

# Enhanced Smart-card based License Management \*

Mikhail J. Atallah    Jiangtao Li

CERIAS and Computer Science Dept., Purdue University, West Lafayette, IN 47907

## Abstract

*In many e-commerce situations, the owner of a digital object wants to enforce policies on the object after the object is in the customer's hands. The object can be thought of as being software, because data is often protected by forcing access to it to take place through a particular authorized software (e.g., a "reader" for an encrypted media file, in which case a license to view the movie is, in some sense, a "software license"). One of the ways that were proposed for such policy enforcement is the use of smart cards.*

*This paper describes an enhanced solution to software license management based on tamper-resistant smart cards. Our public-key protocols for binding software licenses to smart cards improve on previous schemes in that they support flexible and partial transfers of licenses between cards. The license is verified by checking the presence of the associated card. The user can therefore have several software licenses all of which are bound to one card, to avoid juggling several cards in and out of the card reader.*

## 1 Introduction

One of the main reasons media owners and distributors have not more widely embraced the Internet as a distribution medium, and still cling to inefficient and outmoded distribution mechanisms such as selling physical media in physical stores, is because they do not want to make pirates' task any easier than it already is (and it already is easy enough). To make e-commerce more widespread in such digital objects, it would help if the owner could still enforce policies after the digital object has been delivered to the customer. This is tied closely to the issue of software licensing, because whatever one can do for software licenses also applies for movie-viewing licenses, or other remote policy-enforcement mechanisms. This is because access to media can be forced to occur through a particular piece of "autho-

rized" viewing software<sup>1</sup>. So whenever, in what follow, this paper talks about "software license", one could just as well view that as a statement about "movie license" or indeed any kind of other media.

Aura and Gollmann recently proposed an elegant software license management scheme with tamper-resistant smart cards [2]. In their scheme, software licenses are bound to smart cards, and software cannot be run without the appropriate card being in the card reader. They designed protocols for transferring licenses between cards, so that customers only need one card to use all of their software. Their scheme is fine in a home environment, but it is not scalable and flexible enough for large company based license management. One drawback of their scheme is that once a smart card transfers its licenses to another card, the original card is destroyed ("zero'ed out") and licenses cannot be transferred back. Another drawback is that their scheme does not support partial transfers, that is, if card A wants to transfer a license to card B but not the other licenses in it (or, if it contains a single kind of license but with a multiplicity of  $k$ , it needs to stay alive for the remaining  $k - 1$  licenses remaining in it).

In this paper, we build on the work of Aura and Gollmann and extend it to eliminate the above mentioned drawbacks. In our model, each smart card can contain one or many types of software licenses, each with a different multiplicity. A software verifies its license by checking the presence of the smart card in the card reader. We design protocols such that a software license can be flexibly transferred between any two cards. Since smart cards have a limited amount of storage, it is not acceptable to explicitly store in the card one entry for each software license bound to it. We next describe three schemes for managing license information with smart cards of  $O(1)$  storage space, i.e., not dependent on the number  $n$  of software licenses bound to the card.

Our paper is indirectly related to one-way accumula-

---

\* Portions of this work were supported by Grants EIA-9903545 and ISS-0219560 from the National Science Foundation, Contract N00014-02-1-0364 from the Office of Naval Research, and by sponsors of the Center for Education and Research in Information Assurance and Security.

---

<sup>1</sup>Taking this idea to its logical extreme, any media (such as a large movie) can be encrypted and combined with its custom-decrypting reader as a single self-extracting unit (i.e., as software); the reason it may not be too inefficient to include a special reader with every movie (a reader for viewing only that particular movie) is because a movie is hundreds of megabytes long whereas a reader is typically smaller than 1% of that.

tors [6], memory checking [7, 13], incremental cryptography [4, 5], trusted database [18], and certificate revocation [15, 21]. Our scheme uses a judicious combination of techniques from these areas.

The rest of the paper is organized as follows. We begin with a short introduction to copy protection with tamper-resistant smart cards in Section 2. Then Section 3 introduces our enhanced model for software license management and Section 4 gives the protocol details. We propose three schemes for managing license keys within smart cards in Section 5. Section 6 concludes the paper.

## 2 Smart-card based copy protection

In token based software protection mechanisms, a license is embodied by a copy-resistant piece of hardware [8, 19]. The software checks for the presence of such a token, and refuses to run unless the token is present. As smart cards become popular and cheap, a smart card becomes a natural choice for a token.

Unfortunately, the token based protection mechanisms are unpopular with the users. As was pointed out at [2], a major problem for smart-card based protection scheme is that a single card must not be allowed to monopolize the card reader. In order to prove the presence of a card for different software packages, one may have to repeatedly insert different cards into the reader, an annoying practice sometimes referred as *smart card juggling*.

Aura and Gollmann describe a solution for binding software licenses to smart cards and for transferring them from card to card in such a way that juggling is eliminated [2]. Their transfer protocols are based on delegation. The smart card having some licenses simply signs a certificate stating its willingness to give its rights to the key of another card. This kind of certificate with which one key delegates access rights to another one is called a *delegation certificate* [3, 12]. As we can notice from their scheme, if one card transfers licenses to another, all the licenses are transferred to the new card and the old card is destroyed, so it is an “all or nothing” process.

## 3 License management model

In this section, we describe the model we use for software license management. In order to have flexibility and ease of use, our goal is to allow a single smart card to act as a token for arbitrarily many software packages, and to allow licenses to be readily transferred between smart cards. The reader already familiar with previous work in this area can skip the discussion subsection and go directly to Section 3.2. As commonly assumed in this literature, we assume (i) that the smart card is tamper-resistant in the sense

that its stored data can be protected against unauthorized access and modification, and (ii) that the license checking procedure cannot be bypassed by an adversary who is attempting to “crack” the software (preventing such cracking is the subject of much investigation in a different kind of literature c.f., [10, 11, 25]).

### 3.1 Informal discussion

A smart card in our model is used for two purposes: managing license information and run-time license verification. Smart cards are distributed along with the software package at purchase time.

Each smart card has a unique public-private key pair. The private key is known only to the card and never revealed outside. The public-private key pair is used for proving to the software the presence of the card in the reader. To prove a card has the private key responding to the public key, the card encrypts a challenge issued by the software (usually a nonce) with its private key and sends it back. After the software decrypts the message with the card’s public key and verifies the nonce, the software believes the card is authentic.

Now we introduce how to bind a software license to a smart card. In Aura and Gollmann’s scheme, they use a license certificate to tie the software license to a card key. However, this is not suitable for our model, because our goal is to let licenses readily “flow” to any cards, whereas the software publisher initially has no way of knowing the target card key. In our model each license has a unique public-private key pair and a license certificate. The license’s public key may be obtained by everyone, whereas the license’s private key is only known to one card. Indeed, initially a license’s private key is encrypted by a card’s public key, so that only that card knows it. The license certificate is signed by the software publisher’s master key to tie the license’s public key to a particular software package.

Each card keeps a list of license keys (referred to later as *license key list*) that are currently bound to it. Although this appears to be contradict the  $O(1)$  space limitation of the smart card, in fact it does not because of the list will be maintained implicitly (more on this later). A license is bound to a card by two conditions. First, the license’s private key is encrypted by the card key so that only the card can decrypt it. Second, the license’s public key belongs to the card’s license key list. In other words, the software believes a card has a license if the card can prove that it knows the license’ private key. To avoid revealing the license’s private key, the card proves to the software its knowledge of license’s private key in same manner as card’s own private key: it responds to a challenge from the software by encrypting the challenge with the license’s private key.

### 3.2 Formal description of the model

In this subsection, we give a formal description of our model, and explain how our model can be implemented in practice. We use the notations from [9] to describe our model and protocols.

The notations used in this paper are listed in Table 1. In our model,  $(K_C, K_C^{-1})$  and  $(K_L, K_L^{-1})$  are only used for encryption and decryption, whereas  $(K_{SP}, K_{SP}^{-1})$ ,  $(K_{CP}, K_{CP}^{-1})$  and  $(K_{CA}, K_{CA}^{-1})$  are only used for verification and signature. We assume there is a widely trusted  $CA$  who is responsible for giving certificates to software publishers and card producers.

Principal Name	Notation	Key Pair
Workstation	$W$	
Smart Card	$C$	$(K_C, K_C^{-1})$
Software License	$L$	$(K_L, K_L^{-1})$
Software Publisher	$SP$	$(K_{SP}, K_{SP}^{-1})$
Card Producer	$CP$	$(K_{CP}, K_{CP}^{-1})$
Certificate Authority	$CA$	$(K_{CA}, K_{CA}^{-1})$

**Table 1. List of notations**

Each software publisher has a certificate from the certificate authority  $CA$  of the form  $\{K_{SP}\}_{K_{CA}^{-1}}$ . The notation  $\{M\}_K$  means that the message  $M$  is encrypted under the public key  $K$  and  $\{M\}_{K^{-1}}$  means that the message  $M$  is encrypted under the private key  $K^{-1}$ . (In the case of certificates,  $\{M\}_{K^{-1}}$  means the message  $M$  is signed with  $K^{-1}$ .) Note that we intentionally omit other miscellaneous information (such as software publisher name, address, ID, etc) in the certificate. The implementations should follow accepted standards such as PKCS [16]. Similarly, each card producer has a certificate from  $CA$  of the form  $\{K_{CP}\}_{K_{CA}^{-1}}$ .

In our model, the smart cards are issued by  $CP$ , whereas the software licenses are issued by  $SP$ . We assume the software or the operating system of the workstation has already embedded  $K_{CA}$  in the code so that the software can verify the certificates of  $K_{CP}$  and  $K_{SP}$ , and conclude that they are authentic card producer's key and software publisher's key. To simplify our explanation in the rest of our paper, we also assume the software has embedded  $K_{CP}$  and  $K_{SP}$  in its code. (Without such assumption, we need only one more step of certificate verification).

In addition to the keys, each card stores a *card certificate* signed by its card producer  $CP$ . Anyone knowing  $K_{CP}$  can verify the certificate on the card and conclude that the card is an authentic card approved by  $CP$ . A card certificate is of the form

$$\{K_C, \text{"is a valid card key."}\}_{K_{CP}^{-1}}.$$

Each software license has a *license certificate* signed by

software publisher  $SP$ . Anyone knowing the  $K_{SP}$  can verify the license certificate and conclude that the license key is an authentic license key approved by  $SP$ . A license certificate is of the form

$$\{K_L, \text{"is a valid license key for", software}\}_{K_{SP}^{-1}}.$$

Beside the card key and the card certificate, each smart card also maintains a license key list. This list contains all the license keys that are bound to that card. It is not necessary to keep the entire license key list inside the card; we can store the list outside the card and only keep the integrity check value of the list in the card. We will give solutions to this problem in Section 5.

As in [2], the software license information is stored outside the smart card (i.e. user equipment). All the licenses bound to a smart card are stored together in a table. Each table entry has a license number, a license key pair, and a license certificate. The license key pair is encrypted by the card key. We also put the license's public key in the clear text so that it can be read by anyone. Usually if the card does not store the license keys inside the card, in the license information table, each entry also contains a proof that the license key in that entry is a valid member of the card's license key list. Table 2 shows an example structure of license information table with three licenses for software  $A$  and one license for software  $B$ .

License ID	License Key	Certificate
License 1	$K_{L_1}, \{K_{L_1}, K_{L_1}^{-1}\}_{K_C}$	$\{K_{L_1}, A\}_{K_{SP}^{-1}}$
License 2	$K_{L_2}, \{K_{L_2}, K_{L_2}^{-1}\}_{K_C}$	$\{K_{L_2}, A\}_{K_{SP}^{-1}}$
License 3	$K_{L_3}, \{K_{L_3}, K_{L_3}^{-1}\}_{K_C}$	$\{K_{L_3}, A\}_{K_{SP}^{-1}}$
License 4	$K_{L_4}, \{K_{L_4}, K_{L_4}^{-1}\}_{K_C}$	$\{K_{L_4}, B\}_{K_{SP}^{-1}}$

**Table 2. Example of license information table**

## 4 License verification and transfer protocols

This section describes protocols for license verification and license transfer. Some crucial issues and details will be given in later sections. We begin with a brief summary of the characteristics of our protocols.

### 4.1 Characteristics of our Protocols

The characteristics summarized below include a description of the communication patterns for the protocols (who needs to participate, and who talks to whom), as well a "before" and "after" snapshot of what each participant gets (or learns). In what follows, the card that contains the licenses is the same one as the card that establishes at run-time the

existence of the licenses (i.e., there are not two separate kinds of cards).

- *Software verifies license from smart card:* The participants are the software and the smart card. The software publisher is not a party to the license verification protocol. Before the protocol begins it is assumed that the software has the public keys of  $CP$  and  $SP$ . After the protocol completes, the software believes that the license is bound to the card. The software has not learned anything else about the smart card (e.g., the license's private key, the card's private key).
- *License card owner purchases online additional licenses from software publisher:* The participants are the smart card and the software publisher. Before the protocol begins it is assumed that the smart card has certificate from  $CP$  whose public key is known to software publisher and the software publisher has a certificate from  $CA$  whose public key is known to the card. After the protocol completes the smart card has (implicitly) the kind and quantity of licenses purchased, and the software publisher has not learned anything else about the smart card (e.g., what other licenses were in it, from other software vendors or from itself). *Note:* Because of space limitations, we have not included in the paper the online purchase protocol; it is easily inferable from the protocol for transferring licenses between two distinct cards.
- *Transfer of licenses between two cards:* The participants are the two smart cards, i.e., the software publisher is not a party to the transfer protocol. Before the protocol begins it is assumed that each card has a certificate from a  $CP$  whose public key is known to the other card (it may be a different  $CP$  for each card). The net effect of the transfer protocol is the effective migration of a license between the two cards (no replication of licenses occurs). Neither card learns anything about the other smart card (e.g., what other licenses were in it).

## 4.2 Protocol for license verification

The verification consists of two certificate verifications and one interaction with the smart card. The two certificate verifications are done within the workstation. We assume the software already has the public keys of  $SP$  and  $CP$ . The software first picks up the corresponding license entry from the license information table (it is possible to have several valid licenses for that software, and in that case, we just randomly pick up one). Then the software verifies the license certificate issued by  $SP$  in that license table entry. Since the license is associated with a card key  $K_C$ , the software proceeds to verify the card certificate signed by  $CP$ .

After the software succeeds in checking the two certificates, it interacts with the smart card for two purposes: verifying that the card can decrypt the license's private key and verifying that the card is authentic. Protocol 1 shows how the software verifies licenses in our model.

### Protocol 1 (license verification)

1. Workstation  $W \rightarrow$  Card  $C$ :  
 $\{K_L, K_L^{-1}\}_{K_C}, N_W$
2. Card  $C \rightarrow$  Workstation  $W$ :  
 $\{N_W\}_{K_L^{-1}}, \{N_W\}_{K_C^{-1}}$  if  $K_L \in$  License Key List

In the first step of Protocol 1, the workstation sends the card the encrypted license key pair and a nonce identifier. The card first decrypts the license's key pair  $(K_L, K_L^{-1})$ , then verifies whether  $K_L$  is in its license key list. (We will describe how the card verifies the membership of a license key in Section 5.) If  $K_L$  does not belong to the license key list, the card refuses to respond. Otherwise, the card sends the nonce identifier back encrypted with  $K_L^{-1}$  and  $K_C^{-1}$  respectively. Finally The software decrypts the response message and compares with the original nonce identifier. When the software sees the correct response, it believes that the card has the corresponding license.

## 4.3 Protocol for license transfer (preliminary version)

To avoid unnecessarily cluttering the exposition, we start by ignoring (for the time being) issues of accidental interruption of the transfer protocol. If the workstation has two card readers, then the two cards can directly communicate with each other. If only one card reader exists, the user might need to swap the cards in the reader several times to finish the transfer protocol. Two cards can even transfer licenses from different machines via network communication.

Suppose Card  $A$  wants to transfer a license to Card  $B$ , Card  $A$  first verifies that the license is bound to itself, i.e., verifies it can decrypt the license's private key inside the card and the license's public key belongs to Card  $A$ 's license key list. Then Card  $A$  deletes the license key from its list and encrypts the license's key pair with Card  $B$ 's public key. Finally Card  $B$  adds the license key into its license key list. (We will describe how to update the license key list in Section 5.) Figure 1 shows an example of license transfer between two cards with only one license exists in the system. The upper half of the figure shows the binding structure before the transfer, the lower half of the figure is the binding structure after the transfer. Notice that after Card  $A$  gives the license to Card  $B$ , the license is no longer bound to Card  $A$  any more.

Protocol 2 is the protocol for transferring a license between two cards. In the protocol,  $A$  is the source card,  $B$  is

## Protocol 2 (license transfer between cards)

1. Card  $A \rightarrow$  Card  $B$ :  
 $K_A, \{K_A\}_{K_{CP}^{-1}}$
2. Card  $B \rightarrow$  Card  $A$ :  
 $K_B, \{K_B\}_{K_{CP}^{-1}}$
3. Card  $A \rightarrow$  Card  $B$ :  
 $\{N_A, K_A\}_{K_B}$
4. Card  $B \rightarrow$  Card  $A$ :  
 $\{K_B, N_A, N_B, \text{"Please transfer } K_L \text{ to me"}\}_{K_A}$
5. Card  $A$ :  
 Decrypt  $K_L, K_L^{-1}$   
 Delete  $K_L$  from  $C_A$ 's License Key List
6. Card  $A \rightarrow$  Card  $B$ :  
 $\{N_B, \text{"Please add } K_L \text{ to License Key List"}\}_{K_B}$   
 $\{K_L, K_L^{-1}\}_{K_B}$
7. Card  $B$ :  
 Add  $K_L$  into  $C_B$ 's License Key List

the destination card.  $K_A$  and  $K_B$  are the card keys for  $A$  and  $B$  respectively;  $N_A$  and  $N_B$  are their nonce identifiers.

The first four steps of the protocol are used to exchange secrets (nonces) between two cards and set up a secure communication channel against eavesdropping or tampering. We use the approach from a fixed version of the Needham and Schroeder's public-key protocol [22] proposed by Lowe [17]. In the first two steps of the protocol, Card  $A$  and Card  $B$  obtain each other's public keys and card certificates. Then in Step 3 and 4, Card  $A$  and Card  $B$  exchange their nonce identifiers for further communication. Later, if Card  $B$  receives a message  $\{N_B, M\}_{K_B}$ , then card  $B$  believes the freshness of  $M$ , and may deduce that Card  $A$  sent  $M$ .

Step 5 to 7 of the protocol are to transfer a software license after the secure channel is set up. In Step 5, Card  $A$  decrypts the license key pair inside the card and verifies that the 'to-be-transferred' license is currently bound to itself. Then Card  $A$  deletes  $K_L$  from its license key list. Card  $A$  sends the encrypted license key pair to Card  $B$  in Step 6 and informs Card  $B$  to continue. Finally Card  $B$  adds  $K_L$  into its license key list in Step 7.

### 4.4 Extention of the transfer protocol

In the last subsection, we show how to transfer software licenses via a secure communication channel. In practice, if we plan to build a real license management system over our model, it is crucial to make the transfer protocols atomic.

One problem is about the interruptions of the transfer protocols. If during the transfer, the power is accidentally cut off by a honest customer, or the card is intentionally pulled out from the reader by a malicious user, the transfer protocol should either complete or return to the original

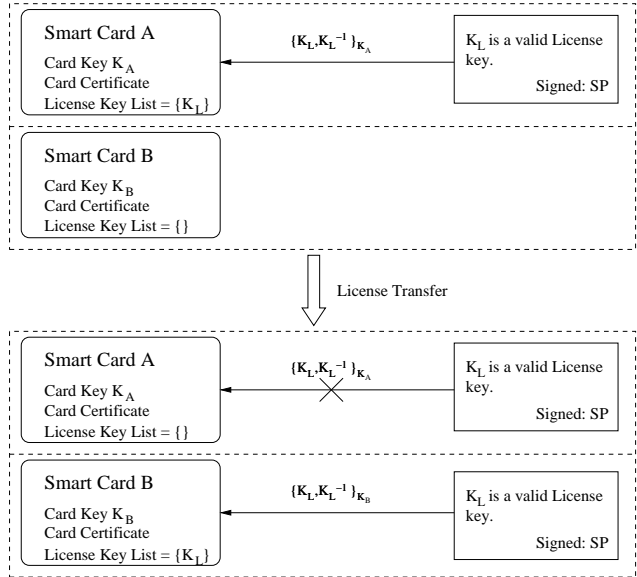


Figure 1. Example of license transfer

state before the transfer procedure. In either case, we want to eliminate the possibility of license loss or license duplication that happened during the license's transfer.

To solve this problem, we need to make our transfer protocols atomic. We can use some well known protocols such as three-phase commit protocol [24] to ensure the atomic property. Indeed if we look at the license transfer as a transaction, the transfer protocol in our model essentially consists of two operations: (i) deleting from the source card's license key list and (ii) inserting into the destination card's license key list. We can extend the protocol by updating the license key list in the cards' RAM, and not writing into the permanent storage of the cards until the two cards commit the transfer transaction. If the license transfer protocol is interrupted, it will fail to commit, thus both cards will return to the original states.

*Note:* Fair exchange protocols do address the problem of exchange of digital objects between two parties that do not trust each other, and address the issues of atomicity, simultaneity, nonrepudiation. They do so using elegant techniques and ideas, but at a rather large cost, at least in terms of the limited space and speed of a smartcard (not to mention the occasional requirement of a third party – trusted or untrusted – that does not fit within our framework).

## 5 Managing license keys within the smart card

Whereas the previous section focussed on protocols, this section deals with the details of how a smart card managing license information.

## 5.1 Introduction

As we already described early, the key issue for the license transfer is to maintain a list of license keys that are bound to the smart card. Given that a smart card has  $O(1)$  storage capacity, we cannot store the entire license key list inside the card. In this section, we describe three schemes to store the license key list in untrusted user equipment in a way that it cannot be tampered by an adversary.

### 5.1.1 Problem definition

The problem of managing license keys involves two parties: a trusted smart card and an untrusted workstation. In our model, we assume the tamper-resistant smart card is a trusted entity. The card is pre-programmed and would not malfunction. The software publisher trusts the behavior of the smart card, but it does not trust the workstation. The smart card has a list of license keys that evolves over time through insertions and deletions of keys. The workstation maintains a data structure to store the list, whereas the smart card keeps the integrity check value of the list. The smart card does not need to know the complete list. Instead, the card typically performs membership queries on the list of the type “is license key  $k$  in my license key list?”. The workstation provides the card with a YES/NO answer to the query. If the answer is YES, the workstation also needs to provide an evidence so that the card can verify it. The integrity check value in the card is used to verify the answer from the workstation.

More formally, let  $W$  be the workstation and  $C$  be the smart card. Let  $L = \{k_1, k_2, \dots, k_n\}$  be the license key list with  $n$  license keys. Let  $D_L$  be a data structure representing  $L$  in  $W$  and  $I_L$  be the integrity check value of the data structure  $D_L$  in  $C$ . We have following four operations:

- A *membership query* is issued by  $C$  of the form  $\langle k \rangle$ .  $W$  responds the query with answer  $\langle a, e \rangle$  where  $a \in \{\text{YES}, \text{NO}\}$  and  $e$  is an evidence to the answer.
- A *validation* operation is associated with a membership query. Given an answer  $\langle \text{YES}, e \rangle$ ,  $C$  verifies the evidence  $e$ .
- *Insertion* operation is to insert  $k$  into  $L$ , where  $k \notin L$ .  $W$  updates its data structure  $D_{L'}$  to represent the new list  $L' = L \cup \{k\}$ , and  $C$  updates the corresponding integrity check value  $I_{L'}$ .
- *Deletion* operation is to delete  $k$  from  $L$ , where  $k \in L$ .  $W$  updates its data structure  $D_{L'}$  to represent the new list  $L' = L \setminus \{k\}$ , and  $C$  updates the corresponding integrity check value  $I_{L'}$ .

Note that  $C$  only verifies the answer to the query when the answer is YES. Our goal is to minimize the computational cost for the smart card.

One may worry about malicious users add unauthorized licenses into the license key list by making use of the insertion operation. In fact, the insertion operation in our model is a subroutine that cannot be called directly from outside. The insertion operation is executed only during the transfer protocol. As we described in the previous section, the atomic property of the transfer protocol guarantee no illegal insertion of licenses possible.

### 5.1.2 Related work

The problem of storing and retrieving data from unreliable workstations and keeping its integrity check value in the smart card is related to memory checking [7, 13], incremental cryptography [4, 5], trusted database [18], certificate revocation [21], and authentication dictionaries [15].

Blum et al. [7] extended the notion of program checking to include programs which alter their environment. In their model, checker resides in a small amount of reliable memory, whereas the data structure resides in a large but unreliable memory. The checker is used for detecting errors in the data structure. They construct an online checker for RAMs using a variant of Merkle’s hash-tree authentication scheme for digital signatures [20].

Incremental cryptography was first introduced by Bellare, Goldreich and Goldwasser [4, 5]. The goal of incremental schemes is to quickly update the value of a cryptographic primitive when the underlying data is modified. They propose an incremental authentication scheme based on a 2-3 search tree in order to allow efficient insert/delete/replace block operations.

Maheshwari, Vingralek and Shapiro [18] built a trusted database system on untrusted storage by making use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

In the certificate revocation problem, directory is an entity that get updated certificate revocation information from the certificate authority and serve as a certificate database accessible by the users. An authenticated dictionary, defined in [21], is a data structure used by the directory to efficiently maintain a set of elements. Naor and Nissim [21] construct to an authenticated dictionary by a 2-3 hash search tree to support insertion and deletion of elements. Goodrich, Schwerin and Tamassia [15] develop a data structure for authenticated directories based on one-way accumulators [6]. Their underlying idea is to dynamically maintain a one-way accumulator function over the set elements.

Our model differs from these related existing frameworks in that, in addition to the only  $O(1)$  storage per smart card, there is also an asymmetry in computing power (slow

smart card, fast workstation) that imposes the necessity for protocols to place most of the computational burden on the workstation rather than on the smart card.

## 5.2 One-way accumulator scheme

A family of *one-way accumulators*, as defined by Benaloh and Mare [6], is a family of one-way hash functions each of which is quasi-commutative. A well-known example of a one-way accumulator function is the *exponential accumulator*,

$$\text{exp}(x, y) = x^y \bmod N,$$

for suitably-chosen values of the generator  $x$  and modulus  $N$  [6]. As denoted in [6], a prime  $p$  is called to be *safe* if  $p = 2p' + 1$  where  $p'$  is an odd prime. In particular, Benaloh and Mare choose  $N = pq$  where  $p$  and  $q$  are distinct safe primes such that  $|p| = |q|$ . As we may notice, almost any odd number is relative prime to  $\phi(N)$ . To see why, let  $N = pq$ ,  $p = 2p' + 1$ ,  $q = 2q' + 1$ , such that  $p, q, p'$ , and  $q'$  are all large distinct primes. Thus,  $\phi(N) = (p-1)(q-1) = 4p'q'$ . Any odd number is relative prime to  $\phi(N)$  if it is relative prime to  $p'q'$ . Numerically, a 200 bit  $N$  would result in around 100 bit size in  $p'$  and  $q'$ , the probability of a randomly chosen odd number that is not relative prime to  $\phi(N)$  is well below  $10^{-30}$ . Therefore, we neglect this probability in the rest of this paper.

Note that if  $y$  is relatively prime to  $\phi(N)$ , by Euler and Fermat theorems (see [23]) there exists  $y$ 's multiplicative inverse  $y^{-1}$  modulo  $\phi(N)$ , such that  $(x^y)^{y^{-1}} \equiv x \bmod N$  for any  $x$  relative prime to  $N$ .

Let  $L = \{k_1, k_2, \dots, k_n\}$  be the list of license keys the smart card need to manage. The card has two safe primes  $p$  and  $q$  that are suitably large, and a suitably-large generator  $x$  that is relatively prime to  $N$ . ( $p, q$ , and  $x$  are already chosen by the card producer during production time.) The values of  $x$  and  $N$  can be obtained by the workstation, but  $p$  and  $q$  are kept secretly in the card.

For each license key  $k_i$ , we calculate  $y_i = 2k_i + 1$  which is an odd integer. Assume that every  $y_i$  is relatively prime to  $\phi(N)$ . The smart card computes the accumulated hash  $z = x^{y_1 y_2 \dots y_n} \bmod N$  and keeps this value inside the card as the integrity check value to the list. The workstation for each element  $k_i$  from the list computes  $z_i = x^{y_1 \dots y_{i-1} y_{i+1} \dots y_n} \bmod N$  which represents the accumulated hash of all the  $y_j$  with  $j \neq i$ . The workstation keeps the  $(k_i, z_i)$  pairs as the data structure. At the initial state, the license key list is empty. Therefore, initially the data structure in the workstation is null, whereas the integrity check value in the smart card is the generator  $x$ .

**Query:** The smart card sends a membership query to the workstation with format  $\langle k_i \rangle$ . The workstation will look up its data structure to find whether  $k_i$  exists. If  $k_i$  does not

exist in the list, the workstation simply replies NO. If  $k_i$  exists in the list, the workstation replies  $\langle \text{YES}, z_i \rangle$ . Note that if  $k_i$  is a member of the list, there is no reason for the workstation to reply NO, which will eventually result in the loss of a software license.

**Validation:** To verify the answer of a query  $\langle \text{YES}, z_i \rangle$ , the smart card first computes  $y_i = 2k_i + 1$ , then computes  $z_i^{y_i} \bmod N$  and compares it to  $z$ . If  $z = z_i^{y_i} \bmod N$ , then the smart card is reassured of the validity of the answer. Indeed, it is generally accepted to be computational infeasible for someone who does not know the values of  $p$  and  $q$  to compute a value  $w$  such that  $z = w^{2k_i+1} \bmod N$  when  $k_i \notin L$ . Therefore, it is computational infeasible for an adversary to fake a particular membership.

**Insertion:** To insert a license key  $k_{n+1}$  into the list  $L$ , both the data structure  $D$  and the integrity check value  $I$  need to be updated. The card updates the integrity check value of the new list  $L' = L \cup \{k_{n+1}\}$  by setting the accumulated hash  $z' = z^{y_{n+1}} \bmod N$ , where  $y_{n+1} = 2k_{n+1} + 1$ . The workstation updates the data structure by computing new  $z'_i = z_i^{y_{n+1}} \bmod N$  for each element. The workstation also adds  $(k_{n+1}, z_{n+1} = x^{y_1 y_2 \dots y_n} \bmod N)$  into the data structure.

**Deletion:** To delete a license key  $k_j$  from the list  $L$ , both the workstation and the smart card first verify  $k_j \in L$ , then update the data structure  $D$  and its integrity check value  $I$ . The list will become  $L' = L \setminus \{k_j\}$ . The smart card computes  $y_j = 2k_j + 1$  and  $y_j^{-1}$  as multiplicative inverse modulo  $\phi(N)$  (recall that each  $y_i$  is relative prime to  $\phi(N)$ ). The smart card then updates the integrity check value by setting the accumulated hash  $z' = z^{y_j^{-1}} \bmod N$ . Indeed,  $z'$  is equal to the accumulated hash of all  $y_i$  with  $i \neq j$ , which is exactly the accumulated hash of new list  $L'$ . The workstation first deletes the entry of license key  $k_j$ , then computes  $z_i$  for all  $i \neq j$  from scratch. Note that the smart card cannot reveal  $y_j^{-1}$  to the workstation, otherwise, the workstation could factor  $N$  by the knowledge of  $y_j$  and  $y_j^{-1}$ . The performance of this scheme using the exponential accumulator is summarized in Table 3.

	Smart Card	Workstation
space	$O(1)$	$O(n)$
insertion time	$O(1)$	$O(n)$
deletion time	$O(1)$	$O(n^2)$
query/verify time	$O(1)$	$O(1)$

Table 3. One-way accumulator scheme

## 5.3 Balanced tree scheme

In the one-way accumulator scheme, there is a need to perform exponentiations for validation and updating a li-

cense key in the list. As computing modular exponentiation is expensive, we avoid it by using only cryptographic hash functions, in a kind of balanced tree scheme that has been used by many others before (e.g., [7, 4, 5, 18, 21], to mention a few). Because this is a standard technique, we only sketch it briefly.

Support each card has a pseudorandom function [14]  $f_S$  with seed  $S$  only known to itself. We maintain a 2-3 tree with leaves corresponding to the license keys in the list (details of 2-3 trees can be found in [1]). Recall that a 2-3 tree has all leaves at the same level/height and each internal node has either 2 or 3 children. And a 2-3 tree is an ordered tree, thus its leaves are in order.

Let  $L = \{k_1, k_2, \dots, k_n\}$  be the sorted list of license keys the card need to manage. For any  $i < j$ , we have  $k_i < k_j$ . We build the balanced tree in a bottom-up fashion:

- For each leaf, let the value of the  $i^{th}$  leaf be a pair  $(f_S(k_i), \text{size})$ , where  $\text{size}$  is 1.
- For each non-leaf node  $w$ , let the value of node  $w$  be a pair  $(f_S(v_1, v_2, v_3), \text{size})$ , where  $v_i$  is the value of the  $i^{th}$  child of  $w$  (in case  $w$  has only two children,  $v_3 = 0$ ) and  $\text{size}$  is the number of leaves in the subtree rooted at  $w$ .

The root value of the balanced tree is kept in the card. The  $\text{size}$  value is used to prevent an adversary from inserting a subtree instead of a license key.

To query the membership of license key  $k$ , the smart card sends a membership query with value  $k$  to the workstation. If  $k$  belongs to the list, the workstation sends all the nodes on the path from the corresponding leaf to the root and their children to the card. The card verifies that each nodes in the path is equal to the hash value of its children. The card also checks that the subtree sizes of the children sum-up to be subtree size of their parent. Due to memory limitation of the card, the card starts from the leaf up to the root verifying one node at a time. To insert a license key into the list, the card updates the values of the nodes on the insertion path. To delete a license key from the list, the card updates the values of the nodes on the deletion path. Again, the card updates these values one by one from the leaf upto the root. In order to prevent the workstation sending fault values, the card verifies the path before updates the root value. The performance of this balanced tree scheme is summarized in Table 4.

#### 5.4 License ticket scheme

In this subsection, we propose a simple and efficient scheme, given the assumption that the license verification happens much frequently than license transfer. This scheme has  $O(1)$  validation and insertion time and  $O(n)$  deletion

	Smart Card	Workstation
space	$O(1)$	$O(n)$
insertion time	$O(\log n)$	$O(\log n)$
deletion time	$O(\log n)$	$O(\log n)$
query/verify time	$O(\log n)$	$O(\log n)$

Table 4. Balanced tree scheme

time. Due to space limitation, we sketch this scheme only briefly, the details will be given in the full version of the paper.

Suppose each card has a pseudorandom function  $f_S$  with seed  $S$  only known to itself. The card can give an authenticated tag  $f_S(k)$  for each license key  $k$  in the list. Later on, a license key can prove its membership by showing the authenticated tag. Unfortunately, it cannot prevent replay attack (i.e. using an obsolete tag).

This scheme is inspired by the authenticated tag, the card gives each license key  $k$  a *license ticket* if  $k$  belongs to the list. The license ticket is of the form  $f_S(k, \text{sn})$  where  $\text{sn}$  is a serial number. The card assigns each license ticket an identical serial number. The integrity check value of the license key list will be those valid serial numbers. Given  $O(1)$  storage capacity of a card, the card cannot afford to store each serial number within its storage. Instead the card only store a boundary in the card. We manage these serial numbers in a way that all serial numbers between the boundary are valid and any serial number small than the lower bound is obsoleted.

Initially the list is empty, the boundary in the card is also null. Suppose in a given time, there are  $n$  licenses in the list, the boundary is  $[a, a + n - 1]$ , for some  $a > 0$ . Verifying the membership of a license key is easy: simply check whether its ticket's serial number is between the boundary. To insert a new license key  $k$  into the list, the card issues a license ticket for  $k$  with serial number  $a + n$  and set the new boundary to  $[a, a + n]$ . To delete a license key  $k$  with ticket serial number  $b \in [a, a + n - 1]$ , the card set its new boundary as  $[b + 1, b + n - 1]$ . The card effectively expires  $k$ 's ticket, since  $b$  is not in the boundary any more. On the other hand, it also expires some valid license tickets, in this example, all tickets between  $[a, b - 1]$  are expired. We can solve it by replacing each old ticket with a new ticket, i.e., replacing tickets of serial number between  $[a, b - 1]$  into tickets of serial number between  $[a + n, b + n - 1]$ . The performance of this license ticket scheme is summarized in Table 5.

## 6 Conclusion

We gave an enhanced solution to software license management based on tamper-resistant smart cards. We pre-

	Smart Card	Workstation
space	$O(1)$	$O(n)$
insertion time	$O(1)$	$O(1)$
deletion time	$O(n)$	$O(n)$
query/verify time	$O(1)$	$O(1)$

**Table 5. License ticket scheme**

sented public-key protocols for binding software licenses to smart cards and transferring licenses between cards. Our model supports software distribution through retail stores, wholesale from software publishers, and over the Internet. The user can partially transfer licenses from several cards onto a single card so that juggling between several cards in the reader is eliminated. Given that the smart card has only limited storage capacity, most of the license information is stored outside the card where it is managed in a secure way.

## References

- [1] A. Aho, J. Ullman, and J. Hopcroft. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
- [2] T. Aura and D. Gollmann. Software license management with smart cards. In *Proceedings of the USENIX Workshop on Smart Card Technology*, USENIX Association, May 1999.
- [3] T. Aura. Distributed access-rights management with delegation certificates. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS. Springer, 1999.
- [4] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology CRYPTO 94*, volume 839 of Lecture Notes in Computer Science, pages 216-233, 1994. Springer-Verlag.
- [5] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 45-56, 1995.
- [6] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures (extend abstract). In T. Helleseth, editor, *Advances in Cryptology (Proceedings of EuroCrypt '93)*, pages 274-285, Lofthus, Norway, May 1993.
- [7] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, vol. 12, pages 225-244, 1994. Springer-Verlag.
- [8] T. A. Budd. Protecting and managing electronic content with a digital battery. *Computer*, 34(8):2-8, August 2001.
- [9] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18-36, February 1990.
- [10] H. Chang and M. J. Atallah. Protecting software code by guards. *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, Pennsylvania, November 2001.
- [11] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998.
- [12] C. M. Ellison, B. Franz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet draft, IETF SPKI Working Group, March 1998.
- [13] M. Fischlin. Incremental cryptography and memory checkers. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT*, LNCS 1233, pages 393-408, 1997.
- [14] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792-807, October, 1986.
- [15] M. T. Goodrich, A. Schwerin, and R. Tamassia. An efficient dynamic and distributed cryptographic accumulator. Technical Report, Johns Hopkins Information Security Institute, 2000.
- [16] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specification, version 2.0. Internet draft, IETF Network Working Group, September, 1998.
- [17] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131-133, 1995.
- [18] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 135-150, October, 2000.
- [19] T. Maude and D. Maude. Hardware protection against software piracy. *Communications of the ACM*, 27(9):950-959, September 1984.
- [20] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology CRYPTO 89*, volume 435 of Lecture Notes in Computer Science, pages 218-238, 1990. Springer-Verlag.
- [21] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 217-228, Berkeley, 1998.
- [22] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993-999, 1978.
- [23] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1995.
- [24] D. Skeen. A quorum-based commit protocol. In *Berkeley Workshop on Distributed Data Management and Computer Network*, number 6, pages 69-80, February 1982.
- [25] C. Wang. A Security Architecture for Survivability Mechanisms. PhD thesis, University of Virginia, School of Engineer and Applied Science, October 2000. <http://www.cs.virginia.edu/~survive/pub/wangthesis.pdf>.