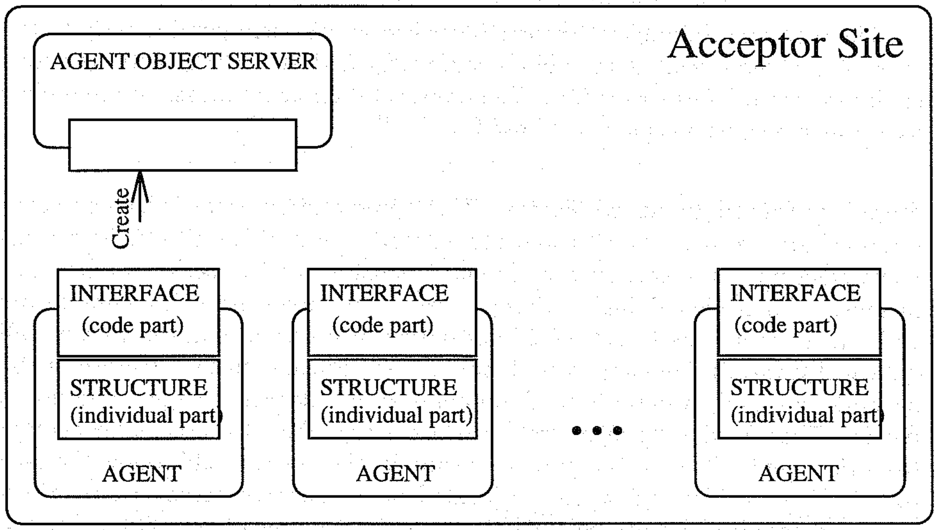


Fig. 1. Structure of an agent

The use of object-oriented concepts enable an easy design of the system. An object is composed of two sets of elements : the interface (code), and the structure (data). Creating an object is achieved by sending a *create* message to an object server as shown in Figure 1.

A representation agent can then be viewed as an object. On each acceptor site, an agent object server is responsible for the agent creation. To create an agent within an acceptor site, a *create* message must be sent to the agent server on the site. Once a blank agent is created, the user personalizes it by sending a *personalize* message with a set of data related to the user and a set of scripts defined by him.

To process an agent migration, a *create* message is sent to the target acceptor site by the agent that wants to migrate. It then personalizes it and transfers its own personal data and scripts. When these operations are correctly performed, it destroys the copy remaining on the user site.

2.2 Network features

Network description. A specific system must be set up to realize the user's representation agents. The network features include :

1. *The acceptor sites :* The acceptor sites are stations designated to provide representation agents with resources such as CPU and memory. Any site can accept several agents but must ensure privacy and integrity with respect to the external world and between agents.
2. *The agent address servers :* These servers are used to contact a person via his agent. As the representation agents are mobile, their addresses cannot be known by the recipient. By a table consisting of an identifier and an associated agent address, the server can reply to provide anyone with the means to contact the required person. These tables can be copied into other address servers since agent migration is a relatively rare event, and so the update operations will be less often executed. Thus duplication decreases both the latency and the number of requests number performed on a same server.
3. *The acceptor site address servers :* These servers allow an agent to access any target acceptor site address when it decides to migrate from its current site to another.

Acceptor sites diagnosis requirements. We have indicated that these agents are located on sites designated to them and are called acceptor sites. The correct working of the communication relies on the ability to detect site failures. As agents can migrate from one site to another under certain conditions, a site failure leads to two kinds of problems : (1) all agents in the faulty site are lost, and (2) an agent can migrate towards a faulty site. To prevent the latter, and to be able to recover from the former, it is important to provide the system with a site-diagnosis [Dah95] methodology, efficient in terms of message latency and adapted to the constraints caused by mobility (the number of sites increases).

The algorithm is divided into two steps. We assume that the network is divided into areas, or *vicinities*. Vicinities are pre-defined and are do not overlap, *ie* a node cannot belong to two vicinities at the same time.

The first step aims to diagnose the fault situation of every vicinity. As nodes in vicinities are predefined, this stage can be parallelized to reduce the information latency. At the vicinity level, nodes are connected in *virtual cells* in order

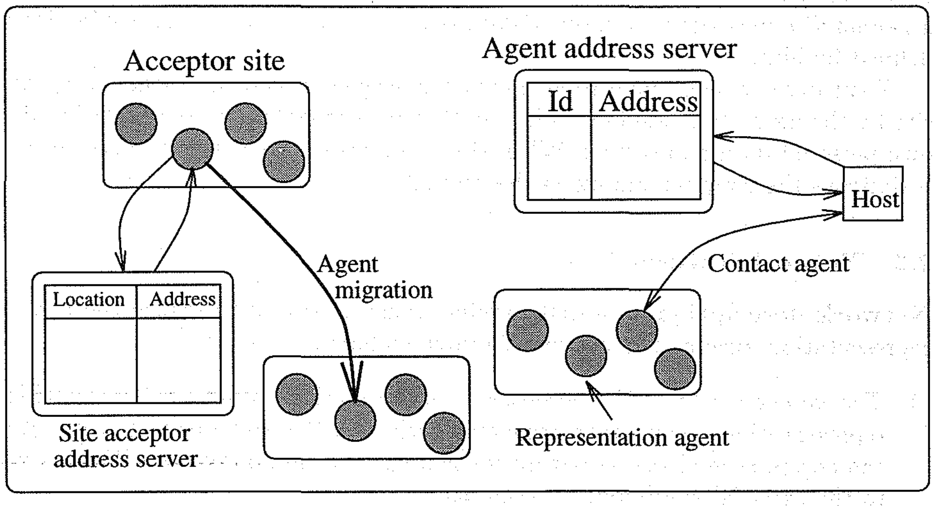


Fig. 2. The network required to use representation agents

not to get a single information on a node while performing a test, but on all the nodes of the virtual cell tested. Suppose a vicinity has, say, 4 acceptor sites or simply *nodes*, identified by $n_0, ..., n_3$, and all nodes are assumed to be fault-free. n_0 tests n_1, n_1 tests n_0 and simultaneously n_2 and n_3 process the same tests on each other. By doing so, they created two virtual cells, made respectively of $\{n_0, n_1\}$ and $\{n_2, n_3\}$ as represented in figure 3. Later, when n_0 sends a message to n_2, it will not only test n_2 but the virtual cell made of $\{n_2, n_3\}$, as n_2 already knows n_3's fault state. n_0 will then know the state of the whole vicinity, in only two tests. If a node in the vicinity is faulty, it does not change the result since, by definition, for a testing round to be completed, a node must have tested another fault-free node in the system. If n_2 is faulty, n_0 goes directly to another node of the virtual cell tested (here n_3). At the end of the stage, any fault-free node of the vicinity will know the fault situation of the vicinity. To complete the first step, a leader is chosen in the vicinity. As any node knows about the fault situation of the vicinity, we assume that the fault-free node whose identifier is the highest is chosen to be the leader.

The next step is to gather all leaders together and implement a classical on-line distributed system-level diagnosis on these nodes [Bian91], knowing that a leader will not test another leader but only another leader's vicinity (data structures are then different from the one proposed in the reference mentioned above). As a leader is by definition *fault-free*, the worst case happens when all nodes in a vicinity are *faulty*, that is, when no leader can be chosen. At the end of this process all leaders will know about the current fault state of the network. Leaders then broadcast that information to all nodes of their vicinity.

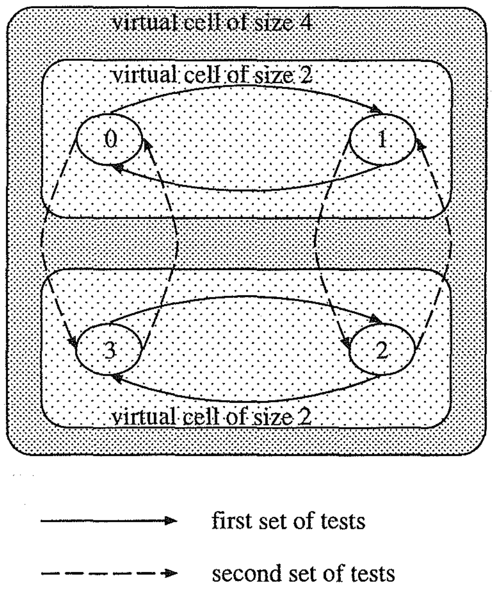


Fig. 3. Vicinity strategy.

For this reason, it is assumed that a node has a perfect knowledge of its physical neighbors.

Agent recovery. As the system relies entirely upon the correct fonctioning of agents, it must also be able to recover from possible agent losses.

Two problems are identified. Reconstruction of an agent after its acceptor site has failed and restarting from a consistent global state without any message losses. The solution proposed also focuses on low-overhead dynamic reconfiguration strategies. Problems are as follow :

Checkpointing [Ach94] must be performed with care so that the saved states form a consistent global state. Suppose agent A (we should say process P_a from agent A) sends a message M_1 to agent B. A takes a checkpoint C_a before sending. B then takes a checkpoint C_b immediately after receiving this message as shown in Figure 4. Subsequently, A fails and restarts from its last checkpoint C_a. At this stage, the system global state is inconsistent for A's local state shows no message sent to B while B's local state indicates that a message has been received from A. This remains true even if B restarts from C_b.

Rollback-recovery [Koo87] from consistent checkpoints may also cause message losses. In Figure 4 again, B sends a message M_2 to A. A receives it and subsequently fails. Both A and B rollback to their respective last checkpoints. B's local state shows that it has already sent M_2. A's local state indicates that it has not been received. The system recovers to a consistent state. However, the channel from B to A is empty. Consequently, M_2 is lost.

The O.O. agent approach, that is, code and data, is particularly suited to provide fault-tolerance taking into account the problems mentioned above. However, to ensure consistency and integrity, an agent has to be modelled as follow :
code: a block of code
data: (a) a data block, (b) a queue of incoming messages and in case the agent was active when the failure occured, (c) its history since last message reception. As the code is the same for all agents, we directly eliminate the code reconstruction problem. To be able to restart properly, a counter of outgoing messages is also required. A change in an agent's local state occurs if either (1) the agent sends a message *i.e.* the outgoing message counter increases, or (2) the agent receives a message, *i.e.* its incoming queue changes, or (3) the agent completes a task, *i.e.* a new checkpoint is performed.

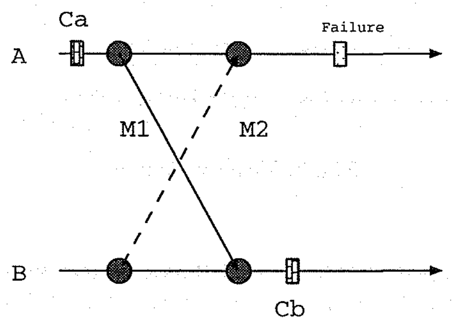


Fig. 4. Identification of problems

To be able to recover properly from a sudden failure, two messages are sent together with the original one. A *shadow* of the original message is sent to the shadow of the recipient. A message informing the shadow of the sender that the sender has sent a message is also sent so as to increase the message counter. Consider Figure 5. To ensure fault-tolerance, four actors are required. The sender (Agent A), its shadow (Agent A'), the receiver (Agent B) and its shadow (Agent B'). The process is ordered as follows : *(first)* Agent A sends a copy of the message he wants to send to Agent B', *(second)* Agent A sends a message to its shadow A' to increment its message counter and finally *(third)* Agent A sends the message to the receiver B. All transactions are simultaneous and atomic. When these operations are completed, Agent A checkpoints and sends a copy of its state to its shadow. Agent A' then replaces its data with the copy it has received. Agent A also keeps a copy in case it has to regenerate the shadow. In case the shadow has to be generated, it just takes both the checkpointed data block and the queue of incoming messages of the agent and transforms itself into a real agent.

3 Advantages of using smart cards

3.1 A compulsory element to ensure security

Securing this agent based system is a real problem. An agent belongs to a unique person. The agent and the sole user must mutually be identified and authenticated before any confidential communication. The use of a designated user smart card thus appears very appropriate. The smart card is currently used to fulfill a part of this task for the GSM [Mou91]. The SIM card, as defined in [Etsi94], allows the subscriber to be identified and authenticated by the network and to generate an encryption key to ensure privacy of data exchanged between the user's mobile equipment and the network.

There is a major difference between a communication in the GSM and a communication in this agent based system. In the former system, several users communicate with the same recipient forwarding messages to the target person, whereas in the latter a user exchanges data only with his agent. The card therefore must identify and authenticate the agent before any communication.

The smart card brings privacy to the system and allows a correct running. When a user logs in, he must send the hardware description of the station used to the agent. This data determines the agent's behavior according to the associated user. That is why, this data must be certified before being sent. The card is the only device able to achieve this task with such security guarantees.

Using a card enables a person to communicate securely with his agent. That is why, the smart card is the appropriate support : secure and portable. The user can easily carry it and use it whatever the terminal, data inside is protected and it allows privacy between the user and his agent. Thanks to the card, people do not communicate with a station but with a person.

3.2 A mean to share data among each station used

It would be very useful to share some data among potentially usable stations. The representation agent could facilitate this, but from mobile equipment it involves an extra use of the wireless link. For this reason, the smart card pools data used by several stations. For example, configuration data can be set within the card for each station device : screen, type of terminal system and other confidential (or not) data needing to be available even if the holder inserts the card into the slot of an off-line terminal.

Moreover, data such as that related to banking applications (bank access or electronic purse) cannot be duplicated. As electronic payment has become widespread and is performed more and more often on a computer linked to the Internet, the smart card appears to be an excellent support to contain securely bank data and to make it available on whatever terminal used.

3.3 Management of the agent from the user's terminal

When the user logs in, he must contact his agent. Without the card, the terminal is unable to access the agent's location. Thus, it must contact an agent address

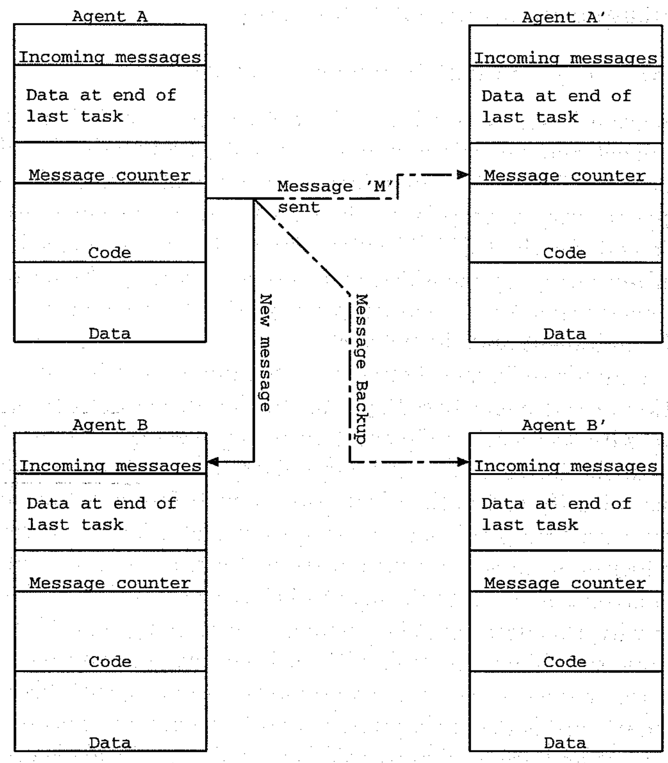


Fig. 5. Process to maintain fault-tolerance

server and wait for a response. With mobile equipment, it involves extra latency and extra use of the wireless link.

The user can carry the agent address inside his card and provide any terminal in which the card is inserted with this information. However, it is necessary that the agent transmits its new address after any migration. Therefore, the agent cannot migrate when its associated holder is not logged in. Furthermore, the agent migrates so as to get closer to the user. If he is not connected, the agent has no way of knowing his location.

4 Making a session more secure

4.1 Transaction problems related to mobility

A communication between a user and another partner or a distant application can be interrupted due to various causes.

The link between the sender and the destination may suddenly be broken. It is all the more possible that a station is using a cellular network to communicate. As the coverage of this kind of network is not global, some areas are not equipped with cells. That is why, when a communication fails, it can be canceled. In this case, either all the data must be re-sent or be continued from the stage before failure. The latter solution is preferable as no extra computation in the user's terminal or no re-emitting (through a wireless link for example) is required.

Another major problem is concurrent access to the information system within a distributed network. Many users could access the same data and modify them at the same time. To take this possibility into account, it is necessary to use a use a lock mechanism or to be able to order the different tasks. As mobile computers are more often disconnected from the network than on-line, they create many problems with data consistency between the mobile station and its network representation. Concurrent accesses are planned to be a part of our future work.

4.2 Definition of transaction

A solution to solve these problems would be to define the notion of transaction.

A session between a user and another interlocutor can be defined as a set of transactions. A transaction is a set of operations enclosed between the following instructions :

'BeginTransaction' and *'EndTransaction'* (commit) or *'AbortTransaction'* (abort).

The main advantages of using transactions include [Hea83] :

1. *Consistency* : A transaction gets data in a coherent state and returns data in another coherent state.
2. *Failure atomicity* : Actions are atomic. If a 'Commit' is received, all actions in the current transaction are validated. On the contrary, if an 'Abort' is received, all actions are cancelled.
3. *Serializability* : Data access must be ordered. The order depends on the action : for example, in case of a *read* access, *write* accesses on the same data must not be authorized.
4. *Persistency* : When a transaction is validated, data is saved on a secure support. In this case, a backup copy is required.

4.3 How to restart after a crash

If we use a transaction language, a online crash between the user and another party leads to an 'Abort' message. At this stage, two problems arise : (1) How to detect the crash and (2) how to restart after this crash. In this section, we will consider how to restart after a failure.

Different problems can be classified in four phases [Gray78]. If the session is interrupted by an 'Abort' transaction (1), actions of the transaction only have to be undone. If a processor correctly stops, we must execute a global undo from the transactions that have been executed on this processor (2). As in an 'Abort' procedure, we have to undo the transaction's actions. In case of uncontrolled crashes (3) (for example a network crash or an entry into an area without wireless network coverage) or a disk failure (4), a warm start (when, for example, the RAM storage is lost) or a cold start (if the hard drive storage is lost) is required.

Presentation of the different kinds of logs. In order to undo a transaction, the old data must be archived. To do this, three different ways are cited in [Ruff92].

1. *Value log use*
   When a transaction is using a piece of data, the value log stores the previous value in a previous value log, and the new value in a current value log.
2. *Transition log use*
   This log stores only the differences between the previous data values and the current values. This solution offers size optimization, and appears to be the better method to undo a transaction. Cancelling a transaction is very easy : the current value is stored in the memory and the difference between the previous value and the current one in the transition log.
3. *Action log use*
   This contains the name, the arguments and eventually the results of each performed action of the transaction. To perform the action, the system must know the inverse operation. So, if an addition is carried out, the system must know that the opposite operation is a substraction. This solution however does not allow easy undoing of transaction, but it is useful to end a transaction after a failure.

Log use to make transaction secure in our architecture. The value log or rather the transition log (low-memory consumption) appears to be very appropriate for the smart card. Only the differences between the initial value and the value computed during the transaction is stored in this log. A simple computation can recreate the initial state to undo the whole transaction. The smart card is appropriate to contain such a log because all data is stored within, so no data is exported outside the card. The security of the data is therefore guaranteed and no extra use of the link between the card and the outside (such as a wireless link) is required. Furthermore, as the transition log is inside the card, the transaction can be undone from any terminal.

It would be very interesting to use an action log within the representation agent. Since the agent is an intermediary between the card and the external application, each action exchanging data can be stored inside the action log. After a communication crash, the transaction is cancelled for the external application whereas the transaction is pending for the smart card. When the transaction is retrievable, the agent will decide whether the transaction can continue or must be undone for the card. Continuing the transaction can bring various advantages especially if the computation already performed has been time consuming and involved several outside communications. In this case, the transaction continues for the card from the last action performed before the crash. On the other hand, a transaction may change with respect to the date, in which case the transaction must be undone for the card. When the link between the external application and the card is restored, the agent again performs the same transaction with the application from the beginning. It acts as the card acted before the crash thanks to the content of its action log. In this case, the card is passive and the agent acts on the behalf of the card. The agent also checks whether the actions sent to it are the same as those sent during the interrupted transaction. If an action is different (for example, a stock value has changed), the previous transaction must be undone for the card. Otherwise if no actions are changed, when all actions of the agent log are done, the transaction can be ended between the application and the card.

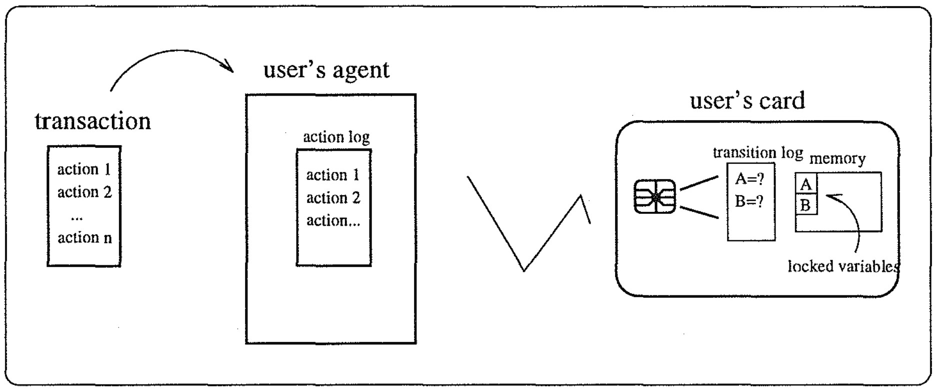


Fig. 6. Structure of an agent

A study case In this section, an application needing a transaction system performed by a user's smart card containing an electronic purse and his representation agent is given. When he wants to buy something in a cyberstore, three partners are concerned by the corresponding transaction : the user, the cyberstore and the bank. The transaction takes action on the following data :

1. the user data : purse-count (in FF), purse-debit (in FF)
2. the cyberstore data : object-value (in \$), value-credit (in \$)
3. the bank data : exchange-rate (FF-\$)

Several invariants between these data can be noticed :

1. purse-debit = object-value $*$ exchange-rate
2. value-credit = object-value
3. new-purse - count = purse-count - object-(value $*$ exchange-rate)

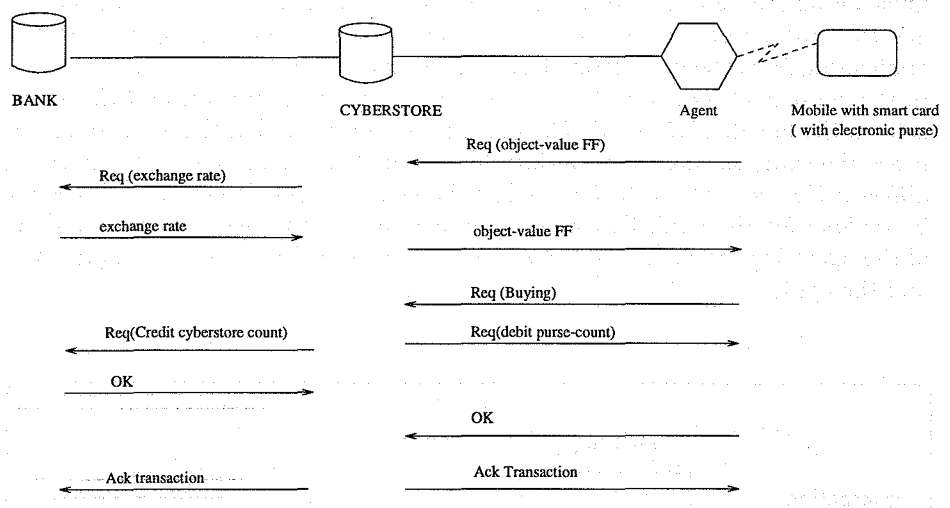


Fig. 7. A short example

When all relations are satisfied, the system is in a coherent state.
In this example, the transaction T is divided into two sub-transactions :

1. T1: The first step of the transaction is to obtain the current cost of the requested object in french francs. To do so, the cyberstore is forced to contact the bank to get the exchange rate from dollars to francs. At the end of this first part of the transaction, there are two data locked (exchange-rate and object-value).
2. T2: The second stage credits and debits the different accounts. If the user decides to buy the object, he thus prepares his electronic purse to be debited and he sends a message to the Cyberstore to confirm the object purchase. Then, the Cyberstore prepares its account to be credited, by sending a credit message to the bank. When the cyberstore receives a response to the effect that the bank is ready, it can validate the transaction by executing a two phase commit protocol, the cyberstore's account is credited and the electronic purse debited.

5 Conclusion

This work aimed at presenting the role of smart cards in mobile computing. So as to be as clear as possible, we emphasized the role of representation agents to make the use of wireless links more adapted to the user's needs. We then focused on a description of a network suitable to the use of agents. Some necessary requirements to make this system acceptable have been presented. A fault-tolerant approach (system diagnosis and agents recovery) has also been proposed insisting on the fact that relying on acceptor sites on which the agents are located is of major concern.

On this basis, one of the main advantages of using smart cards is to ensure security. The use of a smart card dedicated to a single user appears very suitable in providing a good authentification and identification. In addition, smart cards can be used to carry data structure, or the agent address. The smart card allows the user to bring data related to himself to any station he wants to use.

The last section focuses on the use of transactions to make sessions more secure and more adapted to a wireless link. We must be able to control network failure, or processor failure. The proposed scheme easily allows the cancelation or continuation of a transaction (when it is possible) after a failure to preserve all previous processing. To achive this, the use of a transaction language can be appropriate.

References

[Ach94] A. Acharya, B. Badrinath, *"Checkpointing Distributed Applications on Mobile Computers"*, in proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, September 94

[Bha95] K. Bharat, L. Cardelli, *"Migratory Applications"*, in proceedings of ACM Symposium on User Interfaces Software and Technology, November 1995

[Bian91] R. Bianchini, R. Buskens, *"An Adaptative System-Level Diagnosis Algorithm and Its Implementation"*, in Proceedings 21st Int. Symp. Fault-Tolerant Computing, pp 222-229, June 1991

[Cac94] R. Cáceres, L. Iftode, *"The Effects of Mobility on Reliable Transport Protocols"*, in proceedings of the 14th Conference on Distributed Computing System, June 1994

[Car94] L. Cardelli, *"Obliq : A Language with Distributed Scope"*, SRC report 122, June 1994

[Dah95] T. Dahbura, S. Rangarajan, *"A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies"*, IEEE Transactions on Computers, Vol. 44, No2, pp 312-334, February 1995

[Etsi94] *"European digital cellular telecommunications system ; Specification of the subscriber Identity Module - Mobile Equipment interface"*, ETSI, pp 38-47, November 1994

[Esw76] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger, *"The notions of consistency and predicate locks in a database system"*, Communication of the ACM, Vol. 19, No11, pp 624-633, November 1976

[For94] G.H. Forman, J. Zahorjan, *"The Challenges of Mobile Computing"*, in IEEE Computer, pp 38-47, April 1994

[Gad94] S. Gadol, M. Clary, *"Nomadics Tenets - A User's Perspective"*, Sun Microsystems Laboratories Inc. Technical Report, SMLI-TR-94-24, June 1994

[Gray78] J. N. Gray, *"Notes on Database Operating Systems"* Lecture notes in comtuter science, Vol. 60, Operating systems, An advanced course, Edited by R. Bayer, pp 393-481, 1978

[Guer92] R. Guerraoui, *"Programmation Repartie par objet: Etudes et propositions"*, PhD thesis, Universite Paris-Sud, 1992

[Hea83] T. Header, A. Reuter, *"Principles of transaction oriented database recovery"* ACM computing surveys, Vol. 15, No4, pp 287-317. December 1983

[Reed79] D.P. Reed, *"Naming and synchronisation in a decentralized computer system"*, PhD thesis, Massachusset institut of technology, Cambridge, 1979

[Imi94] T. Imielinski, S. Viswanathan, *"Adaptative Wireless Information Systems"*, in proceedings of SIGDBS Conference, Tokyo, October, 1994

[Iso7816-1] International Norms ISO 7816-1, *"Physical Features of the Smart Card"*

[Java95] *"The Java Language Environment : A White Paper"*, Sun Microsystems, May 1995

[Koo87] R. Koo, S. Toueg, *"Checkpointing and Rollback-Recovery for Distributed Systems"*, in proceedings of IEEE Transactions on Software Engineering, Vol. SE-13, No 1, pp 23-31, January 1987

[Mar93] B. Marsh, F. Douglis, R. Cáceres, *"System Issues in Mobile Computing"*, Technical Report TR94-020, Matsushita Information Technology Laboratory (Princeton), February 1993

[Mou91] M. Mouly, M.B. Pautet, *"The GSM System Mobile for Mobile Computing"*, 1991

[Ruff92] M. Ruffin, *"KITLOG, un service de journalisation générique"*, PhD thesis, Paris VI university, Paris, 1992

[Whi94] J.E. White, *"Telescript Technology : The Foundation for the Electronic Marketplace"*, White Paper, General Magic, 1994

This article was processed using the LaTeX macro package with LLNCS style

How Smart Cards Can Take Benefits from Object-Oriented Technologies*

Patrick Biget, Patrick George, and Jean-Jacques Vandewalle

RD2P: Recherche et Développement Dossier Portable
CHRU Lille, Hôpital Calmette, Rue du Prof. J. Leclerc, 59037 Lille Cedex, France
Tel.: +33 20 44 60 44, Fax: +33 20 44 60 45, E-mails: *{biget, pg, jeanjac}@rd2p.lifl.fr*

Abstract. We submit that a key enabling technology for non-predefined and multi-purpose smart cards is object-oriented technology. Object-oriented concepts and skills have proved their efficiency to model, design, and implement information systems made from small components. As smart cards become more and more personal environments for multiplicity of services, they need to allow downloading of unpredictable services and to be easily integrated into information sytems. This paper presents usage of object-oriented technologies to implement a *generic smart card operating system* and to provide a *card object adapter* to access smart card services from distributed object-oriented information systems based on CORBA architecture.

Keywords. Smart card operating system, Smart card integration, Object technologies, CORBA.

1 Smart Cards and Object-Oriented Technologies

This section starts with a short survey of today's limitations of smart card applications [24]. It focuses on the difficulties to realize a generic smart card operating system able to download new services all along the smart card life cycle. Considering the benefits of object-oriented concepts and the fast development of interoperable objects as basic components to construct large scale information systems, we explore these technologies to solve the problem of code downloading and smart card integration into information systems.

1.1 Limitations of Smart Card Operating Systems and Applications

This section lists current multi-application smart card limitations which restrict use of smart cards as servers for a multiplicity of personal services which could be *dynamically* downloaded all along the card life cycle:

* This paper presents the first results of the OSMOSE project (Operating System and Mobile Object SErvices) conducted by the research laboratory RD2P of the Universities of Lille I and Lille II, the French national research center CNRS and the Gemplus company.

1 Present multi-application smart cards are issued by a single organization (called issuer), which is responsible of the split of the smart card non-volatile memory among the different users implied in the application, the creation of the access rights attached to each user, the loading of data which are specific to the card holder, and the delivering of the smart card to the card holder. These operations are done during the initialization/personalization process before smart card starts to be used. The reasons are the small size of the non-volatile memory and the absence of memory management facilities needed for inter-application security. Thus, the configuration of the basic building blocks of smart card security, authorized users, and data storage structure is set and cannot be altered after during the smart card life. *Smart card configuration is not flexible.*

2 Smart cards are not able to accommodate executable codes loaded by different application suppliers. Functions that a smart card can perform are those of its operating system stored in ROM during manufacturing, and possibly those added in non-volatile memory at the personalization process by the issuer. No longer after the smart card can load a new code to adapt its behaviour to external changes. *Smart card functionalities are frozen.*

These issues have to be overcome to develop non-predefined and multi-purpose smart cards offering many services which are issued by different providers and available to fulfil card holder's needs. The future of smart cards is to become real mobile computers integrated to heterogeneous networks and distributed systems [3].

1.2 Smart Cards as Elements of Distributed Object-Oriented Information Systems

Object-oriented techniques, which originated in the area of programming, have been widely applied in all areas of computer science including software design, database technology, artificial intelligence, and distributed system. Large scale applications often use the paradigm of a *global object space* [2] to qualify the set of resources available to clients. This object space provides dynamic and uniform interactions between clients and distributed processes. Utilization of the object paradigm is promoted by its good characteristics to support efficient and transparent interoperability upon heterogeneity.

Smart cards are now of central importance in a broad range of wide area applications in which they make available a plethora of new services capable of being tailored to suit the card holder characteristics or needs. As personal and portable information systems, smart cards can be seen as parts of the global object space, and can take benefits by importing object-oriented features. In this section we will try to show that using object-oriented techniques in building smart card applications seems to be promising.

Overview of Object-Oriented Concepts Object orientation [12, 26] is an abstraction mechanism in which the world is modeled as a collection of independent objects which communicate with each other by exchanging messages. An object is characterized by a *private* static part, a set of data, and a dynamic part, a set of operations which works on these data. Data and operations are *encapsulated* into a same entity, called an *object*. The data and the implementation of operations are hidden. Accesses to data are only possible via the set of operations which constitute the *interface* of the object. How an operation is performed is under the responsability of the object itself. Calling an operation onto an object is like a request, a *message* sent by a caller to a service provider, asking for the execution of an action.

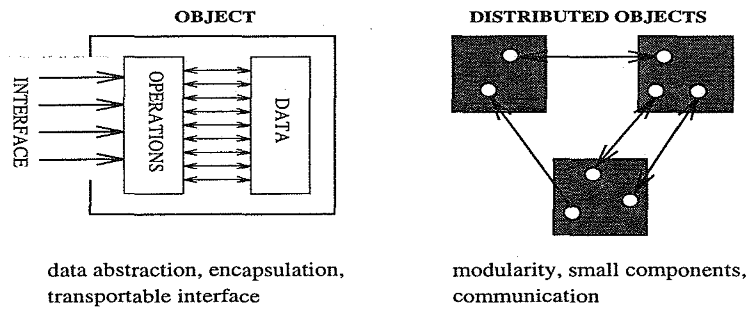

OBJECT
INTERFACE
OPERATIONS
DATA
data abstraction, encapsulation, transportable interface
DISTRIBUTED OBJECTS
modularity, small components, communication

Fig. 1. Object principles

The object-oriented principles listed below offer qualities to design and implement information systems as a space of distributed components (see Fig. 1):

- **data abstraction** implementation and internal representation of an object are unknown from users
- **encapsulation** an object encapsulates its proper state by data and a set of operations applicable on it, it controls itself how an operation is carried out onto its data
- **transportable interface** the interface of an object is independant of its state and implementation and can be distributed to users
- **modularity** an object is self-contained and can be modified independently of other objects
- **small components** objects structure the whole world as a collection of small and independent components
- **communication** objects communicate by exchanging messages which can be transported over different address spaces

Modelling Smart Card Services as Objects Roughly speaking, a smart card is like an object, it stores data which are accessible only through an interface which is the card set of commands: if anyone wants to access data he/she has to request "politely" the smart card interface. In fact, smart cards are such objects: operation implementation, data internal representation, and internal state of smart cards are encapsulated by the smart card ROM code, also called the *mask*. This mask defines the smart card interface as the set of commands available for the outside world and controls internally the execution of requested operations. It is a strong argument for using smart cards to contain secure services in a heterogeneous environment. But this way of running makes different application data have the same interface (the smart card operating system set of commands) though data are used to provide different services. In generic smart cards each service should be seen as a proper object implemented into millions of smart cards and offering its own interface to embedded data.

The growing researches – and before long the market – for generic smart cards can benefit from object-oriented technologies in terms of software development, reliable and secure execution environment, code sharing and reuse. Advantages of object-orientation for software engineering are well-known because it enforce writing code in small, and self-contained units. That brings two important characteristics for computer security [20] discussed below.

Modularity Smart-card services designed as objects are small *modules* which perform independent tasks providing *testability* and *auditability* since a single module with well-defined inputs, outputs, functions can be tested exhaustively by itself, without concern for its effects on other modules.

Encapsulation and information hiding Encapsulation and information hiding means objects independence and object interactions restricted to well-defined interfaces. It is a form of isolation in which every object operation can be controlled to only operate onto the object data and to call only the other internal object operations. Limited exchanges of information can be restricted to messages sent to other objects which will execute operations and return results without revealing how they have completed their operations (see Fig. 2).

Interfacing Smart Card Objects Interfacing smart card with information systems to develop card-based applications is a major challenge to promote the use of generic smart cards as personal, portable and secure servers.

So far, smart card interfaces are carried out by specific libraries provided by smart card manufacturers. The library implements a set of functions – an Application Programming Interface (API) – which correspond to the set of commands of the smart card operating systems. As the smart card interface cannot be changed since it is written as a mask in ROM code, this solution was satisfactory. But, with generic smart cards able to download new services in the form of objects, the smart card interface is the sum of all the downloaded object interfaces. Furthermore this set of object interfaces will be continuously modified at

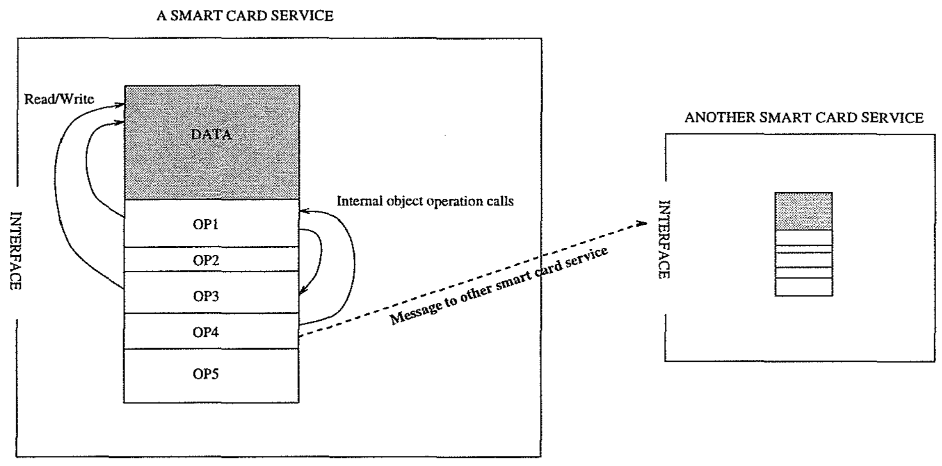

A SMART CARD SERVICE
Read/Write
DATA
OP1
OP2
OP3
OP4
OP5
Internal object operation calls
Message to other smart card service
INTERFACE
ANOTHER SMART CARD SERVICE
INTERFACE

Fig. 2. Smart cards services encapsulation and information hiding

each new object downloading or deleting. Thus, it becomes impossible to define in advance an interoperable API for all the possible smart card services since, at a given time, we cannot know all the future services which will be downloaded into a smart card.

Our approach to that problem is to use independent interface descriptions of smart card services to configure at real-time a generic smart card server. Object-orientation provides well-defined smart card service interfaces and consequently a good modelization of the smart card interface. Moreover, these smart card service interfaces are independent of the service implementation. They can be described in an implementation-independent language often called an *Interface Description Language* (IDL). Two usages of these IDL descriptions are possible:

1. they can be compiled to produce *stubs* for interfacing smart card objects with application programs, or
2. they can be used to dynamically construct object requests to smart cards which are transported through a generic smart card server.

The second solution is more flexible since it does not require to recompile the application programs. A smart card client can configure itself to the smart card interface according to the set of the smart card service interfaces (see Fig. 3).

1.3 Outline of the Paper

We envision that smart card operating systems should adopt object-oriented technology [24, 17, 4]. The object-oriented approach is useful to encapsulate

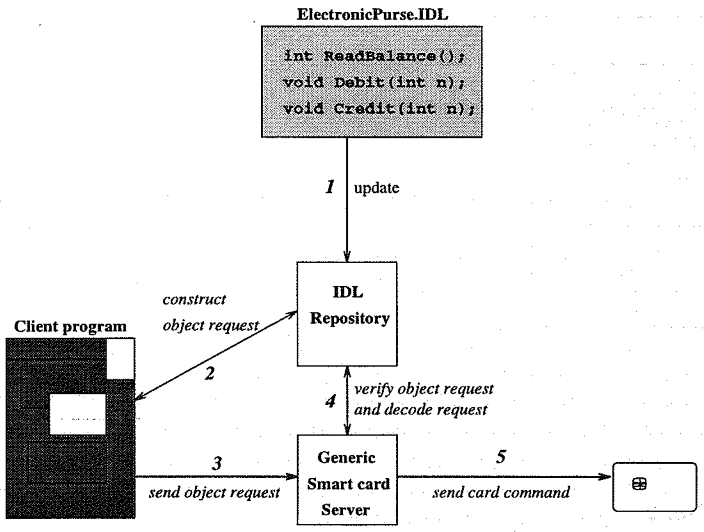


Fig. 3. Smart card object description in IDL to configure client program

into a single address space the execution of an operation onto a smart card service. Secure interpretation of the code of an operation is performed by delimiting *read/write* operations onto the object's data space and *call* operations onto the object's code space. Moreover, transportable descriptions of smart card services can be brought by object interfaces described in an *Interface Description Language*. Such interfaces can be distributed to millions of smart card users and used to dynamically invoke smart card objects from their applications without recompiling them. This last point has been prototyped on a CORBA-based architecture by implementing a *Card Object Adapter* (COA).

In section 2 we present generic smart card principles and reveal object-oriented technologies useful to design appropriate operating system. In section 3 we describe the CORBA architecture and we propose a generic gateway to interface smart cards with information systems based on CORBA. Finally, we draw some conclusions and outline future works.

2 Generic Smart Card Operating System

The work presented hereafter has been partially carried out with the support of the European Open Microprocessor Initiative (OMI) under the project name "CASCADE" (Chip Architecture for Smart CArds and portable intelligent DEvices). This project aims at providing the basis for a new smart card generation based on a 32-bit RISC processor [19]. Inside this project, RD2P is in charge of architecture evaluation tools and smart card operating system development.

2.1 Generic Smart Card Principles

Basic principles of our generic smart card are based on the definition of a new flexible card life cycle which integrates dynamic code downloading [18], and secure execution environment.

Card Life Cycle To define a card life cycle, two kinds of operations must be distinguished. The first ones are one-time operations, like manufacturing and personalization. These operations could happen only once and their impacts on the smart card organization cannot be modified later during the card life. The second ones are multi-time operations, like service execution request. These lasters can happen all along the life of the smart card.

Life cycle of today's cards consists in three steps. The first one is obviously the manufacturing and is handled by the smart card issuer. In cooperation with a service provider, they personalize the smart card with specific services and, possibly, personal data dedicated to the card holder. This being done, the smart card is valid and users can request, at any time, the execution of a service functionality. The term *users* is not restricted here to the card holder but comprises all the entities using the smart card services during the exploitation phase, e.g. merchants, and banks for an electronic purse (see Fig. 4).

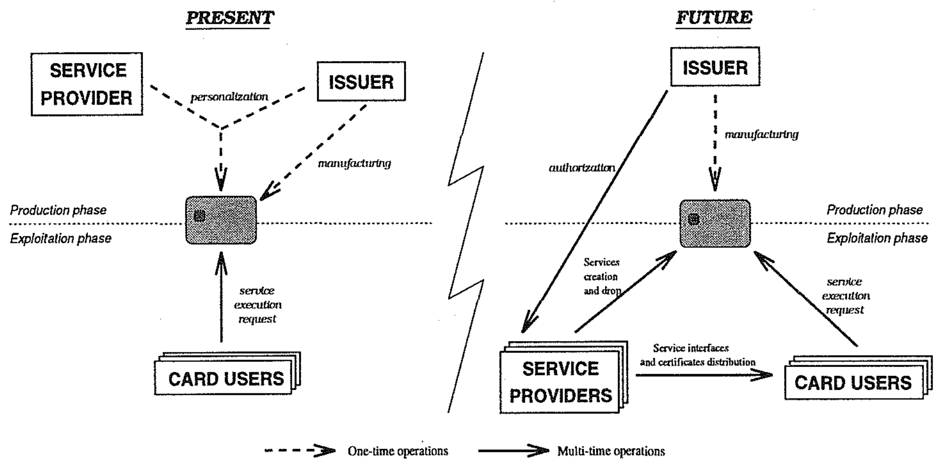


Fig. 4. Card life cycle evolution

This way of running implies two major constraints:

- no possible evolution (in term of new services) of smart card after the personalization,

– the service provider is dependent of the issuer for writing the code of his/her services.

To remedy these drawbacks, a new approach is necessary. On the one hand, the card holder needs to have the opportunity to request service providers to add new services into his/her card at any time. On the other hand, the service provider and the issuer roles must be clearly split.

The new card life cycle concept enables a more flexible usage of smart cards. The issuer manufactures smart cards and authorizes the service providers he/she trusts to download services. The authorized service providers write service codes, download services into smart cards, and distribute service interfaces and certificates to users. Using these certificates, users may request smart card service operations into their applications.

The direct consequence and the main improvement coming from this new card life cycle is the ability to dynamically download code. Smart cards become real interactive and personal portable devices that card holders can manage. They decide to add or remove services according to their needs or requirements. The card content is built on its owner's demand.

Services are loaded into the smart card as objects. The interfaces of a service is composed by the set (or a subset) of the object operations. A service can also be seen as a set of operations corresponding to its external functions.

Services are loaded in the non-volatile memory of smart cards, that implies a new memory management to suit dynamic object creation and deletion. Moreover, we plan to assume that a service could evolve in such a way that its set of operations could grow. Memory management must therefore be quite dynamic as well.

Security Relevant Features While codes can be dynamically downloaded and even if service providers must be authorized by the issuer to do that, we are obliged to imagine that intruder's code could be able to slip inside the smart card (Trojan Horse). Consequently, we have to consider the code not to be reliable. That is why its execution must be secured to prevent forbidden accesses.

Effectively, with this new architecture, different service providers are intended to download code into the smart card. Each of them does not know the others and all the more so does not have any agreement with them in such a way that they could trust each other. Then, if the smart card operating system does not ensure that code will be executed securely, nobody will accept to store data inside the smart card. By securely, we mean that the code of a service is not able to address data of another service nor to modify its code.

The solution is based on the use of an interpreter taking advantages of object-oriented technologies and particularly the encapsulation principle. The use of an interpreter is due to the impossibility to check out *a priori* intrusion attempts. Read/write and call operations can in fact reference memory dynamically. Thus, the verification must be done at execution time. The only way to do so comes trough an interpreter verifying (see Fig. 5):

1. for each memory access if it is targeted legal location i.e. the data space of the object
2. for each subroutines call if it is to another operation of the same object i.e. inside the code space of the object

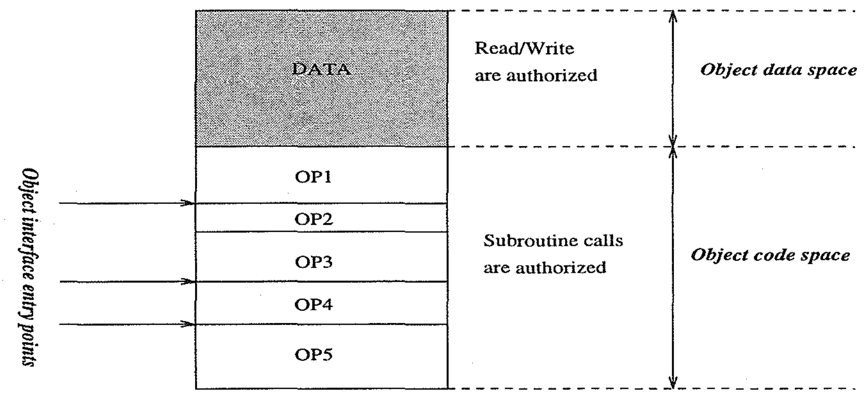

DATA
Read/Write are authorized
Object data space
OP1
OP2
OP3
OP4
OP5
Subroutine calls are authorized
Object code space
Object interface entry points

Fig. 5. Object memory space

Thanks to data encapsulation, the only entry points to a service are defined by the object interface which can be all or a subset of the object operations. The smart card operating system interface accepts only operation executions which are available at the object interface and verify at execution time every memory access to be include into the object memory space. Data cannot be directly handled since the user does not know nor where and how they are stored nor how access them without passing through the object interface.

2.2 Smart Card Object Interface

The operating system is made of three logical layers, each of them being in charge of specific functions (see Fig. 6). The first one manages smart card hardware resources which comprise the volatile memory, the non-volatile memory and the I/O. This is the basic layer which provides facilities management for the other layers. Security is undertaken by the second layer, the *interpreter*, that embodies several tools to load and execute code in a reliable way [7]. The interpreter will be described in details in the following section. The *User Interface* is the sole external access to the smart card. It fulfils two separate tasks: user authentication and object management.

Users have to be authenticated for each object, each service. An authentication protocol, using public key cryptography algorithms, could be used to mutually authenticate users and smart cards. At each object downloading, the

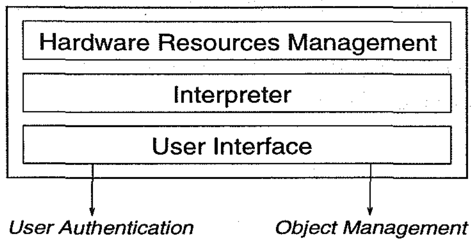


Fig. 6. Generic smart card operating system architecture

service provider could deliver to users a certificate giving them execution rights on service operations. In the same way, service providers could receive certificates from the issuer to download smart card services.

Concerning object management, only three basic commands are recognized by the smart card interface:

- `create_object`
- `drop_object`
- `execute_operation`

The `create_object` command is used for downloading service into the smart card. Before sending this command, the entity must have been authenticated by the smart card (only an authorized service provider is intended to do so). To create an object, we need a reference, a secret, the size of the data (and possibly their initial values) and the code of the operations. The reference is a unique number. As all references must be distinct, they must be distributed by a single authority who may be the issuer but also a public authority. The code of operations are transmitted in a special format according to the interpreter language (see 2.3).

The `drop_object` command is the opposite procedure. This command just requires the object reference. Code as well as data are cleared. The operation can only be performed by the entity who has created the object. Obviously, it must have been authenticated by the smart card previously.

Card users can only order the last command: `execute_operation`. By giving the object reference, the operation number and input parameters if any, the code of the adequate operation is executed. Unlike the two previous commands which just send back status information, this one could send back data to the user. The reception of this command by the smart card will always trigger the interpreter running.

2.3 Interpreted Execution Environment

For security reasons a secure interpreter is used to execute the code of an object operation (see 2.1 *Security Relevant Features*). When the smart card operating system receives an `execute_operation` onto an object, it sets software security registers to restrict memory accesses to object memory space. Then, it launches the interpreter onto the entry point of the requested operation. The interpreter executes the code of the operation with *on-the-fly* address boundaries checking accordingly to the security registers. The security registers are inacessible from the outside and the machine code of the interpreter is written in ROM, thus it cannot be hijacked to bypass address boundaries checking.

Virtual Machine The interpreter defines a virtual machine which is a stack-based abstract machine made of the following logical components (see Fig. 7):

- a *data stack* `ds` for data evaluation using Reverse Polish Notation
- a *return stack* `rs` to temporarily save return address at each subroutine call
- a *data space* `DataSpace` to store object data
- a *code space* `CodeSpace` to store object operation codes
- a set of security registers `BDS LDS BCS LCS` to control memory accesses

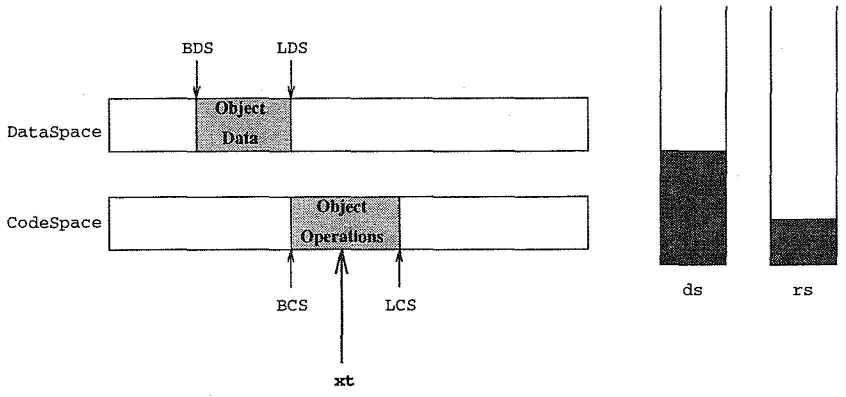
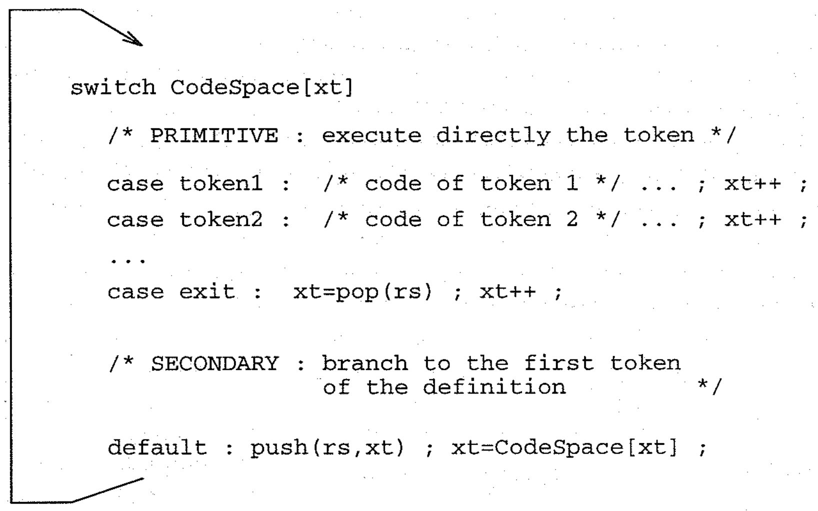


Fig. 7. Virtual machine

Interpreter Implementation The interpreter implements a *Threaded Interpretative Language* (TIL) with the technique called *Token-Token Type* [8]. In our implementation, an instruction is represented by a *token* (or a number) which can be:

- *a primitive:* a basic instruction defined in the core of the interpreter using the machine language of the host processor, or
- *a secondary:* a sequence of tokens which may contain primitives or other secondaries in its definition.

Each object operation defines a secondary which is written as a list of tokens. To execute an object operation, the interpreter starts with an *execution token* (xt) set to the first token of the secondary, and execute in turn each primitive in its definition. If the interpreter encounters a secondary it performs a subroutines call using the return stack for saving the return address. All secondaries terminate with the primitive **exit**. The action perfomed by **exit** is to pop the value on top of the stack and load this into xt. If there is no address on top of the return stack the execution is ended.



```
switch CodeSpace[xt]

   /* PRIMITIVE : execute directly the token */

   case token1 :  /* code of token 1 */ ... ; xt++ ;
   case token2 :  /* code of token 2 */ ... ; xt++ ;
   ...
   case exit :  xt=pop(rs) ; xt++ ;

   /* SECONDARY : branch to the first token
                  of the definition         */

   default : push(rs,xt) ; xt=CodeSpace[xt] ;
```

Fig. 8. Interpreter execution scheme

This implementation saves space and provides flexibility to implement on-the-fly boundaries checking and stack overflow control. The primitive token set is defined as the basic command set of the virtual machine. In order to fit with target processor capabilities, implementation of addresses, numbers, tokens might be customized to improve the interpreter speed.

Compilation Line The following techniques keeps the interpreter simple and leads to high-speed execution [15]:

- unique format for numbers (all are integers)
- arithmetic expressions in Reverse Polish Notation
- TIL language without any string-identifier (all is token)
- TIL language without syntactic analysis

But these techniques are difficult to handle for a programmer. Fortunatly, they can be taken in charge by a compiler which translates a high-level language into the TIL language (see Fig. 9).

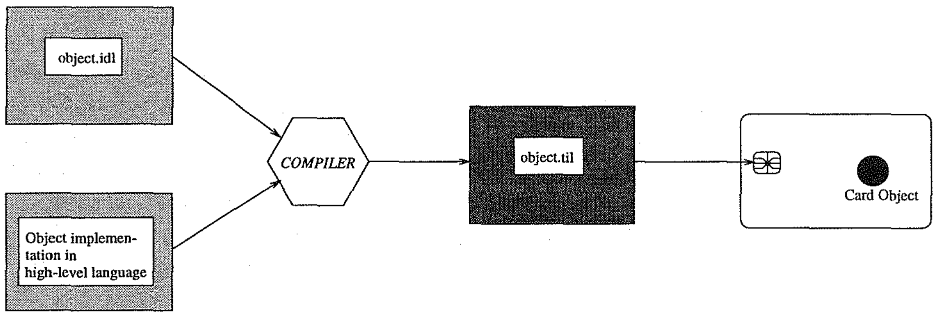


Fig. 9. Compilation line

3 Card Object Adapter

In this section we address the problem of smart card integration into information systems. Basically, smart cards and applications run onto heteregeneous environments: hardware, operating system, interface, communication protocol, programming language, and services availability. Considering smart cards as personal and portable object servers, we identify the object technology as a key enabling technology for application interoperability. Object-orientation offers well-structure modularity and clearly defined interfaces for defining services in a heterogeneous environment.

In this domain, a consortium of object-oriented software vendors, developers, and users – the *Object Management Group* (OMG) [22] – has been founded to promote the use of objects for the development of interoperable distribued computing systems. They proposed an *Object Management Architecture Guide* [21] as a reference model in which each piece of software is represented as an object, communicating with other objects via the *Object Request Broker* (ORB). The ORB is the key communication element providing mechanisms by which objects transparently make request and receive responses.

This architecture is a good model for developping interoperable applications in which smart cards should take place to bring personal services to global applications. Following, we present the core of the ORB technology, and we describe a generic tool to interface smart cards with ORB.

3.1 CORBA Architecture

The *Common Object Request Broker Architecture* (CORBA) [16] is the proposed standard for ORB. The basic service provided by CORBA is delivery of messages (requests) from a client to an object implementation, and delivery of a response (return or exception) to the caller. The *client* is the entity that wishes to perform an operation on an object and the *object implementation* is the code and data which actually implement the object.

The messaging support provided by CORBA is made possible by a single specification language, called IDL (for *Interface Description Language*) which is used to specify interfaces independent of execution languages. This language may be mapped to any implementation language. The same IDL source is compiled to produce code for the client of the object (*stubs*) and code for the object implementation (*skeletons*). These codes assume the marshalling and unmarshalling of requests and responses invisibly transported through the ORB (see Fig. 10).

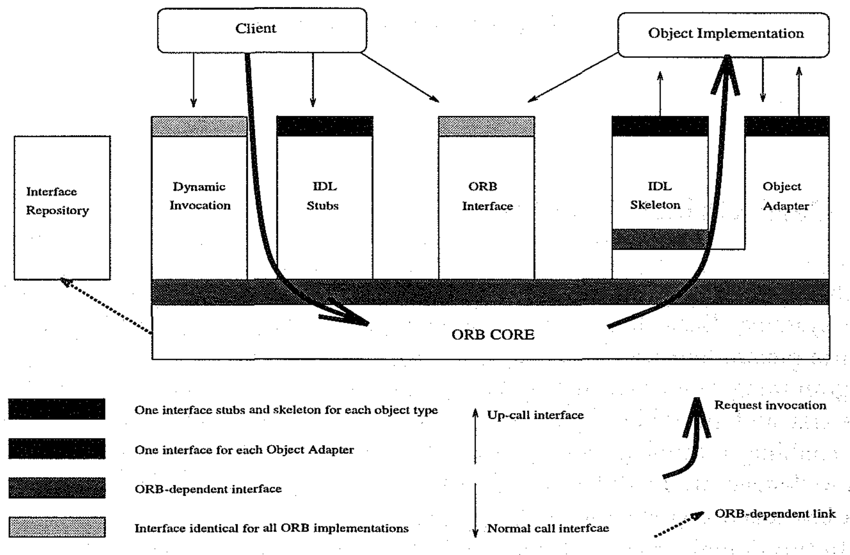


Fig. 10. Common Object Request Broker Architecture

When the client has the stubs corresponding to the objects it wants to access, it is a *static* calling scheme. Moreover, CORBA provides a *dynamic* calling scheme in which the client can discover available interfaces of objects, construct a request for use of a service, send the request, and retrieve the response. This is all possible without any *a priori* knowledge of the interfaces at compile time, i.e. without any stub for the object which is accessed. This mechanism, called *Dynamic Invocation Interface* (DII), uses run-time discovery of available interfaces from a standardized object services called the *Interface Repository* (IR). The IR maintains a fully dynamic representation of interfaces of all objects available in the distributed computing system.

An important extension of the ORB core communications layer is the *object adapter*, which is an abstraction mechanism for hiding the details of object implementation from the messaging substrate. The object adapter adapts from the generic ORB model of objects to the varieties of implementation details of objects in various different schemes. For example, objects may be implemented as application code within operating system processes, or they may be implemented as objects in object-oriented databases.

3.2 Generic Smart Card Requirements

The problem of interfacing smart cards with CORBA architecture is that the object implementation of a card service is inside a smart card which uses specific communication protocols [10]. As a result, a smart card object cannot be directly plug in above the ORB substrate. It is required to translate ORB object messages to smart card messages which can be described uniformly by *Application Program Data Units* (APDU) commands and responses. That leads to write an object adapter for smart cards, we call a *Card Object Adapter* (COA), providing interoperability between CORBA applications and smart cards.

The second problem is that card client programs and the COA do not know in advance interfaces of the smart card services. We take advantage of transportable descriptions provided by implementation-independent object interfaces which can be described in IDL. IDL descriptions of smart card services can be distributed by service providers to millions of smart card users to update their Interface Repository. Both client program and COA can request the IR to dynamically operate with the smart card services.

3.3 COA Implementation

The COA is a generic gateway between CORBA and smart cards. It provides an object interface of the smart card services by creating "proxys" which are CORBA-compliant objects which act as surrogate objects of smart card services. A proxy, or smart card object, is dynamically created at the first client binding to a smart card service. The COA uses the IR to retrieve the IDL description of the smart card service and to construct the proxy. Next operation invocations to the smart card service are supported by the proxy which controls the correctness of the request thanks to the smart card object signature found in the Interface Repository, translates requests and arguments to smart card commands using the *Card Code Repository* (CCR), transmits commands to the smart card and sends back card response to the client.

The CCR contains the translation of each operation invocation to a set of APDU requests. Translation is described by a script language which converts operation arguments to smart card command arguments and smart card returns to operation results. By initialization of the CCR, the COA is able to interface any smart cards, from object-oriented to legacy ones since the CCR can be configure to interface any specific smart card operating system set of commands.

3.4 Generic Smart Card Application Scenario

In this section we illustrate an example of a CORBA-based smart card application (see Fig. 11).

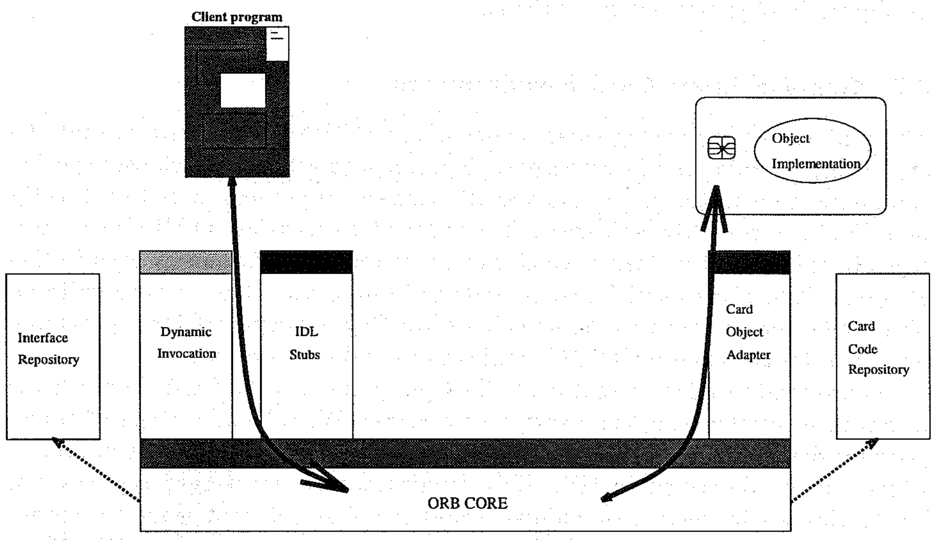


Fig. 11. CORBA-based smart card application

1. Let every smart card user have an ORB as basic object messaging substrate of their information system. To design a smart card application, a smart card reader/writer is connected to a machine, and IR, COA and CCR are available.
2. Service providers implement services into smart cards and distribute IDL descriptions, CCR scripts and access control certificates to smart card users.
3. Users udpate their IR and CCR. They can compile IDL descriptions to have stubs they have to include into their application programs with compiled object requests. But they can also use the DII to dynamically construct smart card object requests at execution time.
4. When a smart card is connected, the user presents his/her certificate to have access to a smart card service. In case of success, smart card object requests are sent to the COA which performs request and return translations according to the CCR scripts.

4 Conclusion and Future Directions

4.1 Summary

Object-oriented technologies for generic smart card operating system and smart card integration tools are proposed and prototypes have been developed. Object-orientation is capable of handling the twin challenges, genericity and plug-and-play, in large scale application integrating smart card services. The main contributions in the proposed system are: encapsulation and interpretation to provide dynamic code downloading and secure execution environment inside smart cards, and transportable smart card service interfaces to configure client applications.

4.2 Status of the OSMOSE Project

The generic smart card operating system has been implemented in the form of a simulator on SUN Sparcstation under Solaris 2.4. It has been also implemented onto ARMulator for Windows NT, a program which emulates the instruction set of the ARM 7 processor [1] which is the processor choosen for the CASCADE project. In CASCADE, a telecom application has been tested using this prototype.

The COA has been implemented on SUN SPARCstation under Solaris 2.4 with IONA Orbix 1.3 [9], a commercial CORBA implementation. It is intended to be used to provide a dynamic and generic access to smart cards from World Wide Web browers through a generic gateway between WWW and CORBA [14].

We are working to finish stable prototypes of the OSMOSE system in the near-term which fully incorporate the features described in the paper. We plan to set definitively the high-level language used to develop smart card objects and the instruction set of the TIL. We also plan to implement the CCR using *CorbaScript*, a interpreted scripting language used to apply any operation on any CORBA object [13].

4.3 Future Works in the OSMOSE Project

There remains much to do in the field of generic smart cards. A primary goal is to demonstrate the effectiveness of OSMOSE prototypes in the near-term providing more implementations, experimentations, and benchmarks. Future research includes:

- further research in types of cooperation techniques among smart card services
- research in security protocols for:
  - authentication certificate distribution to trust service providers and smart card users
  - capabilities distribution to control access rights to smart card and application objects
- research in transaction protocols for reliable communications between smart cards and informations systems

Acknowledgments

We gratefully acknowledge Philippe Merle from the university of Lille I for his help and support in the COA development. Also, Thomas Chamussy, graduate student at the university of Lille I, for his participation to the development of the first version of the COA, and Sylvain Lecomte, PhD candidate at RD2P, for his work on the compiler. We are grateful to Pierre Paradinas from Gemplus and Vincent Cordonnier, director of RD2P, for support to our work and for helpful suggestions and comments on earlier and final drafts of this paper.

References

1. ADVANCED RISC MACHINES LTD (ARM). *ARM Software Development Toolkit Reference Manual and Programming Techniques*, version 2.0 ed. Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, United Kingdom, June 1995.
2. BRODIE, M. L. The promise of distributed computing and the challenges of legacy information systems. In *Proceedings of the IFIP WG2.6 Database Semantics Conference on Interoperable Database Systems (DS-5), Lorne, Victoria, Australia, 16-20 November, 1992* (North-Holland, 1993), Elsevier Science Publishers B.V., pp. 1–31.
3. CORDONNIER, V. Assessing the future of smart cards. In *CardTech/SecurTech 1992 conference proceedings, April 18 to 22, 1992, Arlington, Virginia, U.S.A.* (11619 Danville Drive, Rockville, MD 20852, U.S.A., Apr. 1992), CardTech/SecurTech, Inc., Ed.
4. GAMACHE, A., PARADINAS, P., AND VANDEWALLE, J.-J. Worlwide smart card services. In *Proceedings of CARDIS first Smart Card research and advanced application conference, Lille, France, October, 1994* (CHRU Calmette, Rue du prof. Jules Leclerc, 59037 Lille cedex, France, Oct. 1994), V. Cordonnier and J.-J. Quisquater, Eds., RD2P : Recherche et Développement Dossier Portable, pp. 141–148.
5. GENERAL MAGIC, INC. *The Telescript Language Reference*, version 1.0 ed. 420 North mary Ave., Sunnyvale, CA 94086 USA, Oct. 1995. <URL:http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html>.
6. GOSLING, J., AND MCGILTON, H. *The Java Language Environment: A White Paper.* Sun Microsystems, Inc., 2550 Garcia Ave., Mtn. View, CA 94043-1100 USA, 1995. <URL:http://java.sun.com/whitePaper/javawhitepaper_1.html>.
7. HARTEL, P. H., AND DE JONG FRZ, E. K. Towards testability in smart card operating system design. In *Proceedings of the first smart card research and advanced application conference, october 24–26, 1994, Lille, France* (CHRU Calmette, Rue du prof. Jules Leclerc,V. Cordonnier and J.-J. Quisquater, Eds., RD2P : Recherche et Développement Dossier Portable, pp. 73–88.
8. HONG, P. J. Threaded code designs for forth interpreters. *ACM SIGFORTH Newsletter 4*, 2 (Dec. 1992), 11–16.
9. IONA TECHNOLOGIES LTD. *Orbix : advanced programmer's guide*, 1.3 ed. 8-34 Percy Place, Dublin 4, Ireland, Apr. 1995.
10. ISO INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts – Part 3: electronic signals and transmission protocols.* P.O. Box 56, 1211 Geneva 20, Switzerland, Nov. 1994.

11. Khanna, R., Ed. *Distributed computing : Implementation and Management Strategies.* PTR Prentice Hall, 1994.
12. Masini, G., Napoli, A., Colnet, D., Léonard, D., and Tombre, K. *Les langages à objets.* InterEditions, 1991.
13. Merle, P., Gransart, C., and Geib, J.-M. CorbaWeb : A Generic Object Navigator. In *Proceedings of the Fifth International World Wide Web Conference, Paris, France, may 6–10, 1996* (May 1996).
14. Merle, P., Vandewalle, J.-J., and Dufresne, E. Intégration d'environnements hétérogènes : World Wide Web, Carte à microprocesseur et Corba. In *Actes du XIVieme congrès INFORSID, 4–7 juin 1996, Bordeaux, France* (20, rue Axel Buboul, 31100 Toulouse, France, June 1996), Association INFORSID, INFormatique des Organisations et Systèmes d'Information et de Décision, pp. 235–248.
15. Noyelle, Y. *Traitement des langages évolués – compilation, interprétation, support d'exécution.* Manuels Informatiques Masson. Masson, Paris, 1988.
16. OMG. The Common Object Request Broker : Architecture and Specification. OMG Document 93-12-43, Object Management Group, Headquaters, 492 Old Connecticut Path, Framingham, MA 01701, USA, Dec. 1993.
17. Paradinas, P., and Vandewalle, J.-J. New directions for integrated circuit card operating system. *Operating System Review 29*, 1 (Jan. 1995), 56–61.
18. Peltier, T. *La Carte Blanche : Un nouveau système d'exploitation pour objets nomades.* Ph.D. dissertation, Université des Sciences et Technologies de Lille, Cité Scientifique, 59655 Villeneuve D'Ascq cedex, France, Dec. 1995.
19. Peyret, P. Risc-based, next-generation smart card microcontroller chips. In *CardTech/SecurTech 1994 conference proceedings, April 10 to 13, 1994, Arlington, Virginia, U.S.A.* (11619 Danville Drive, Rockville, MD 20852, U.S.A., Apr 1994), CardTech/SecurTech, Inc., Ed., pp. 9–36.
20. Pfleeger, C. P. *Security in Computing.* PTR Prentice Hall, 1989.
21. Soley, R. M. Object Management Architecture Guide. OMG Document 94-11-1, Object Management Group, Headquaters, 492 Old Connecticut Path, Framingham, MA 01701, USA, Nov. 1994.
22. Soley, R. M. *Role of object technology in distributed systems.* In Khanna [11], 1994, pp. 469–478.
23. Sun Microsystems, Inc. *Frequently Asked Questions - Applet Security*, version 1.0.2 ed. 2550 Garcia Ave., Mtn. View, CA 94043-1100 USA, May 10, 1996. <URL:http://www.javasoft.com/sfaq/index.html>.
24. Vandewalle, J.-J. Loading several services into multi-purpose integrated circuit cards. In *Proceedings of European Research Seminar on Advances in Distributed Systems, ERSADS'95, L'Alpe d'Huez, France, April 3–7, 1995* (Apr. 1995), IMAG & INRIA, INRIA, pp. 317–322.
25. Venners, B. Under the Hood: The lean, mean, virtual machine - An introduction to the basic structure and functionality of the Java Virtual Machine. *JavaWorld 1* (June 1996). <URL:http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html>.
26. Wegner, P. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger 1*, 1 (Aug. 1990).
27. Yellin, F. Low Level Security in Java. In *Proceedings of the Fourth International World Wide Web Conference, Boston, MA, USA* (Dec. 1995), World Wide Web Consortium. <URL:http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html>.

Appendice: Card Objects and Java Applets

Mobile code technologies have seen their influence to grow with the popularity of languages such as Java [6] or Telescript [5]. These languages have been proposed for running downloaded code that comes from a remote host and trough an unsecure network. The remote code is executed locally into the client machine by an interpreter. To thwart malicious remote code these languages implement security mechanisms which are basically based on the object-oriented's encapsulation propertie.

Using object-oriented concepts and an interpreter to implement secure card objects is closely related to the Java Applets technology. But the similarity stops here : card objects have been developed independently from Java Applets since they were defined before any Java delevering.

To understand the exact relationships between these two technologies, the table below points out their similarities and their differencies. This table has been fulfilled with the help of [23, 25, 27] considering applet security mechanisms available in Netscape Navogator 2.0.

	Card Objects	**Java Applets**
General		
Originator	card service provider	Web server provider
Destination	generic smart cards	Web browsers
Purpose	non-predefined card services	(inter)activity into Web documents
Lifetime	until explicit drop by the service provider	during the visualization of the Web document and shortly after
Language		
High level language	similar to Forth	similar to C++
Interpretated language	threaded interpretative language type token-token	bytecode with operands, primitive types, and class files
Virtual machine	stack-oriented, objects are persitent in a DataSpace	stack-oriented, objects live in a garbage-collected heap
Security		
Access control	certificates	none
Secure execution	on-the-fly boundaries checking	static byte-code verifieur
Access to system ressources	input/output parameters via a DataStack	keyboard, audio, screen, mouse, and network with its server of origin
System access	restricted to primitives	restricted to applet base classes
Inter-communication	between card objects originated from the same service provider	between applets originated from the same server

This article was processed using the LaTeX macro package with LLNCS style