International Journal of Information Security

Downloaded from antoniomana.es

Please cite as:

A. Mana, et al., “A Framework for Secure Execution of Software”,
International Journal of Information Security (IJIS), vol. 3, pp. 99-112, 2004.

# A Framework for Secure Execution of Software [*]

**Antonio Maña, Javier Lopez, Juan J. Ortega, Ernesto Pimentel, Jose M. Troya**

Computer Science Department, University of Malaga
Campus de Teatinos, 29071 - Málaga, Spain
e-mail: {amg, jlm, juanjose, ernesto, troya}@lcc.uma.es

Received: date / Revised version: date

**Abstract** Nowadays, piracy is considered one of the major problems of software industry. We can find in the literature many research initiatives that have tried to solve this problem, and most of them are based on the use of tamperproof hardware tokens. This type of solutions depends on two basic premises: (i) to increase the physical security by using tamperproof devices, and (ii) to increase the complexity of the analysis of the software. The first premise is reasonable. The second one is certainly related to the first one. In fact, its main goal is that the pirate user can not modify the software to bypass an operation that is crucial: checking the presence of the token. However, the experience shows that the second premise is not realistic because the analysis of the executable code is always possible. Moreover, the techniques used to obstruct the analysis process are not enough to discourage an attacker with average resources.

In this paper, we review the most relevant works related to software protection, present a taxonomy of those works and, most important, we introduce a new and robust software protection scheme. This solution, called SmartProt, is based on the use of smart cards and cryptographic techniques, and its security relies only on the first of previous premises; that is, Smartprot has been designed to avoid code analysis and software modification. The entire system is described following a lifecycle approach, explaining in detail the card setup, production, authorization, and execution phases. We also present some interesting applications of Smartprot as well as the protocols developed to manage licenses. Finally, we provide an exhaustive analysis of its implementation details.

## 1 Introduction

There is a new generation of distributed applications, like distributed object systems, web services, electronic commerce and grid computing, that can provide substantial advances in the use of Internet resources. However, security problems become an insurmountable barrier for the widespread deployment of those types of applications.

We are facing a situation where security is still considered as a supplement for applications. That is, most of times it is not considered as a requisite during the design phase of the systems. On the contrary, security services are added during the implementation. Moreover, it is sometimes considered just as an external service. Everybody can observe a big gap between what technology can provide and what consumers actually get. The situation is producing an increasing consumers' skepticism.

The protection of software applications is one of the most important problems to solve because it is the foundation of other security issues. However, solving *software protection* problems is not a trivial task. Several areas of research concerning different aspects of this problem are still open:

- *Intellectual property protection*: Its objective is to link the software with information about its author by using techniques such as *watermarking* [6].
- *Protection against function analysis*: The objective here is to prevent a malicious host from discovering what function is computed by a software element. Techniques such as *code obfuscation* [5] or *function hiding* [17] are used, sometimes complemented by the use of hardware tokens [7].
- *Software use-control*: It is aimed to guarantee that only authorized users can run the software according to some contractual conditions.

The work presented in this paper focuses on the last of those problems. We also discuss the possibilities and implications of protecting software in order to: (i) avoid

[*] Work partially supported by the Spanish Ministry of Science and Technology under the Project TIC2002-04500-C02-02.

the analysis of the software operation, and (ii) ensure that the software performs the intended tasks.

It is important to realize that the secure software execution can open many possibilities in most of information security areas. The most important and direct consequence of the protection of software is the prevention of software piracy, which causes a loss of several billion dollars every year to software industry. In fact, the global piracy rate for PC business software applications reached 40% in 2002, with an estimate cost of $11 billion. In some countries the piracy rate climbed up to 96%.

According to the *Business Software Alliance* (BSA) estimations, the western European software industry lost $2.7 billion in 2001 due to illegal software. Also according to BSA, if software piracy had been reduced to 27% in Western Europe, more than a quarter of million of new employments would have been created, producing around $11.8 millions in taxes and $31.6 millions of total revenue.

Previous figures clearly show that the software protection problem remains unsolved. Therefore, the fight against software piracy is becoming an extremely important issue for the software industry. Surprisingly, the best results against software piracy during the last years have not been instigated because of new software protection mechanisms. On the contrary, they have been the consequence of other reasons like:

Software companies tried to have effective legal sales presence in all areas of the world; thus, software has become easier to purchase legally.

Software companies achieved better user support for their products (especially outside of the U.S.), thus promoting the purchase of legal software.

Price of software was reduced, narrowing the difference between legal and illegal versions.

Some organizations, like the BSA, promoted high profile legal proceedings against companies using illegal software.

Governments have cooperated to provide legal protection for intellectual property and to criminalize software piracy.

Unfortunately, after an initial slight decline trend, we are witnessing again an increment in software piracy. It seems that the previous measures will not achieve better results. Advances in code analysis tools and the popularity of Internet, among other circumstances, create new opportunities to copy software illegally. Moreover, actual legal protection tools such as *trade secrets*, *copyright*, *patents,* and *trademarks* are not adapted for the protection of software. In this sense, in [21] the creation of specific legal protection means for software products was proposed. However, that type of solution is very difficult to put in practice because it requires international agreements.

In this paper, we introduce a low cost software protection and license management scheme that is secure, flexible and convenient for users. This scheme, that is based on smart cards, avoids some of the most common attacks to software protection mechanisms, like multiple installations from a single legal license, reverse engineering analysis, and production of unprotected (pirated) copies of the software. The paper also shows how this protection system can be applied to build secure distributed applications.

In this sense, the rest of the paper is organized as follows. Section 2 reviews the most relevant work related to software protection. Section 3 introduces the new scheme that we propose. Some other interesting applications of this scheme are presented in section 4. Section 5 analyses the importance of implementation details and, finally, section 6 summarizes the conclusions and presents ongoing research and work.

## 2 Background

### *2.1 Taxonomy of software protection mechanisms*

In this section, we present a classification of the different approaches to the software protection problem. We focus on security, convenience and practical applicability. More extensive reviews of the state of the art in software protection can be found in [18][9].

Nowadays, much of the software does not include protection mechanisms. In those cases where software includes it, serial numbers and user/password schemes are usual protection schemes used. This lack of protection is mainly derived from two facts: (i) software manufacturers know that the usefulness of protection tools is unsatisfactory and (ii), users are reluctant to accept protection mechanisms that are inconvenient. Among the huge diversity of proposals, we can find two main categories of protection systems: the autonomous ones and those based on external collaboration.

*2.1.1 Autonomous systems* Systems in this category use protection mechanisms that rely on the software itself. Some systems are based on the difficulty of reverse engineering the protected software. However, most of times, the foundation are mechanisms that check if certain conditions are met.

*Systems based on the difficulty of analysis*

Several techniques can be applied to the software products in order to verify self-integrity. Anti-tamper techniques, such as checksumming, anti-debugging, encryption, anti-emulation and some others [23][25] are in this category. Some schemes are based on self-modifying code, and code obfuscation [5].

A different approach is represented by software watermarking techniques [28][9]. In this case the purpose of the protection is not to avoid analysis but to detect whether the software has been copied or modified.

The relation between these techniques is strong. In fact, it has been demonstrated that neither perfect obfuscation nor perfect watermark exists [4]. All of these techniques provide short-term protection; therefore, they are well suited in situations where software is useful for a short time (this is the case with agents and applets).

*Systems based on "checks"*

This is the most common case. In these systems the software includes "checks" to test whether certain conditions are met. We can distinguish solutions based exclusively on software, and other ones that require some hardware component. One of the most popular protection mechanisms consists in a password or key check that enables installation or execution of the software. If the check fails the software is not installed or it works in "demo" mode with restricted functionality. Because the password validation function is included in the software, it is easy for dishonest users to produce key generation programs. Authentic passwords are also easy to find in certain Internet sites.

Other schemes adapt the software for each specific computer. For instance, some of them extract information from some of the hardware devices (hard disk, network adapter, etc.) or from the operating system configuration. During its execution, the protected software checks that it is running on the computer it was personalized for.

Among the solutions relying on hardware components, tokens that are difficult to duplicate are quite widespread. The protected software checks the presence of the token and refuses to run if the check fails. Examples of this type of systems are hardware keys or *dongles*. In this case, the check of the presence can be done in different ways. The simplest way is to read a value from the communication port; however, the interception of the communication in that port allows the attacker to replicate the token. Usually, and in order to avoid this attack, the software sends a value (called challenge) that the token has to process to obtain a return value. The software has to predict the result that the token should send back. Whatever the check is, it is not particularly hard to bypass this protection because the access to the communication port or the reader can be easily found in the executable code.

Summarizing, in both software and hardware solutions, the validation function is included in the software. Therefore, reverse engineering and other techniques can be used to discover it. All checks can be bypassed obtaining a completely functional copy of the software by simply modifying the code. This process can even be automated by specially designed programs called "cracks" or "patches". Theoretic approaches to the formalization of the problem have demonstrated that a solution exclusively based on software is unfeasible [8]. By extension, all autonomous protection techniques are also insecure.

*2.1.2 Systems based on external collaboration* It is a fact that if we want to obtain a provable secure protection scheme, a tamperproof processor must be used for both the storage and execution of the protected software [10]. Any collaboration-based scheme where the external collaborator entity is considered trustful can be included in this category. Among these systems we consider those which use online or offline collaborations, and those based on tamperproof hardware.

*Offline collaboration*

In some scenarios, such as agent-based ones, the protection required is limited to some parts of the software (code or data). In this way, the function performed by the software, or the data processed, are hidden from the host where the software is running. An external offline processing step is necessary to obtain the desired results.

Among these schemes, the most interesting approach is represented by function hiding techniques. In [22] the authors present a scheme that allows evaluation of encrypted functions. The fundamental idea is to establish an homomorphism between the spaces of cleartext and encrypted data, with the objective of evaluating a certain function on some data without revealing them. This process can be expressed this way:

Let $P$ be the domain of cleartext data and $Q$ the domain of encrypted data. Let $f : P \to P$ be a function that the user wants to evaluate on some $x \in P$, and let $e : P \to Q$ and $d : Q \to P$ be, respectively, the encryption and decryption functions of some cryptosystem. Then, under certain conditions of the original function $f$, it is possible to find $f' : Q \to Q$ such that $\forall x \in P \; f'(e(x)) = e(f(x))$ or, using an alternative of the previous expression, $\forall x \in P \; d(f'(e(x))) = f(x)$. This property is useful because it allows a piece of software to store $e(x)$ and implement $f'$ in order to compute $f'(e(x))$ without revealing $f$, $x$ or $f(x)$. Unfortunately, this property only holds for certain families of functions (polynomial ones in this case).

*Online collaboration*

The case of online collaboration schemes is also interesting. In these schemes, part of the functionality of the software is executed in one or more external computers. The security of this approach depends on the impossibility for each part to identify the function performed by the others. This approach is very appropriate for new distributed computing architectures such as agent-based systems or grid computing.

*Tamperproof hardware*

Some protection systems have been proposed based on tamperproof hardware devices. The software is distributed encrypted and a tamperproof embedded processor is used to decrypt it before it runs on the computer. The drawback is that, once decrypted, the software is stored in the RAM memory of the user's computer. Different techniques can be used at that moment to recover the software (e.g. producing a core dump).

An excellent variation of the previous scheme is the distribution of encrypted code that the tamperproof processor decrypts and executes [3]. This made us consider the use of tamperproof processors as the first building block of our software protection scheme.

On the other hand, Aura and Gollmann presented in [1] an interesting scheme based on smart cards and digital certificates that solves the card juggling problem and provides mechanisms for license management and transfer. In addition, a compilation of countermeasures against attacks are reviewed in their work. Unfortunately, as their proposal relies on the check of the presence of the smart card, it is vulnerable to the code modification attacks described before.

Additionally, it is possible to design a secure software protection scheme based on a secure coprocessor without cryptographic capabilities. In this scheme some sections of the software to be protected would be substituted by functionally equivalent sections stored and processed in the coprocessor. The protected software is divided and will not work unless it cooperates with the right coprocessor. Code modification attacks will not succeed in this case. In fact, the most practical attack is to analyse the data transmitted to and from the coprocessor, trying to guess the functions that it performs. If the number of functions, their importance in the main code, and their complexity are large enough, the analysis attack described will become impractical. This scheme requires that the processor is distributed with the protected software sections preloaded. It uses one processor per application; hence, it is affected by the card-juggling problem. The quantity and complexity of the protected sections are limited by the capacity of the processor. Moreover, this scheme does not facilitate the Internet distribution of the protected software because the processors must be distributed with the protected software sections preloaded.

A slightly modified version of the previous scheme is introduced in the ABYSS architecture [29]. In this system some processes of the software to be protected are substituted by functionally equivalent processes that run inside a secure coprocessor. The processes are encrypted while outside of the secure coprocessor. The main disadvantages of the ABYSS system are the need to use special tokens to authorize the execution of the protected software, the impossibility to distribute the protected software through Internet because the tokens (physical objects) must be distributed with the keys preloaded, and the fact that the encryption of the sections is done with a common supervisor key of the protected processors. The last disadvantage is especially important because it introduces the possibility for dishonest users to produce fake protected processes. This is so even in the case that the protected processor uses an asymmetric encryption scheme because, in this case, the public key of the processor will be known by everyone. SmartProt has been designed to overcome the previous disadvantages.

It is important too point out that the lack of a code authentication mechanism represents an open door for a dangerous attack in any system. In the case of software protection systems, such attack can be based on the substitution of some of the authentic protected sections by other fake sections produced by the dishonest user. For instance, a false section could be produced to extract the data stored in the secure processor. This attack can be considered a kind of “Trojan horse”. To avoid this kind of attacks we conclude that the protected code needs to be authenticated.

### *2.2 Smart Cards*

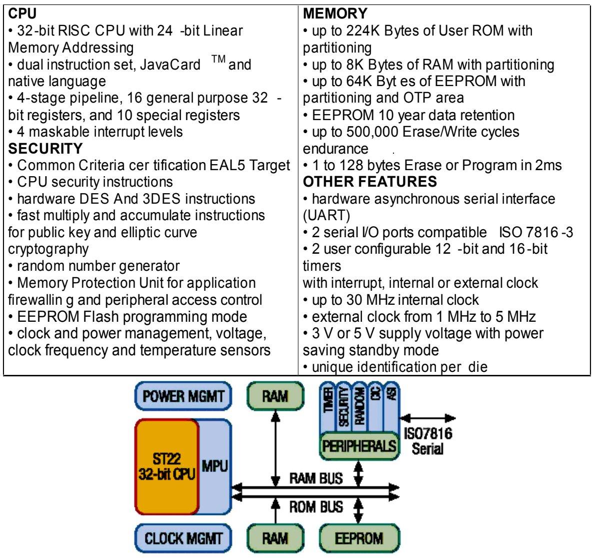
Smart cards represent a qualitative advance in the way to practical information security. Until the introduction of smart cards, the ability to produce digital signatures and other cryptographic primitives was limited by the necessity of using a provable secure and trusted computing environment. In practice, this necessity was very difficult to achieve, especially in environments with a high degree of mobility. Smart cards solve this problem because they are secure, tamperproof and portable computing devices capable of storing sensible information (such as biometric profiles) and performing computations required in digital signatures and other cryptographic primitives.

Programmable smart cards, such as Java Card [26], facilitate the development of specific applications and provide tools to achieve security properties that can not be supported by cryptographic protocols and algorithms alone. These cards allow the issuer to control the information that they contain. In this sense, the combination of the physical security and the fact that the software that they execute is under control of the issuer, are the key to achieve those security properties.

Two main problems have traditionally hindered the widespread use of these devices: (i) the difficulty of integration of smart card applications in personal computing environments and, (ii) the reduced data transmission speeds between cards and hosts. The new dual-interface smart cards open the door to the solution of both problems because they make use of two contacts “reserved for future use” in the ISO7816 [11] standard to provide a USB interface in addition to the traditional ISO 7816.

Nowadays, semiconductor industry has achieved important advances in the development of smart card processors. Among these, we must highlight the availability of RISC processors, the integration of USB controllers in the smart card chip, and the implementation of the Java Card virtual machine in hardware.

To illustrate the power of current smart cards, we can use the ST22 family of 32 bit processors from ST Microelectronics, shown in figure 1. These processors have been specifically designed for multi-application smart cards. The ST22 processors have a 32 bits RISC CPU, with hardware support for most of the Standard Java Card 2.1 virtual machine instructions, as well as a proprietary native code. Some of them include a hardware USB controller.



**CPU**
- 32-bit RISC CPU with 24 -bit Linear Memory Addressing
- dual instruction set, JavaCard TM and native language
- 4-stage pipeline, 16 general purpose 32 -bit registers, and 10 special registers
- 4 maskable interrupt levels

**SECURITY**
- Common Criteria cer tification EAL5 Target
- CPU security instructions
- hardware DES And 3DES instructions
- fast multiply and accumulate instructions for public key and elliptic curve cryptography
- random number generator
- Memory Protection Unit for application firewallin g and peripheral access control
- EEPROM Flash programming mode
- clock and power management, voltage, clock frequency and temperature sensors

**MEMORY**
- up to 224K Bytes of User ROM with partitioning
- up to 8K Bytes of RAM with partitioning
- up to 64K Byt es of EEPROM with partitioning and OTP area
- EEPROM 10 year data retention
- up to 500,000 Erase/Write cycles endurance
- 1 to 128 bytes Erase or Program in 2ms

**OTHER FEATURES**
- hardware asynchronous serial interface (UART)
- 2 serial I/O ports compatible ISO 7816 -3
- 2 user configurable 12 -bit and 16 -bit timers
with interrupt, internal or external clock
- up to 30 MHz internal clock
- external clock from 1 MHz to 5 MHz
- 3 V or 5 V supply voltage with power saving standby mode
- unique identification per die

**Fig. 1** Architecture and summary of features of the ST22 family of processors (Courtesy of ST Microelectronics)

Attacks to smart cards are often divided in physical or invasive and logical or non-invasive [27][20]. The first category is based on the physical destruction of the chip in order to access the information contained in it [14][2].

Therefore, the security of smart cards is highly related to their tamperproof design. Many different techniques, such as *Layering*, *Address Scrambling*, *Bus Scrambling*, *Tamper Detection Mechanisms*, *Zeroization Mechanisms* and *Glue Logic*, among others are used to prevent physical attacks.

The second category is based on exploiting the vulnerabilities and bugs in the design of some smart card elements, such as software and communication protocols. Some of the most well known attacks of this type are: *Timing Attack* [13], *Simple Power Analysis*, *Differential Power Analysis* and *High Order Differential Power Analysis* [12]. We must note that these attacks require the knowledge of the card PIN because they require the execution of many cipher operations on specific inputs. There are effective solutions to avoid these attacks [24]. Some other attacks are based on *Trojan Horses*, *Social Engineering* and *Trust Splits* [18].

## 3 Description of SmartProt

This section shows the details of the protection scheme. In the description we will use the following conventions:

– $X1 \rightarrow X2 : M$ states that message $M$ is send to user $X2$ by $X1$.
– $M(A, B)$ emphasizes that message $M$ is composed of data items $A$ and $B$.
– $Xp$ and $Xs$ denote, respectively, the public and private key of user $X$.
– |I| represents the digest of some information. It is computed using a secure one-way hash function.
– $K[I]$ represents the encryption of some information using the key $K$.
– $X\{I\} = (I, Xs[|I|])$ represents the signature of $I$ by user $X$.
– $X1 \ll X2 \gg$ denotes the certificate of user $X2$ issued by $X1$.

In order to avoid the problems identified in the study of previous software protection systems, we introduce cryptographic techniques as the second building block of our software protection scheme. In particular, we use an asymmetric cryptosystem to secure the communication among actors and a symmetric cryptosystem to protect the software. The design of our scheme is illustrated using smart cards as secure coprocessors. However, and as mentioned previously, other tamperproof coprocessors are also suitable for our approach.

We describe now the basic infrastructure required by our system. The system includes three types of actors: software producers, card manufacturers and clients (each client possessing a smart card). For the sake of the description we will assume the simplest certification scheme, where card manufacturers certify the public keys of the smart cards and also those ones of the software producers. More sophisticated schemes are considered in [18].

Our system requires smart cards that have cryptographic capabilities, contain a key pair generated inside the card, and ensure that the private key never leaves it. The cards must also contain a specific symmetric key, the public key of the card manufacturer and some support software. In particular, the cards must contain the SmartProt virtual machine, which is described in section 5.1.

The basic idea is to use the smart card as a secure coprocessor to enforce the correct execution of the protected parts of the program. These parts must be carefully selected in order to obtain the best protection. The entire system is described, following a lifecycle approach, by dividing it into four phases: card setup, production, authorization and execution. Figure 2 shows the actors and responsibilities involved in the different phases. The following sections explain in detail each one of these phases.

### *3.1 Card setup*

The card setup phase prepares the card to be used in the system. This phase can be repeated whenever the client

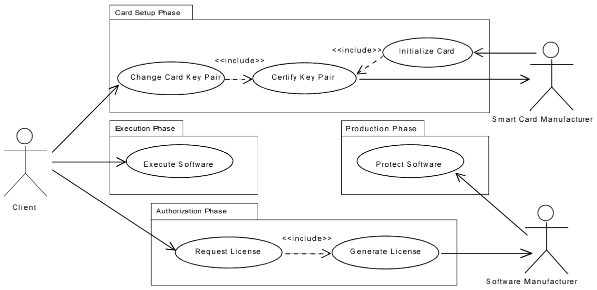


**Fig. 2** Overview of actors, responsibilities and phases

(**C**) wants to change the key pair of the card (**CC**). The setup of the card starts when a new key pair is generated inside it. The new public key and the card secret ID (**CCID**) are sent to the client card manufacturer (**CCM**), encrypted with its public key. The card manufacturer verifies that the encrypted secret ID corresponds to that card and creates a certificate of that public key, which is sent back to the card and stored inside it.

### *3.2 Production*

The production phase is depicted in figure 3. During this phase the software is transformed in order to protect it against reverse engineering analysis and unauthorized execution. This phase is performed only once time for each release of the software because it does not depend on the client card.

The first steps of the production phase consist in the selection and translation of some specific sections of the original application code with functionally equivalent sections of *SmartProt* card-specific code. The code selection process also reorganizes the code to build manageable protected sections and finds dependencies between these sections in order to identify which values can be kept inside the smart card. Basically, the translation is a semantic-preserving transformation from the Java object code (bytecode) to the *SmartProt* virtual machine code. Additionally, obfuscating transformations are applied to the rest of the code, and fake code is introduced in order to make more difficult for the attackers to guess the functions of the protected sections. This confusion (obfuscation) process is highly convenient because it hides the purpose of the protected sections. We must emphasize that this process is not applied to the protected sections but to the rest of the code.

Once translated, the selected sections are encrypted using a symmetric cryptosystem with a randomly generated key. As shown in figure 3, the last step substitutes the original code sections with calls to a function that transmits the respective equivalent protected sections, including code and data, to the card. Some additional support functions are also included. The protected software application generated in the production phase can be distributed and copied freely.

### *3.3 Authorization*

Once the card has been set up, a license (specifically created for the smart card of the user) is required to use the protected software. In the authorization phase, software manufacturers generate new licenses, as shown in figure 4, containing:

- the random symmetric key used to encrypt the protected sections;
- information about conditions of use (i.e. time limits, number of executions, etc.);
- identification of the software (ID, version number, etc.);
- identification (or serial number) of the license; and, finally,
- a random number received from the client.

The license is generated encrypting all this information with the public key of the smart card of the client, and must be loaded prior to the execution of the application. In some usage scenarios the loading of the license can be part of the protected application. When the client smart card receives the license, it is decrypted, verified and stored inside the card until it expires or the user explicitly decides to extract it.

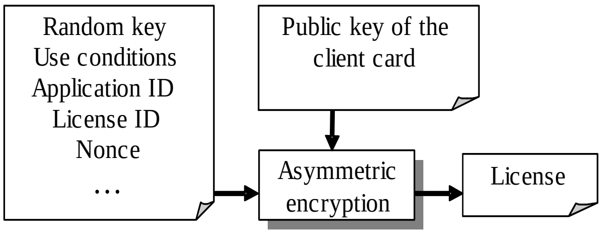


**Fig. 4** License generation (authorization phase)

Figure 5 summarizes and puts into context the processes carried out during the production and authorization phases.

### *3.4 Execution*

The execution phase is depicted in figure 6. Once the license is correctly installed in the card the protected program can be executed, which will require the cooperation of the card containing the license.

The protected sections of the software do not reside in the cards. Instead, during the execution of the protected program, these sections are transmitted dynamically as necessary to the card, where they are decrypted using the installed license, and then executed. When finished, the card may send back some results, but as we

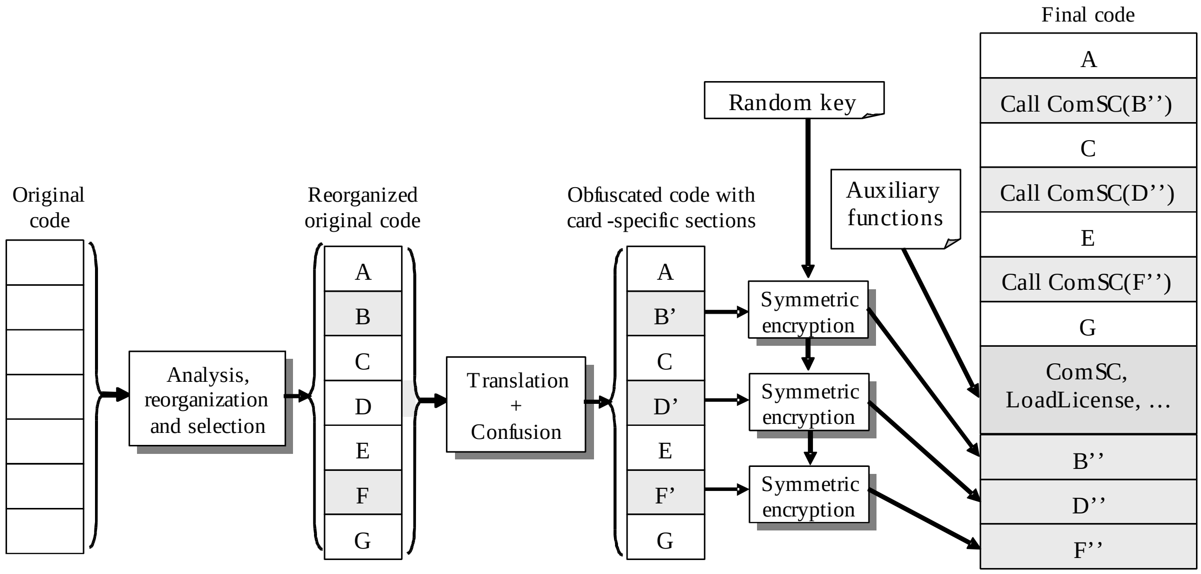


**Fig. 3** Software protection (production phase)

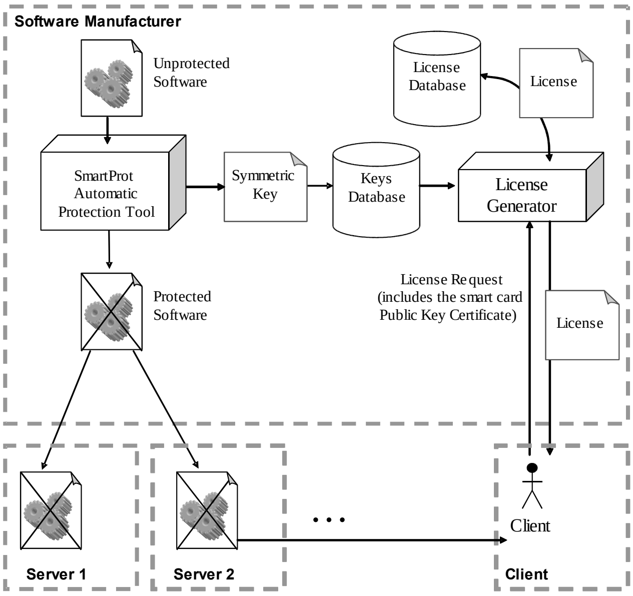


**Fig. 5** Summary of production and authorization phases

will explain later, some partial results will be kept in the card in order to obtain a better protection against function analysis and other attacks.

## 4 License management protocols

It has been mentioned that each license is specific for a smart card. This licensing model does not imply that licensing schemes such as "site licenses", "campus licenses" or other similar schemes are not available. The SmartProt licensing scheme allows all these schemes to be simulated. For instance, a "campus license" for the students of the University of Málaga can be implemented in several ways. One of them, part of our work in progress, is to apply the Semantic Access Control (SAC) [30] model to control the license generation procedures. This approach enables licenses to be issued to a user presenting an attribute certificate attesting the enrollment to the University of Málaga.

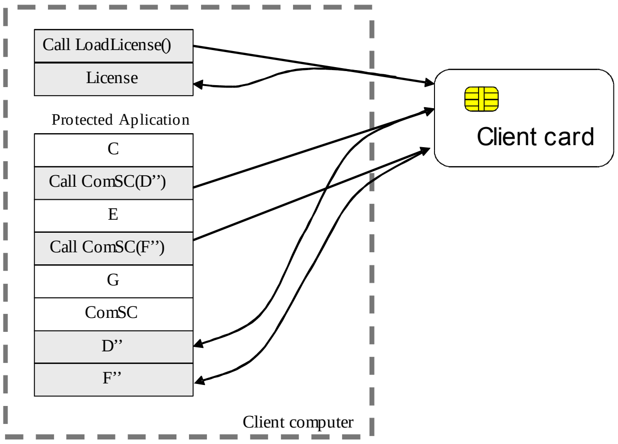


**Fig. 6** Execution phase

### *4.1 Sale*

Because the license for the user (containing the key to decrypt the protected sections) is encrypted with the card public key, it is essential to avoid that the corresponding private key is not disclosed outside the card. In order to achieve this objective the most practical solution is to use special smart cards produced for this purpose. These cards must contain a key pair and some support software. A certificate of the public key of the card is issued by the card manufacturer to guarantee the authenticity of the keys.

When a client (**C**) wants to buy a protected application, requests a certificate of the public key of the software producer (**SP**) issued by the client card manufacturer (**CCM**). The client then sends back a request containing the certificate of the public key of his/her card (**CC**), the identification of the software to be purchased and a random number encrypted with the received public key. The producer verifies the validity of the certificate and, in case the validation succeeds, produces a new license, encrypts the license and the random number using the public key received and, finally, sends it to the client card. The card verifies that the license matches the request (i.e. the license info is correct and the random number matches the one in the request) and stores it. The steps of the protocol are shown in the appendix too.

The producer also stores all the licenses in a database in order to generate new licenses for the client, when needed (theft, destruction of the card, etc.). If a request for a previously generated and valid license is received, the producer will prepare a new license for the client at no extra cost. This new license will include a different serial number (the number is part of the identification of the license). The serial number is an important element of the license transfer protocol, as we will show in section 4.3.

### *4.2 Expiration*

The licenses are always kept protected because they are either encrypted or stored in the smart card. Therefore, the card software, which is trustworthy, is able to destroy licenses when they expire (different parameters can be used to define the expiration of the licenses; for instance, the number of executions, time of use, etc.). The software can even warn the user when the expiration is about to happen. One of the most used parameters in software licenses is the expiration date. To include this feature, cards should include an internal real time clock. Some manufacturers have announced cards supporting this feature.

### *4.3 Transfer*

License transfer is one of the features that we have considered important. License transfer could be used to delegate the right to use a software application to another user or simply to store your license in a new card. In contrast to other systems that can only transfer all the licenses in a pack, our scheme introduces the possibility of selective license transfer.

Our license transfer scheme has been designed to avoid using certificate chains because of the overhead in communication, storage and processing that they introduce. Another important goal has been to avoid storing public keys of external entities in the smart cards.

We call this protocol *direct transfer*, in contrast to *scheduled transfer*, which is mainly used for backup purposes. Protocol steps are shown in the appendix.

The protocol to transfer a license is divided in two phases: delegation (steps 1 to 3) and recover (steps 4 to 6). We can summarize those steps in the following way:

1. The destination card (**DC**) public key certificate is sent to the source card user (**SCU**).
2. The user selects which license (or licenses) is going to be transferred from the source card (**SC**). Note that, opposite to other systems, our scheme does not force the user to transfer all the licenses in the source card (which we consider to be a serious limitation). In the rest of this protocol we will assume that we are transferring one specific license.
3. The source card creates a certificate delegating the license to the public key of the destination card, destroys its own license and, finally, sends the delegation certificate to the destination card.
4. The destination card requests a new license to the software producer. This request includes the delegation certificate received from the source card, together with the destination card public key certificate.
5. The software producer verifies both certificates, and generates a new license for the destination card in case the verification succeeds. The license database is updated accordingly.
6. The destination card decrypts and stores the new license.

Let's suppose now that the protocol described above is interrupted (either accidentally or intentionally to attack the scheme). Any interruption of the protocol previous to step 3 does not produce any problem because it does not change anything in the card. On the other hand, if the protocol is aborted during step 3 (e.g. extracting the card from the reader), just after the source card has destroyed its license, then none of the cards gets the license. However, in this situation, the source card can request a replacement copy of the deleted license from the software producer.

In case the protocol is aborted after step 3, the destination card would possess the delegation certificate but not the new license. In this case, the source card would have already destroyed its license. Then, it could request a replacement copy from the software producer and get a new valid license. Using the delegation certificate that has stored, the destination card could also get a new license. This attack could be used to replicate any number of licenses. To prevent this attack, a serial number, different for each new copy of the license produced, is included in the license (see section 4.1).

In the scenario depicted above, when the source card requests the new copy after aborting the protocol, the software producer generates a new license (with a different serial number) that is sent to the card. The previous license is invalidated and substituted by the new one in the database. Later, when the destination card attempts to use the delegation certificate to get a new license, the request will be denied.

The inclusion of the software producer in the transfer protocol may seem inconvenient. However, if the producer is not included, the source card would need to verify the public key certificate of the destination card which, in turn, would increase the complexity of the protocol and also would introduce weaknesses in the protection scheme.

### *4.4 Recovery*

It is essential for user acceptance to provide efficient and convenient solutions to the problems that the protection scheme itself may introduce. In our scheme, licenses are linked to smart cards based on the fact that the private key is not known outside the card. Consequently, and in case of card failure, all licenses prepared for it will be useless, which means that it will be impossible to run the software. For this eventuality, the user must take some prevention measures.

Considering that the smart cards are inexpensive, it seems reasonable to prepare a replacement card to be used in case of failure of the main card. The preventive process requires the execution of the delegation phase of the scheduled transfer protocol for all the licenses in the card. In case of failure of the main card, the protocol would continue on the recover phase. At the end of the protocol the replacement card will contain the same licenses as the main card.

The difference between the direct transfer protocol and the scheduled transfer protocol is the inclusion of the date when the transfer must take place. This date is included in the delegation certificate. Steps 3 and 4 of the direct transfer protocol are replaced by this sequence in the scheduled transfer protocol:

The source card creates a certificate delegating the license to the public key of the destination card on date "**Date**" and sends it to the destination card. The source card will not be able to delegate that license again to any other card until date "**Date**".

Later, two different situations can arise:

If the user wants to keep using the main card, the replacement card must destroy the delegation certificate and send a new scheduled transfer request before date "**Date**". In this case, the source card will accept the request.

Otherwise, on date "**Date**":

Source card will destroy its own license.

As in the direct transfer case, both cards can request a new license to the software producer but only the first to happen will be accepted.

### *4.5 Deletion*

The licenses can be deleted either when they expire, have been transferred to other card, or are not necessary anymore. In the latter case, the user must explicitly request the deletion of the license. We must emphasize that license expiration depends on the terms of use established in the license and the features of the secure coprocessor used. In the case of smart cards, expiration is currently based on the number of executions of the software, but some manufacturers have proposed smart cards with an internal time source. This feature would enable license expiration and deletion to be based, for example, on the time of use. Automatic license deletion is performed as part of the card initialization process. Every time that the card is inserted in a reader the License Manager checks the validity of the licenses in the card and deletes those that have expired.

### *4.6 Extract and restore*

The process of producing a backup of a license must be done using a second smart card and following the recovery protocol. However, sometimes it might be desirable to temporarily extract a license from a smart card in order to save space and still be able to load it back later.

When the smart card extracts the license it produces and stores a random **Nonce** (a process analogous to the production of the license request –**LicReq**- in the sale protocol). Therefore, the license can be restored simply by sending it to the card as if it was a new license. It will be accepted by the card because the **Nonce** included in the extracted license is registered inside.

## 5 Considerations on implementation and security

Java Cards are the best choice to make our approach a practical solution. The reason is that they provide some of the components that we need for our application. However, some limitations of the Java Card specification have important consequences on our implementation. Among these limitations, we highlight: lack of file management, dynamic code loading scheme and dynamic memory management.

### *5.1 Card software and runtime environment*

The SmartProt virtual machine is composed by an interpreter for the protected code sections, a license manager, a runtime manager and, sometimes, an electronic purse. In this virtual machine, every application runs in a sandbox, isolated from others. The card is only capable of executing one protected code section at a time, but several SmartProt-protected applications can run in the host computer simultaneously. The SmartProt runtime manager keeps separate memory spaces for each application. The contents of the memory used by each application are kept in memory after each call to the card is finished. This allows the application to use the values stored in the card in the processing of subsequent calls to the card.

The structure of the memory of the card during runtime is depicted in figure 7. In our implementation, the core components of the SmartProt virtual machine (code loader, license manager and runtime manager), as well as an optional payment system (e-Purse), are implemented at the application level by a Java Card applet.

However, applications at this level have performance constraints (e.g., they are isolated by firewalls) and limitations. We are currently collaborating with a card manufacturer in the implementation of those functions at a lower level in order to obtain a better performance and, at the same time, to enable the deployment of other applications that take advantage of the software protection infrastructure.

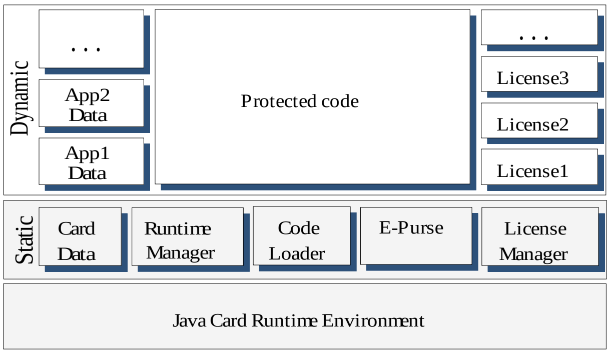


**Fig. 7** Card software components and memory structure

The static elements are preloaded in every card. Basically, they include some card-specific data and the SmartProt applet. The four main components of the SmartProt cards are implemented as a single card applet because of implementation details. These components are:

License Manager. It is responsible for all operations related to licenses (loading, deletion, transfer, backup, etc.).

Code Loader. Upon the reception of an encrypted code section in the card, the Code Loader locates and analyses the corresponding license, decrypts the code and stores it in the Protected Code section.

Runtime Manager. This component is required because of the lack of dynamic memory management in Java Card. It allocates and reallocates memory for the applications and supervises the separation between the different applications.

E-Purse. This is an optional component that has been developed to provide a fair payment mechanism for digital content commerce. The initial goal was to bind together two operations: access to the contents and payment. For this reason, existing e-purse designs were not appropriate, and we have developed an specific one.

The dynamic part of the applet can be configured during its installation by specifying the amount of memory to be used for licenses, for application data and for protected code.

### *5.2 Card code*

The lack of mechanisms that allow on-the-fly code execution in the standard Java Card has forced us to define specific a Smartprot virtual machine and an associated language.

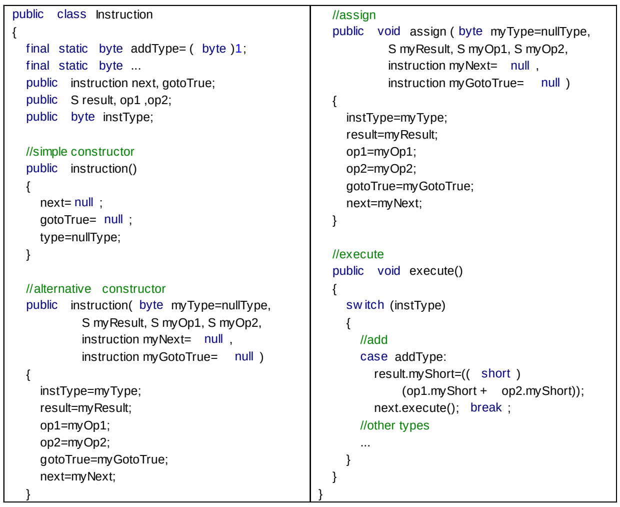
Regarding the language, our basic objective has been to achieve a compact yet powerful and flexible representation of the instructions to be executed in the card. Because the main performance bottleneck of smart cards applications is the communication with the card, we have defined a compact format for the storage and transmission to the card. Upon reception, the card decrypts the protected code using the corresponding license and then translates into the internal format. The use of this particular internal format overcomes the problems associated to the lack of file management functionality in Java Card.

We have defined the Instruction class in the Java Card language in order for the instructions to be self-contained and to achieve easy referencing between instructions. Together with the SmartProt card applet, this class constitutes the SmartProt virtual machine. This representation eliminates the need to put a standard interpreter in the card (the interpreter is the Instruction class itself), which in turn results in greater flexibility. Figure 8 shows the definition of the Instruction class.

Once the code is loaded into the card and converted into an array of Instruction objects, the execution is as simple as calling the Execute method of the first object of the array. Each instruction is linked to the next one(s) to be executed.

### *5.3 Efficiency vs. security*

Smart card technology offers today a set of features that some years ago only personal computers could offer. However, and in comparison to the processing power of actual host computers, every access to the smart card introduces important delays. As our scheme requires the transmission of a considerable amount of code and data to and from the card, it is important to take into consideration the efficiency of the protection scheme. It has



```
public  class  Instruction
{
   final  static  byte  addType= (  byte )1;
   final  static  byte  ...
   public   instruction next, gotoTrue;
   public   S result, op1 ,op2;
   public   byte  instType;

   //simple constructor
   public   instruction()
   {
      next= null  ;
      gotoTrue=  null  ;
      type=nullType;
   }

   //alternative  constructor
   public   instruction(  byte  myType=nullType,
               S myResult, S myOp1, S myOp2,
               instruction myNext=   null  ,
               instruction myGotoTrue=    null  )
   {
      instType=myType;
      result=myResult;
      op1=myOp1;
      op2=myOp2;
      gotoTrue=myGotoTrue;
      next=myNext;
   }

   //assign
   public   void  assign (  byte  myType=nullType,
               S myResult, S myOp1, S myOp2,
               instruction myNext=   null  ,
               instruction myGotoTrue=    null  )
   {
      instType=myType;
      result=myResult;
      op1=myOp1;
      op2=myOp2;
      gotoTrue=myGotoTrue;
      next=myNext;
   }

   //execute
   public   void  execute()
   {
      switch (instType)
      {
         //add
         case  addType:
            result.myShort=((   short  )
                  (op1.myShort +    op2.myShort));
            next.execute();   break  ;
         //other types
         ...
      }
   }
}
```

**Fig. 8** The `Instruction class`

been proved that the main bottleneck in the performance of smart card applications is the communication between the card and the host [LMP00, MaPi01]. Therefore, new USB smart cards with a bandwidth of 256Kbits/s have greatly reduced this problem if we compare the situation with the 9.8Kbits/s of the standard ISO7816 interface.

The amount of data and code transmitted determines the magnitude of the delay introduced. On the other hand, since the main attack to the protection scheme is based on the analysis of the functions performed by the card, the protection scheme will be more secure as the functions grow in size and complexity.

Consequently, it is necessary to find a balance between security and speed. Fortunately, in this case, this balance is possible and it is not difficult to obtain security and speed ranges that satisfy both software producers and clients. A theoretical description and some results about the efficiency of the protection scheme are included in [18].

### *5.4 Interception of communication*

Quite often piracy occurs inside the organization of a legal buyer of the software; that is, multiple illegal copies of a legally acquired software application are done and used into the same organization. In our scheme, these piracy procedures would succeed by having several computers sharing the card reader.

This problem has been considered in previous schemes. The most common solution is to make the software have direct access to the card reader (coprocessor). However, this solution introduces countless problems and computational costs in the protected software because it must manage different situations and hardware features that should be managed by the operating system.

In order to prevent this attack, we have designed a solution in our scheme that is based on the last technique described in section 5.5. The system "links" the calls to the card, storing intermediate results inside it. Therefore, any incorrect sequence of calls, like that one produced when several computers share a card reader, will make the software to produce erroneous results.

### *5.5 Functions executed by the smart card*

This is a very important issue because the security of the system relies on the difficulty of guessing, from the analysis of the input and output data (and possibly the execution time), the functions that the smart card executes.

If we know that the function performed by the smart card takes the form $y = f(x) = ax + b$, then we just need to run the function twice, with different input data, to infer it. Contrarily, other functions, like hash functions and digital signatures are not vulnerable to these attacks. These types of functions are not used in most of software applications. On the other hand, the functions we can frequently find in software applications have more input and output data, what becomes an advantage for our protection purposes.

There are other schemes that require the transformation of some of the functions of the application code into very specific representations. This facilitates to execute them in a very simple secure coprocessor. However, there are two important drawbacks in this approach:

It is very difficult to identify the parts of an arbitrary application that can be transformed and protected by these schemes. Thus, they can only be applied at source code level, and human intervention is required in the protection process.

The assumption about the simplicity of the coprocessor is in contradiction with the requirement of needing a powerful coprocessor to obtain a reasonable level of protection. Because actual coprocessors, such as smart cards, are real computing platforms, there is no need to perform the representations.

In our scheme, and in order to make more difficult for the pirate to analyze the functions, we include false (dummy) input and output data in the protected sections. These data are not used for the computation of the function, though it is transformed to confuse the attacker. Another very effective technique that we use to obstruct the analysis is to mix several functions so that the result of each call to the coprocessor depends on the input data of the previous calls. Moreover, it depends on results of previous calls that have not been sent back as results but stored in the coprocessor memory. The consequence is that the card can not be used to run, simultaneously, two copies of an application if only one license is used. This would produce an incorrect operation of both copies of the application.

## 6 Concluding remarks

Assuming that the encryption algorithm is secure, the attack to the system must be based on the black-box analysis of the protected sections, as stated before. However, we must emphasize that our system is designed in such a way that the card stores one function at a time. Therefore, we can use more complex functions than other systems because all the capacity of the card is now available for every single section. Moreover, this scheme allows the card to execute any number of protected sections. The dishonest user will need to discover all of the protected sections in order to break the protection scheme.

As we have shown along the paper, our scheme allows: (i) that a single card is used to protect many applications; (ii) a high degree of complexity in the protected sections; (iii) that the card executes any number of those sections; and, (iv) the distribution of the software through Internet because none of the components of a protected application needs to be preloaded in the smart card.

The definition of the license structure allows a high degree of flexibility. Furthermore, because each application has its own license, we can manage them individually. This advantage is not possible in other proposals where the sections of all applications are protected using a common key (usually the protected processor key).

We also have explained the need to authenticate the code executed by the card in order to avoid certain attacks. In our scheme, because the protected sections are encrypted using a symmetric key that is kept inside the cards (and, therefore, known only by the software producer), it is impossible for a dishonest user to produce false sections.

Additionally, we have successfully applied the SmartProt scheme to the protection of mobile code (in particular, to Java applets). The results are also valid for other mobile code elements such as agents. In this sense, the XSCD infrastructure [16][30] is based on the dynamic creation of mobile software elements protected by a variant of SmartProt. These elements, that we call *Protected Content Objects* (PCOs), are responsible for the transport of the protected content and the enforcement of the access control policy. The code of the applet contains encrypted sections that must be executed within the smart card. Because the base software of the cards is trustworthy and certified by the card manufacturer, we can control, for instance, the number of times that a license is used to execute the PCO, and introduce an integrated payment system.

Summarizing, we have described a robust software protection scheme based on the use of smart cards and cryptographic techniques. Related schemes based on tamperproof hardware tokens that have been proposed in the literature have been analyzed. We have concluded that all of them are based on the check of the presence of the token and are, therefore, vulnerable to code modification attacks. Considering that the new scheme is not based on that check, code modification is not a potential attack. Finally, we have also introduced possible applications of the scheme. Thus, we can conclude that the advantages of the presented scheme are: (i) robustness against different attacks (bypassing the check, code substitution and attacks to the license management protocols); (ii) efficient use of the computational resources of the smart cards; (iii) free distribution and copy of the software; (iv) selective license transfer; (v) control of the expiration of the licenses; and (vi) applicability in both mobile and distributed computing environments.

Tools to produce protected software automatically from unprotected executable programs, applet protection and payment integration are in an advanced stage of development. Regarding future works, we are involved in the study of the possibilities that can be opened by the integration of function hiding techniques in our scheme. Finally, we are working on the application of SmartProt for mobile code in Digital Rights Management (DRM) applications.

## References

1. Aura, T.; Gollmann, D. *Software License Management with Smart Cards.* Proceedings of the Usenix Workshop on Smartcard Technology (Smartcard'99), pp. 75-86. 1999.
2. Beck, F. *Integrated Circuit Failure Analysis, A Guide to Preparation Techniques.* John Wiley & Sons, 1998.
3. Bennet S. Yee. *Using Secure Coprocessors.* PhD thesis CMU-CS-94-149, Carnegie Mellon University, 1994.
4. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K. *On the (Im)possibility of Obfuscating Programs.* Proceedings of CRYPTO '01. Springer-Verlag. LNCS 2139. pp. 1-18. 2001.
5. Collberg, C.; Thomborson, C. *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.* University of Auckland Technical Report #170. 2000. Available online at http://www.cs.auckland.ac.nz/∼collberg/Research/Publications/CollbergThomborson2000a/index.html
6. Collberg, C.; Thomborson, C. *Software watermarking: Models and dynamic embeddings.* Proceedings of POPL'99 - 26th ACM Symposium on Principles of Programming Languages. 1999. Available online at http://www.cs.arizona.edu/∼collberg/Research/Publications/CollbergThomborson99a/index.html
7. Funfrocken, S. *Protecting Mobile Web-Commerce Agents with Smartcards.* Proceedings of ASA/MA'99. 1999.
8. O. Goldreich, *Towards a theory of software protection.* Proc. 19th Ann. ACM Symp. on Theory of Computing, pp. 182-194. 1987.
9. Hachez, G. *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards.* PhD Thesis. Universite Catholique de Louvain. 2003.
10. Herzberg, A.; Pinter, S. S. *Public Protection of Software.* ACM Transactions on Computer Systems, 5(4)-87, pp. 371-393. 1987.

11. International Organization for Standardization. *ISO/IEC 7816 (Parts 1 to 5)*. 1995-2002. Available online at http://www.iso.ch
12. Kocher, P. Jaffe, J. Jun, B. *Differential Power Analysis*. Cryptography Research, Inc. 1998. Available online at http://www.cryptography.com/dpa/technical/
13. Kocher, P. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. 1995. Available online at http://www.cryptography.com/timingattack/
14. Kuhn, M., Anderson, R. *Tamper Resistance - a Cautionary Note*. Proceedings of Second USENIX Workshop on Electronic Commerce, Oakland, California. pp 1-11. 1996. Available online at http://www.cl.cam.ac.uk/∼mgk25/tamper.html
15. López, J.; Maña, A; Pimentel, P.*Un Esquema Eficiente de Protección de Software Basado en Tarjetas Inteligentes*. Technical Report 14/2000, Department of Computer Science, University of Malaga. 2000.
16. López, J., Maña, A., Pimentel, E., Troya, J.M., Yagüe, M. I. *Access Control Infrastructure for Digital Objects*. Proceedings of 4th. International Conference On Information and Communications Security (ICICS'02). LNCS 2513. Springer-Verlag. 2002.
17. Loureiro, S.; Molva, R. *Function hiding based on error correcting codes*. Proceedings of Cyptec'99 - International Workshop on Cryptographic techniques and Electronic Commerce. 1999.
18. Maña, A. *Protección de Software Basada en tarjetas Inteligentes*. (in spanish). PhD dissertation. Computer Science Department, University of Malaga. 2003.
19. Maña, A., Pimentel, E. *An Efficient Software Protection Scheme*. Proceedings of IFIP SEC'01. Kluwer Academic Publishers. 2001.
20. Petri S. *An Introduction to Smart Cards*. Litronic, Inc. 2001. Available online at http://www.litronic.com/solutions/whitepapers/introduction_to_smartcards/
21. Samuelson, P. *A Manifesto Concerning the Legal Protection of Computer Programs: Why Existing Laws Fail To Provide Adequate Protection*. Proceedings of KnowRight '95, pp 105-115. 1995.
22. Sander, T.; Tschudin C.F. *On Software Protection via Function Hiding*. Proceedings of Information Hiding '98. Springer-Verlag. LNCS 1525. pp 111-123. 1998.
23. Schaumüller-Bichl1, I.; Piller, E. *A Method of Software Protection Based on the Use of Smart Cards and Cryptographic Techniques*. Proceedings of Eurocrypt'84. Springer-Verlag. LNCS 0209, pp. 446-454. 1984.
24. Shamir, A. *Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies*. CHES 2000, Springer-Verlag, pp. 71-77. 2000.
25. Stern, J. P., Hachez, G., Koeune, F., Quisquater, J. J. *Robust Object Watermarking: Application to Code*. In Proceedings of Info Hiding '99, Springer-Verlag. LNCS 1768, pp. 368-378, 1999. Available online at http://www.dice.ucl.ac.be/crypto/publications/1999/codemark.pdf
26. Sun Microsystems. *Java Card Technology Homepage*. 2003. Available online at http://java.sun.com/products/javacard/
27. Ward, R. *Cryptographic Smart Card Capabilities and Vulnerabilities*. Secure Telecommunications Report ECE 636. 2001. Available online at http://ece.gmu.edu/courses/ECE636/project/reports/RWard.pdf
28. Wayner, P. *Dissapearing Cryptography. Information Hiding, Stenography and Watermarking*. Morgan Kauffman. 2002.
29. White, S., Commerford, L. *ABYSS: An Architecture for Software Protection*. IEEE Transactions on Software Engineering. Vol. 16, Nb. 6. June 1990.
30. Yagüe, M.I., Maña, A., López, J., Pimentel E., Troya, J.M. *A Secure Solution for Commercial Digital Libraries*. Online Information Review Journal. Emerald Publishers. 2003.

## APPENDIX. Pseudo-code of license mangement protocols.

*Card Setup*:

```
PARTIES:
C Client
CC Client Card
CCM Client Card Manufacturer

DEFINITIONS:
CCM≪CC≫ ≡ CCM{CCp,|CredCardData|}

SEQUENCE:
C → CC: Setup(|CredCardData|)
CC: CreateKeyPair()
CC → CCM: CCMp[CCp,|CredCardData|,CCID]
CCM → CC: CCM≪CC≫
```

*Sale*:

```
PARTIES:
C Client
CC Client Card
CCM Client Card Manufacturer
SP Software Producer

DEFINITIONS:
CCM≪SP≫ ≡ CCM{SPp}
LicReq ≡ SPp[CCM≪CC≫,SoftID,Nonce]
License ≡ CCp[LicInfo,Nonce]

SEQUENCE:
CC → C: CertReq()
C → SP: CertReq()
SP → C: CCM≪SP≫
C → CC: CCM≪SP≫
CC → C: LicReq
C → SP: LicReq
SP: Store(CCM≪CC≫,SoftID,LicInfo)
SP → C: License
C → CC: License
CC: DeleteReq
```

*Transfer*:

```
PARTIES:
SC Source Card
SCM Source Card Manufacturer
SCU Source Card User
DC Destination Card
DCM Destination Card Manufacturer
DCU Destination Card User
SP Software Producer

DEFINITIONS:
DelCert(SC,DC)≡ SC{OldLicInfo,DCp}
NewLicReq ≡ SPp[DCM≪DC≫,DelCert(SC,DC),Nonce]
License ≡ DCp[NewLicInfo,Nonce]

SEQUENCE:
DC → DCU: DCM≪DC≫
DCU → SCU: DCM≪DC≫
SCU → SC: Delegate(DCM≪DC≫,OldLicInfo)
SC → SCU: DelCert(SC,DC)
SCU → DCU: DelCert(SC,DC)
DCU → DC: DelCert(SC,DC)
DC → DCU: NewLicReq
DCU → SP: NewLicReq
SP: License]
SP: Delete(SCM≪SC≫,SoftID,OldLicInfo)
SP: Store(DCM≪DC≫,SoftID,NewLicInfo)
SP → DCU: License
DCU → DC: License
```