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.*

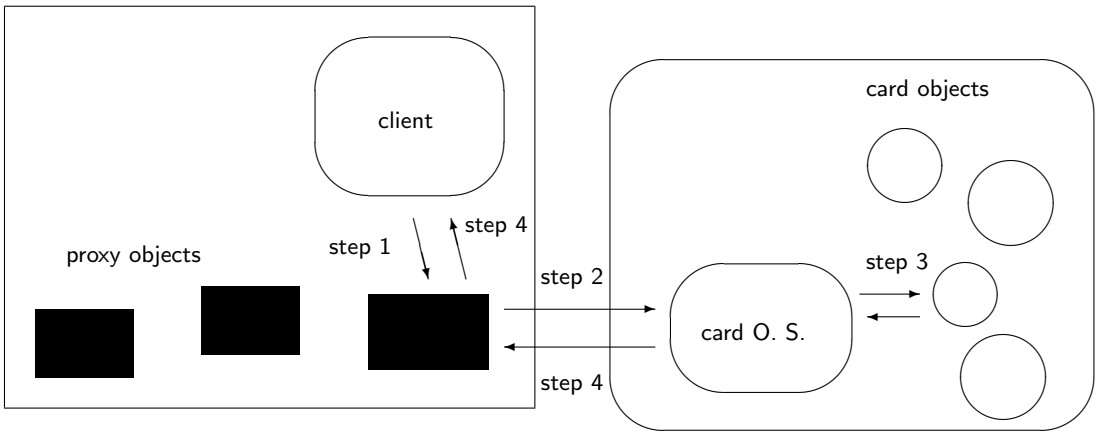

Fig. 1. Protocol 1: on-card computation

4. The card operating system returns the response to the client program.

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.*

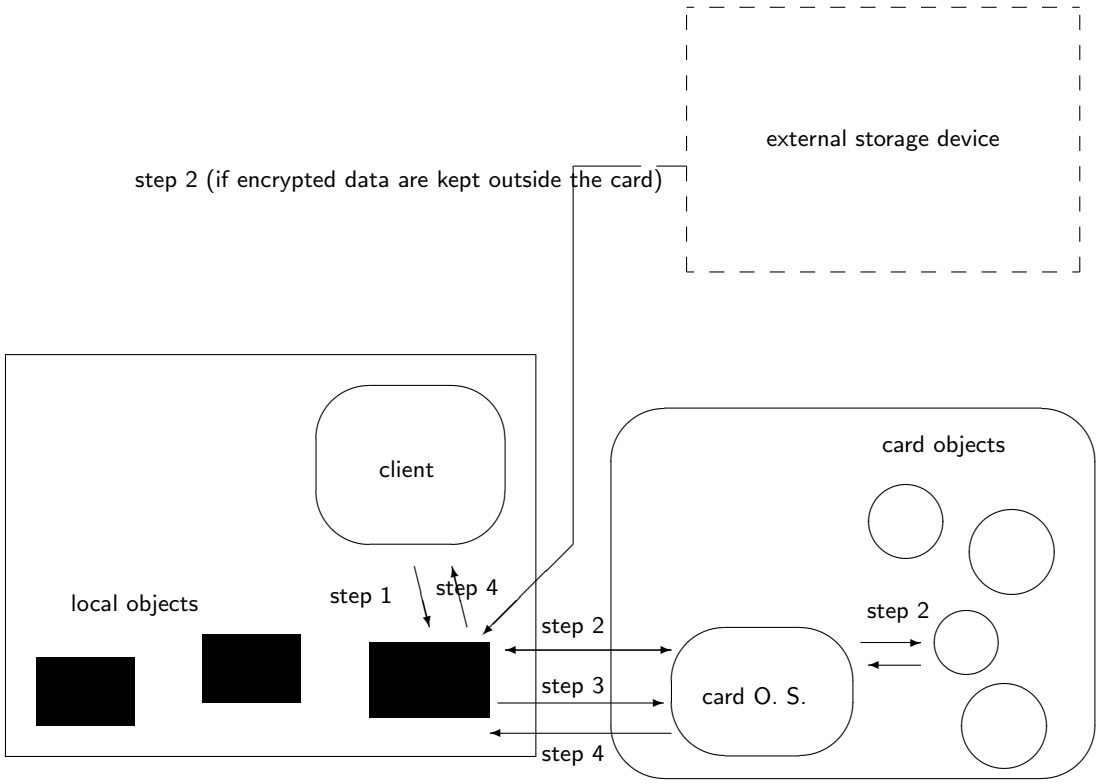

Fig. 2. Protocol 2: off-card computation

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 $r^{-1} \bmod m$ 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 $r^{-j} \bmod m$ to retrieve $a_{.j} \bmod m$. 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