Trusted Code Execution in JavaCard*

Antonio Maña and Antonio Muñoz

Computer Science Department
University of Málaga. Spain
{amg,amunoz}@lcc.uma.es

Abstract. Some important problems in information security such as software protection, watermarking and obfuscation have been proved to be impossible to solve with software-based solutions. By protecting certain actions in order to guarantee that they are executed as desired, trivial solutions to those problems can be implemented. For tamperproof hardware devices such as smart cards to serve this purpose they must provide the capability to execute code on-the-fly. This paper presents mechanism to allow dynamic code execution in Java Card in order for these cards to be used in software protection problems. However, the solution can be used in other applications.

1 Introduction

There are important problems in information security that have been proved to be impossible to solve with software-based solutions. Among these, we find problems such as software protection, watermarking, obfuscation, production of digital signatures, etc. [1-5]. For other problems, such as auditability, anonymity, or fair exchange, existing cryptographic solutions are very complex and inefficient. However, those problems would have trivial and fast solutions if we could assume that certain actions are protected in such a way that guarantees that they are executed as desired and that the function performed can not be determined.

Therefore, for all these problems, solutions must be based in the use of a "trusted element" that can perform the protected actions [6]. Tamperproof hardware devices and external entities (usually known as trusted third parties) are the most frequent "trusted elements". Although required, the use of trusted elements in the solution is not sufficient to guarantee a good solution. In the optimal solution the amount of trust in these elements must be minimal, while the amount of protection is maximal. The problem is that, usually, these two criteria are conflicting and we need to find a balance among them.

In order to increase the level of trust, it is possible to obtain independent certifications of the behaviour of trusted hardware, especially in the case of simpler hardware. This certification is not possible in the case of trusted third parties (TTPs), which therefore require a higher amount of trust and are not able to provide high

* Work partially supported by E.U. through projects SERENITY (IST-027587) and GREDIA (IST-034363) and by Junta de Castilla la Mancha through MISTICO-MECHANICS project (PBC06-0082).

C. Lambrinoudakis, G. Pernul, A M. Tjoa (Eds.): TrustBus 2007, LNCS 4657, pp. 269–279, 2007.

levels of protection. Consequently, we support the use of tamperproof hardware elements for this purpose. Additional reasons for this claim are:

- TTPs are intrinsically not multipurpose (there are specific TTPs for specific problems). Therefore every user has to deal with multiple TTPs, which complicates the trust and certification schemes.
- Many environments require that the trusted element performs certain actions in representation of the user. TTPs require more trust because they could potentially use the knowledge of these actions and the data involved for illicit purposes.
- TTPs do not provide better protection than tamperproof hardware because, after all, they must use computing systems, which are usually easier to attack.
- TTPs require an online connection that introduces performance and availability problems. For many applications this requirement is a serious inconvenient. TTPs cannot be used in offline applications.

Among other tamperproof hardware elements, we claim that secure general coprocessors, such as PCI coprocessors (for instance IBM's 4758 PCI Cryptographic Coprocessor and other similar products from nCipher, Baltimore Technologies, etc.) or smart cards, which are able to collaborate with the standard unprotected processor are necessary [7]. Simpler hardware elements, such as protected memory tokens, or fixed-functionality processors are not able to fulfil our requirements. In order to guarantee that certain software elements are protected as previously defined, the secure coprocessors used must be capable of receiving the protected code and executing it on-the-fly, a feature that we call dynamic code execution.

2 Background and Related Work

Several mechanisms for secure code execution, and their properties have been proposed in the literature. A classification of these approaches is included below discussing their deficiencies. From this discussion we conclude that a trusted element is needed to achieve a secure code execution.

2.1 Different Mechanisms Related to Secure Execution of Code

A classification of the different approaches to the software protection problem is presented in this section. We focus on security, convenience and practical applicability, more extensive reviews of the state of the art in software protection can be found in [7-8].

Some protection mechanisms are oriented to the protection of the computer system against malicious software. Among these SandBoxing is a popular technique to create a secure execution environment which should be used to run non trusted codes. A sandbox is a container that limits, or reduces, the level of access its applications have, and controls the interaction between them. SandBoxing has been an important technique in research since a long time: Butler Lampson in his paper entitled "Protection", back in 1971 defined the antecedents of the SandBoxing technique.

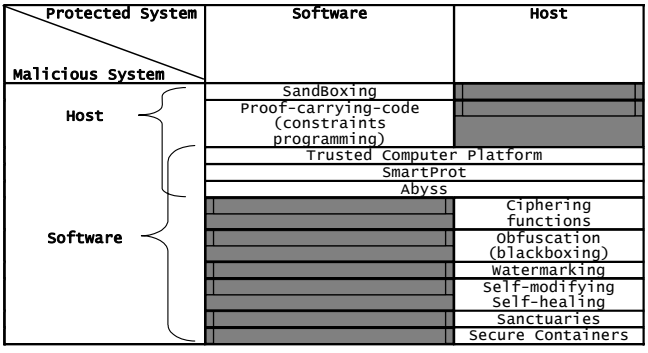

Fig. 1. Classification of the different approaches regarding malicious-protected systems

Proof Carrying Code refers to a general mechanism to verify that part of code can be executed in a host system in a secure way. This strategy requires that every code fragment is associated to a detailed description on how security policy is satisfied, the host just has to verify that the description is correct and the code fits properly. This strategy shares some similarities with Constraints Programming. Both are based on a code that is able to perform a limited set of operations. Furthermore both Proof Carrying Code and Constraints Programming have several problems mainly caused due to the faculty of determining the set of operations permitted. Also in many cases it is difficult to determine restrictions that preserve the semantic integrity. There are Variants of this strategy, such as Proof-referencing-code, do not carry code proofs explicitly [9].

Other mechanisms are oriented to defend software against malicious servers. A example of such mechanism is the concept of Sanctuary [10] for agents. A Sanctuary is a site where a mobile agent can be securely executed. An important precedent, although not directly related with software are Cryptolopes [11-12], developed by IBM. A Cryptolope is a container that includes all management information needed, such as terms and access condition, digital signatures to protect the authenticity, etc. A Similar alternative is Digibox, later named Rights System Platform [13] both of them developed by Intertust, based on contents which are protected even when resold within this system. This platform is designed for high-value digital contents. Due to this, they are not suitable for content commerce environments such as newspaper or information portals for which the value of each element is much reduced. Furthermore this platform only is able to support three different formats (PDF for texts, MPEG for video and MusicMatch for audio).

Several techniques can be applied to software in order to verify self-integrity. Anti-tamper techniques, such as encryption, cryptographic checksumming, anti-debugging, anti-emulation and some others [5] are in this category. Some schemes are based on self-modifying code, and code obfuscation [15]. A related approach is represented by software watermarking techniques [8]. In this case the purpose of 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 [1]. All of these techniques provide short-term protection; therefore, they are not applicable for our purposes.

Many protection systems are based on checks. In these systems the software includes "checks" to test whether certain conditions are met. We can distinguish solutions based exclusively on software, and others that require some hardware component. However, in both types of schemes, the validation function is included into software. Therefore, reverse engineering and other techniques can be used to discover it. Theoretic approaches to the formalization of the problem have demonstrated that a solution exclusively based on software is unfeasible [15]. By extension, all autonomous protection techniques are also insecure.

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 [16] the authors present a scheme that allows evaluation of encrypted functions. The fundamental idea is to establish an homomorphism between the spaces of plaintext and encrypted data, with the objective of evaluating a certain function on some data without revealing them. 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 can be appropriate for distributed computing architectures such as agent-based systems or grid computing, but presents the important disadvantage of the impossibility of application to off-line environments.

Finally there are techniques that establish a two-way protection, such as the Trusted Computer Platform, with recent appearance of ubiquitous computing has raised a needed of a secure platform. Therefore this approach consists of a platform with a trusted component, frequently built-in hardware, which is used to create a foundation of trust for software processes [17]. Another alternative is the ABYSS architecture [18]. Some processes of the software to be protected are substituted by functionally equivalent processes in this system, which runs inside a secure coprocessor. Processes are encrypted while outside of the secure coprocessor. Additionally, the SmartProt mechanism is based on the division of an application's code between a trusted and an untrusted processor in such a way that is not possible to run the application without the collaboration of the trusted processor [19].

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 requirement was very difficult to fulfil, 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 cryptographic keys or biometric profiles) and performing computations required in digital signatures and other cryptographic primitives.

Programmable smart cards, such as Java Card, 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 standard to provide a USB interface in addition to the ISO7816.

Although the amount of memory, computing power, communication speed and physical protection of devices such as PCI coprocessors is higher, smart cards offer several advantages over PCI coprocessors. The most important are:

- Smart cards are portable and can be kept under control of the owner.
- Smart cards circuits are simpler and this facilitates the analysis of possible weak points or hidden traps.
- The level of standardization of smart cards is much higher making them much more interoperable. There are open operating systems for smart cards.
- Smart cards are cheaper.
- Smart cards are multipurpose and can be used in different devices.

Regarding the performance, 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. As an example of the power of current smart card designs, the ST22 family of processors from ST Microelectronics has 32 bits RISC CPUs, 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.

2.3 Hard Security Properties

Nowadays in systems development is relevant to take into account some security properties desirable to be reached. These security properties are hard to get mainly due to the lack of suitable and efficient solutions instead of which we have only partial solutions to these problems. Moreover, in most cases those solutions are difficult to be applied and can be used under restricted conditions or in concrete environments. Finally, it is very usual that solutions to these problems involve the necessity of any kind of trusted device such as Trusted Third Parties (TTPs). These properties we named Hard Security Properties.

These properties are very relevant in order to solve some problems that we find in today applications. Especially relevance we find in web applications and e-commerce applications related. Some properties that we can mention are non-repudiation, fair exchange and secure payment.

We highlight the fair exchange case, because no satisfactory solution exists to solve this problem. Another important issue to note is that almost all existing fair exchange systems base their security in additional security elements such as Trusted Third Parties (TTPs). Furthermore, it is important to mention that today fair exchange solutions put too much confidence in the TTPs.

Similarly in Secure Payment systems, there are a big number of different solutions which are incompatible in most cases. This fact goes in the opposite sense to reach a standardisation in order to consolidate businesses electronic commerce based.

Non-repudiation problem happen some similar fact. In general security solutions applied by different vendors are: each vendor uses a different software-hardware platform. This heterogeneity of platforms is suitable to be reduced dramatically if we arrange of a common platform to perform all security related operations. Obtaining these properties can be easily achieved in case we have a secure element to perform dynamic code execution on-the-fly as part of the runtime environment.

In this paper we highlight the advantages of including on-the-fly code execution capabilities in the Javacard Virtual Machine (JCVM). This new capability of the JCVM enables the development of simple solutions for the aforementioned problems.

3 Dynamic Code Execution in JavaCard

The basic idea of the trusted code execution scheme is that some sections of the software to be protected can be substituted by functionally equivalent sections to be processed in the secure device. In this way, the protected software is divided and will not work unless it cooperates with the appropriate device. Code modification attacks will not succeed in this case and the only possible attack is to analyze the data transmitted to and from the device trying to guess the functions that it performs. By including a large enough number of functions, with enough importance in the main code, and enough complexity, the attack described can become impractical.

Consequently, for the trusted code execution mechanism to be used in different applications and for different purposes, allowing the implementation of simple and secure solutions to the aforementioned problems, the secure device must support dynamic (i.e. on-the-fly) code execution. The dynamic code execution mechanism is inspired on standard Java applets. An applet is a program written in Java that can be included in a web page. When you use a Java technology-enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed on-the-fly by the browser's Java Virtual Machine (JVM).

In the previous discussion we have highlighted the advantages of smart cards versus other secure hardware devices. The popularization of smart cards and their evolution in storage and processing capacity have lead us to consider them the most appropriate choice for the implementation of our scheme. However, our design does not depend on this technology and, consequently, our solution can be implemented using any similar hardware token (for example, some hardware keys and some tokens that integrates smart card and reader functionalities).

Among the different technologies currently available on the smart card market, Java Card represents one of the best alternatives for building a prototype of our

dynamic code execution scheme because it provides features that are useful for this purpose. Java Card standard defines mechanisms for dynamic code loading, but as we will show later, these mechanisms are not enough to support trusted code execution. Other limitations of the current Java Card specification have important impact in our implementation. The most significant are: the lack of file management capabilities, the lack of flexibility in the code loading mechanisms, the lack of code authentication mechanisms; and the lack of dynamic memory management.

Because the basic idea is to execute part of the application code in the smart card, some additional objectives for the code execution mechanism can be established:

- Secure coprocessors (smart cards in this case) must be identified by the protected software. Mechanisms must exist for the code to authenticate the secure coprocessor in order to identify it as a trusted coprocessor.
- Protected software sections must be identified by the secure coprocessor. Mechanisms must exist for the secure coprocessor to authenticate the protected code sections in order to identify them. This is necessary in order to avoid a type of "Trojan horse" attack based on the substitution of some of the authentic protected sections by other fake sections produced by a malicious user. For instance, such a false section could try to extract data of the protected application stored in the card.
- Code must be encrypted while outside of the secure coprocessor. This, in turn, means that the coprocessor must be capable of decrypting the code on-the-fly before execution.
- It must be possible to execute several protected applications at the same time. This does not mean that the coprocessor must support multitasking, but it requires the coprocessor to keep separate memory spaces for each application.
- Enforcement of actions, such as payment, associated to the execution of the protected software must be possible.

3.1 Structure of the Virtual Machine and Execution Environment

The current implementation of the dynamic code execution mechanism is based on a single Java Card applet. Fig. 2 shows the conceptual structure of this applet and highlights the fact that these components are built on top of the Java Card virtual machine. However, applications at this level have limitations (for example, they are isolated by firewalls) and performance constraints.

Therefore, our aim is to propose the implementation of those functions at a lower level in order to obtain better performance and to facilitate the deployment of other applications that can take advantage of the dynamic code execution infrastructure.

The dynamic part of this applet represents the memory assigned to the applet during its installation. The amount of memory reserved for the different purposes (code, application data and optional licenses) can be defined when the applet is loaded into the card. Java Card does not support dynamic memory allocation. For this reason the Runtime Manager component allows protected applications to use this memory in a dynamic way.

The static part is permanently loaded in the card. This static part includes the card-specific data (keys, etc.) as well as these components:

A License Manager that performs all operations directly related to licenses, such as installation, transfer, backup, etc. It also implements the operations required by the license management protocols.

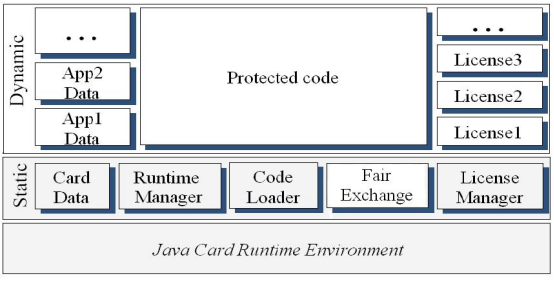

Fig. 2. Logical structure of the card components

A Code Loader that creates the internal representation of the protected code sections when they are downloaded to the card. The Code Loader must then locate the corresponding license and decrypt, translate and create a representation of this section suitable for being executed by the Runtime Manager. The format of this specific code is described in next section. The APDU (Application Protocol Data Unit) used to download the code into the card has three parameters: (i) the license identifier, which is required in order to locate the license and to decrypt the section; (ii) an array of encrypted code; and (iii) an array containing the data required by this section, which can be specified either explicitly or by reference. In the latter case, the actual value of the data is determined at runtime, possibly depending on the results of previously executed sections of the same application.

The most important goal of the Runtime Manager is to overcome the lack of dynamic memory management in Java Card. It is responsible for executing the code, for dynamic allocation/deallocation of memory and for keeping separate memory spaces for different applications. When memory is deallocated it is always overwritten to avoid that other application can try to use it illegitimately. An optional Fair Exchange component, which has been specifically designed to provide a generic fair exchange service that can be used for payment as well as for other purposes.

3.2 Representation of the Dynamic Code Fragments

The lack of mechanisms that allow dynamic code execution in the standard Java Card has forced us to define a specific 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 external storage and transmission to the card. The Code Loader decrypts the protected code using the corresponding license and then translates into an internal format designed to

overcome 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 of the virtual machine to be self-contained and to achieve an easy referencing between instructions. Together with the aforementioned components, this class constitutes the core of the dynamic code execution mechanism. This representation integrates the interpreter in the Instruction class, which in turn results in greater flexibility because it allows the definition of different interpreters without changing the supporting components.

When loaded into the card the code is converted into an array of Instruction objects. The execution is then 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.

```
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. 3. Summary of Instruction class

The lack of file management in Java Card has been solved by (i) Structuring the code in a way that is easily managed by the standard Java Card (that is the reason why the Protected Code memory is defined as an array of Instruction objects) and (ii) using our own software for the management of the instructions. The aforementioned Code Loader is responsible for this second functionality. When the Code Loader has created the Instruction objects that represents the protected code it calls the execute method of the first object in the array. There are jump and loop instructions that open new execution branches. This process goes on until a final instruction is reached, which finalises the execution of the current branch. The execution of the protected section ends when the main branch ends.

4 Applications

The mechanism that provides trusted code execution in JavaCard has a great impact on the ability to solve difficult security problems as explained before. Furthermore,

applications using this concept provide a further justification of the previous assertions. Some of these applications are briefly explained below.

An interesting application of dynamic code execution based on our JavaCard implementation has been developed to protect and distribute audio files in mp3, wma and CD-audio formats. Our system uses dynamic code execution to decrypt and play protected audio files. Moreover, the protection mechanism enables the free distribution audio files. The JavaCard applet cooperates with Java executed in the player device to perform decryption and reproduction. This applet requires to playing audio file a card specific license to be produced for each audio file. This prevents unauthorized use of the file.

A related work consists of a secure platform for pay-TV through the Internet. Pay-per-view platform uses different technologies such as Java Media Framework, Real Transport Protocol, and Video Streaming to provide a fine-grained timeslot model for video distribution. In this case trusted execution code is used to implement a variant of forward secrecy schemes. The ability to execute code dynamically us the key to the controlled generation of time-limited keys for the player device.

A secure framework for digital newspaper distribution named EC-GATE was developed and obtained the gold award in the e-Gate Open International Contest. This application is based on the following idea: The security requirements of all processes related to the secure transmission and commerce of information can be fulfilled if we guarantee that the software running at the other side of the communication line is protected. This solution is based on the notion of "secure container", a protected package of data and administrative information. Opposed to other proposals we use "active" instead of "passive" containers (we use software instead of data) in order to avoid some problems of the latter. The dynamic code execution capability is used to guarantee that decryption and payment operations are performed inside the smartcards as an atomic operation, therefore providing a fair payment mechanism.

The SmartProt scheme [19] is designed to protect software elements from analysis and to ensure that they are executed as desired by its creator. The system works in different phases starting with a card setup phase. The dynamic code execution mechanism presented in this paper is used in SmartProt in order to prevent code modifications attacks. Main goals of these attacks are to produce an unprotected copy of the protected software.

5 Conclusions

An infrastructure to allow dynamic code execution in Java Card has been presented. We have shown the relevance of this functionality for some important information security problems and have discussed how other approaches are less suitable. The infrastructure presented is based on creating a specific virtual machine on top of Java Card. However, applications at this level have limitations (for example, they are isolated by firewalls) and performance constraints. Therefore, our aim is to propose the implementation of those functions at a lower level in order to obtain better performance and to enable the deployment of other applications that can take advantage of the software protection infrastructure.

References

[1] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (Im)possibility of Obfuscating Programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)

[2] Maña, A., Matamoros, S.: Practical Mobile Digital Signatures. In: Bauknecht, K., Tjoa, A.M., Quirchmayr, G. (eds.) EC-Web 2002. LNCS, vol. 2455, pp. 224–234. Springer, Heidelberg (2002)

[3] Pagnia, H., Gartner, F.C.: On the impossibility of fair exchange without a trusted third party. Darmstadt University of Technology, Department of Computer Science Tech. Rep. TUD-BS-1999-02 (1999)

[4] Spalka, A., Cremers, A.B., Langweg, H.: Protecting the creation of digital signatures with trusted computing platform technology against attacks by Trojan Horse programs. In: Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001), Kluwer Academic Publishers, Dordrecht (2001)

[5] Schaumüller-Bichl1, I., Piller, E.: A Method of Software Protection Based on the Use of Smart Cards and Cryptographic Techniques. In: Beth, T., Cot, N., Ingemarsson, I. (eds.) Advances in Cryptology. LNCS, vol. 209, pp. 446–454. Springer, Heidelberg (1985)

[6] Herzberg, A., Pinter, S.S.: Public Protection of Software. ACM Transactions on Computer Systems 5(4)-87, 371–393 (1987)

[7] Maña, A.: Protección de Software Basada en Tarjetas Inteligentes. PhD Thesis. University of Málaga (2003)

[8] 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) http://www.dice.ucl.ac.be/~hachez/thesis_gael_hachez.pdf

[9] Gunter Carl, A., Peter, H., Scott, N.: Infrastructure for Proof-Referencing Code. In: Proceedings, Workshop on Foundations of Secure Mobile Code (March 1997)

[10] Yee, B.S.: A Sanctuary for Mobile Agents. Secure Internet Programming (1999)

[11] Cryptolope link. http://ei.cs.vt.edu/~wwwbtb/book/chap8/sect2/cryptolope.html

[12] Garcia-Molina, H., Ketchpel, S., Shivakumar, N.: Safeguarding and Charging for Information on the Internet. In: Proceedings of the Intl. Conf. on Data Engineering (1998)

[13] Intertrust Technologies. Intertrust Technologies Home Page, http://www.intertrust.com/

[14] Collberg, C., Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. University of Auckland Technical Report #170 (2000) http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborson2000a/index.html

[15] Goldreich, O.: Towards a theory of software protection. In: Proc. 19th Ann. ACM Symp. on Theory of Computing, pp. 182–194. ACM Press, New York (1987)

[16] Sander, T., Tschudin, C.F.: On Software Protection via Function Hiding. In: Aucsmith, D. (ed.) IH 1998. LNCS, vol. 1525, pp. 111–123. Springer, Heidelberg (1998)

[17] Pearson, S., Balacheff, B., Chen, L., Plaquin, D., Proudler, G.: Trusted Computer Platforms. Prentice Hall PTR 2003, Englewood Cliffs (2003)

[18] White, S., Commerford, L.: ABYSS: An Architecture for Software Protection. IEEE Transactions on Software Engineering 16(6) (1990)

[19] Maña, A., López, J., Ortega, J.J., Pimentel, E., Troya, J.M.: A Framework for Secure Execution of Software. International Journal of Information Security (to appear, 2004)