Institut Eurécom

Thèse

présentée pour obtenir le grade de docteur

de l'Ecole Nationale Supérieure

des Télécommunications

Spécialité : **Informatique et Réseaux**

par: **Sergio Loureiro**

Sujet de la thèse:
Mobile Code Protection

Mobile Code Protection

Sergio Loureiro

ENST Paris / Institut Eurecom

Contents

Contents

Acknowledgements

First of all, I must acknowledge my great debt to my advisor Refik Molva for his valuable advice and pragmatic view of research. I am very grateful for his patience in improving my writing and presentation skills.

I would like to thank my thesis committee Nicolas Sendrier, Christian Tschudin, Andrzej Duda, Günter Karjoth and Patrick Solé for their valuable feedback. I am especially grateful to Nicolas Sendrier for his encouragement and for helping me with coding theory.

I want to express my gratitude to Eurecom, which provided a pleasant work environment.

I would like to thank the NSTeam at Eurecom, namely Alain Pannetrat and Yves Roudier for the fruitful discussions we had and for their patience reading my thesis.

Abstract

The definition of the mobile code paradigm appears as a natural step in the evolution of distributed systems and encompasses programs that can be executed on one or several hosts, other than the host from which they have originated. Mobile code is generally justified on the grounds of greater efficiency and increased flexibility, even if these features have not been fully exploited yet. However, flexibility does not come without a price: increased exposure to security threats.

Possible vulnerabilities with mobile code fall in one of two categories:

- Attacks performed by the mobile program against the remote execution environment and its resources;
- Subversion of the mobile code and unauthorized modification of its data by the remote execution environment.

Our work focuses on the second category aiming at the protection of mobile code from the execution environment. This category results in new and challenging problems which have not yet attracted much attention from software manufacturers and for which no practical solutions exist at this moment. This category is also quite atypical since it does not rely on the security of the execution environment which has always been a basic

assumption in classical reasoning about the security of cryptographic systems. We further analyze mobile code protection in two directions: code protection focusing on the integrity and privacy of the code semantics at run-time and data protection focusing on the security of the data transported by the mobile code.

Code protection addresses a more systematic form of maliciousness in which the environment where the mobile code runs cannot be trusted. Code protection means the protection of the code during its execution, considering the environment as a potential adversary, rather than the protection of the code during transmission.

Data protection deals with the security of data gathered by mobile code roaming through a set of competing hosts. Classical data protection techniques are not suited to the protection of data that changes dynamically during the code's trip. We present a protocol based on a cryptographic technique that assures the integrity of a sequence of data segments regardless of the order of each segment in the sequence. The protocol allows each host to update the data it previously submitted, in a way that is suitable for free competition scenarios like comparative shopping or distributed auction, and for highly dynamic environments like stock markets. The set of hosts can be visited several times in random order and a short message digest allows for the integrity verification of all the collected data.

Concerning code protection, we further classify the problems into two categories: privacy of execution and integrity of execution. Privacy of execution aims at preventing the disclosure of the code semantics during its execution in a potentially hostile runtime environment. Integrity of execution assures that a program, executed in a potentially hostile environment actually complies with its original semantics. We present original solutions that deal with both requirements (privacy and integrity of execution). We present solutions without Tamper Proof Hardware (TPH) that address a very limited model of computation.

Then, we build solutions using an auxiliary trusted TPH acting on behalf of the code owner. The limited TPH allows us to deal with a more flexible model of computation. The trusted TPH interacts with the untrusted execution environment in order to fulfill the security requirements akin to privacy

and integrity of execution. The goal is not to execute the code on the trusted TPH, but to extend its inherent security to the more powerful untrusted environment. The solutions minimize the computational and storage requirements on the trusted TPH.

The solution with TPH assumed that the data involved in the computations is stored in secure memory. We further enhance this solution by focusing on the protection of the data stored in untrusted memory. Based on the solutions for the protection of the code execution and the solutions for the storage of data in untrusted memories, we suggest an integrated architecture for code and data protection that relies on a limited TPH.

Abstract

Résumé

Le concept de code mobile apparaît comme une étape naturelle dans l'évolution des systèmes répartis : il correspond à l'ensemble des programmes qui ont la capacité de se déplacer pendant l'exécution ou entre différentes exécutions. L'utilisation de code mobile est généralement justifiée par une plus grande efficacité, ainsi que par une flexibilité accrue, même si ces avantages n'ont pas encore été entièrement exploités. Cependant, la flexibilité s'accompagne d'une exposition accrue aux menaces de sécurité.

Les vulnérabilités possibles du code mobile peuvent être classées en deux catégories :

- Les attaques perpétrées par le code mobile contre l'environnement d'exécution et ses ressources;
- La subversion du code mobile et la modification non autorisée de ses données par l'environnement d'exécution.

Notre travail porte sur la protection du code mobile vis-à-vis de l'environnement d'exécution. La plupart des problèmes de ce type sont nouveaux et représentent un défi ; ils n'ont pas encore attiré l'attention des éditeurs de logiciels et aucune solution pratique à ces problèmes n'existe à

ce jour. Cette catégorie est également tout à fait atypique puisqu'elle ne se fonde pas sur la sécurité de l'environnement d'exécution qui a toujours été une condition de base dans le raisonnement classique au sujet de la sécurité des systèmes cryptographiques. Nous analysons la protection du code mobile à deux niveaux : la protection du code visant à assurer l'intégrité et la confidentialité du code pendant l'exécution et la protection des données afin d'assurer la sécurité des données contenues dans un code mobile.

La protection du code a pour objet une forme plus systématique de malveillance dans laquelle on ne peut pas faire confiance à l' environnement où le code mobile s'exécute. La protection du code s'effectue pendant l'exécution, en considérant l'environnement en tant qu'adversaire potentiel.

La protection des données traite de la sécurité des données recueillies par le code mobile qui se déplace entre un ensemble de serveurs concurrents. Les techniques habituelles de protection de données ne conviennent à la protection des données qui évoluent dynamiquement pendant le trajet du code. Nous présentons un protocole basé sur une technique cryptographique qui assure l'intégrité d'un ensemble de segments de données indépendamment de l'ordre de chaque segment dans l'ensemble. Le protocole permet à chaque serveur de mettre à jour les données qu'il a précédemment soumises. Ce protocole répond aux besoins des scénarios de libre concurrence comme les achats comparatifs ou l'enchère distribuée et il convient particulièrement aux besoins des environnements fortement dynamiques comme les marchés boursiers. Les serveurs peuvent être visités plusieurs fois de façon aléatoire et l'intégrité de toutes les données collectées peut être vérifée par le calcul d'un condensat.

Pour ce qui concerne la protection de code, nous classifions ces problèmes dans deux catégories : confidentialité de l'exécution et intégrité de l'exécution. La confidentialité de l'exécution vise à empêcher la révélation de la sémantique du code pendant son exécution dans un environnement d'exécution potentiellement hostile. L'intégrité de l'exécution assure qu'un programme est exécuté dans un environnement potentiellement hostile conformément à sa sémantique initiale.

Pour chacun de ces deux problèmes , nous présentons d'abord une famille de solutions sans noyau sécurisé (NS) concernant un modèle de calcul très

limité. Puis, nous montrons des solutions utilisant un NS auxiliaire agissant au nom du propriétaire du code. Le NS nous permet d'utiliser un modèle de calcul plus flexible. Le NS communique avec l'environnement d'exécution afin de satisfaire les besoins de sécurité, c'est-à-dire la confidentialité et l'intégrité de l'exécution. L'objectif n'est pas d'exécuter le code sur le NS mais d'étendre la sécurité inhérente de ce dernier à un environnement plus puissant. Nos solutions réduisent la complexité en calcul et en mémoire imposée au NS.

Notre solution avec NS suppose que les données impliquées dans les calculs sont sauvegardées dans une mémoire sécurisée. Nous étendons cette solution en nous concentrant sur la protection des données sauvegardées dans une mémoire potentiellement malveillante. Grâce à l'utilisation des solutions pour la protection de l'exécution du code et des solutions pour la sauvegarde des données dans des mémoires potentiellement malveillantes, nous mettons au point un architecture pour la protection de code et de données qui se base sur un NS limité.

Introduction

1.1 The Mobile Code Paradigm

The mobile code paradigm encompasses programs that can be executed on one or several hosts, other than the host from which they have originated. Mobility of such programs implies a built-in capability for each piece of code to migrate smoothly from one host to another. A mobile code is associated with at least two parties: its owner, and the host that runs the code. Advances in software technology allow for the design of mobile code capable of moving over the network, and running independently at remote hosts. Mobile code systems range from simple Java applets and ActiveX programs to intelligent mobile software agents. Recently, we witnessed the deployment of a large number of mobile code platforms (for example, see the short survey in [KT98]), which provide the required services for code mobility.

Depending on the type of mobility, a mobile code can be further classified into strong and weak mobility, as described in [FPV98] and [Pic98].

Our work is orthogonal to the discussions about the taxonomy of mobile code and different types of mobility, in the sense that we focus on the general case of the security of a code executed on an untrusted host. As a result,

the fact that the code encompasses some form of intelligence or that it can move anytime in a completely autonomous way is not relevant for our study. Therefore, we adopt the more general denomination of mobile code rather than mobile agents. However, the term agent is also used in this manuscript when appropriate.

This chapter presents a brief introduction to mobile code focusing on security threats to better identify the scope of this dissertation. Next, a simple classification of the problems that affect mobile code is established. Finally, the structure of the thesis presented at the end of this chapter is clarified.

1.1.1 Benefits and Drawbacks

Mobile code systems offer several advantages over the more traditional distributed computing approaches [LO99][CHK97] in terms of:

- Network load reduction;
- Network latency avoidance;
- Protocols encapsulation;
- Asynchronous and autonomous execution;
- Dynamic adaptation;
- Support for heterogeneous architectures;
- Robustness and fault-tolerance.

All these advantages need to be critically analyzed. The advantages are mainly due to the fact that the execution is performed locally, close to the data to be analyzed or near the system outputs to be processed. For example, [HI99] shows through an experimental study that bandwidth optimization can be achieved using mobile agents, although the experiments focused on a very specific application and scenario, hence the results should be considered with care. It was particularly clear from the experiments that the overhead on size, and thus on communication and computational complexity imposed by the mobility platform, may compromise the bandwidth advantage.

Nevertheless, we believe that despite controversial arguments about specific criteria, mobile code offers a clear advantage over traditional programming paradigms in terms of flexibility in the development of applications.

1.1.2 Applications

Some applications have been cited in [LO99] as very suitable for mobile agents or mobile code as follows:

- *Electronic commerce*. There are already a large number of mobile agent mediated electronic commerce frameworks [GMM98]. Researchers [MGM99] envision agents embodying the intentions of their owners, acting and negotiating on their behalf while travelling through the network;

- *Personal assistants*. Assistants can operate remotely without being dependent on the state of network connections [Kru97];

- *Distributed information retrieval*. Mobile code that roams the network to gather data is one of the simplest applications envisaged by developers. The ability of mobile code to process searches locally on large databases is very attractive. This specific application will be the focus of Chapter 8;

- *Monitoring applications*. The asynchronous nature of mobile code is highlighted by this application. The code can be dispatched to monitor sources of information available remotely, avoiding network latencies and the need for a reliable connection;

- *Information and code dissemination*. Mobile code can provide additional content delivery services;

- *Parallel processing*. Complex computations can be performed by a set of mobile code fragments that will execute in parallel on different hosts.

This list is not exhaustive, but it gives clues to the scenarios where mobile code can be applied.

1.2 Mobile Code Security Threats

The increased flexibility offered by mobile code comes at the expense of increased vulnerability in the face of malicious intrusion scenarios akin to networking. The following security exposures derive from code mobility:

- Host and mobile code can represent different parties that may exhibit malicious behavior toward one another;
- Mobile code can be exposed to third-party intruders through the network;
- Several mobile code segments representing different parties may exhibit malicious behavior toward one another.

The second and third types of exposures call for solutions based on classical communication security mechanisms, such as the approach suggested in [KLO97]. However, the first type of exposure raises new requirements that cannot be met by classical security techniques. As for the first problem, the type of exposures falls into one of two categories:

- *Host protection from mobile code.* Attacks performed by the mobile program against the remote execution environment and its resources;
- *Mobile code protection from malicious hosts.* Subversion of the mobile code and unauthorized modification of its data by the remote execution environment.

1.2.1 Host Protection from Mobile Code

A first security threat consists of a mobile code generated by a malicious outsider attacking the environment where the code is executed, such as in the example of malicious Java applets. This problem has already been thoroughly studied in recent years. The first solution consisted in restricting capabilities of code segments in order to limit vulnerabilities. Techniques for host protection now evolve along two directions [LMR00]:

- Enhancement of the mobile code infrastructure with authentication, data integrity and access control mechanisms;
- Verification of the semantics of the mobile code.

In the following, we give a brief description of the best known solutions to the problem of host protection in order to show the difference in complexity between host and mobile code protection. The surveys [HLPS98],[RG98],[LMR00] and references indicated below provide more information about this topic.

Sandboxing

Sandboxing consists of running a mobile code inside a restricted environment called the "sandbox". A remote host may execute an otherwise untrusted mobile code inside the sandbox, without worrying about security. This approach is well illustrated by the early Java JDK 1.0 [GJS96], where it was used in order to enable applets available anywhere on the Internet to run within a browser. The major drawback of sandboxing is that applications running in such a restrictive environment are themselves seldom useful because their operations are limited.

Code Signing

Code signing is the process through which a code is digitally signed by the code owner in order to assure strong authentication and integrity of the code to whomever executes the code. This model was first introduced by Microsoft within the ActiveX framework. Java JDK 1.1 also follows the code signing model, with so-called signed applets. Upon receipt of an applet with a valid signature, the Java virtual machine executes the applet as a trusted piece of code, authorizing it to access all features available in Java. An applet without a proper signature is run inside a sandbox as in the previous version of the JDK. However, securing a host from a malicious mobile code program raises more security issues than just making sure that this program has been correctly signed by someone on the Internet.

Access Control

In order to limit the impact of an attack, one way to enhance the previous approaches is to enable more complex access control schemes. This can be seen as the refinement of a monolithic sandbox policy into smaller, applica-

tion-specific policies. The identity of the code signer, as described in the previous section, also helps to further refine the execution policy definition. Java JDK 1.2 security model [Gon98] follows this direction and thus allows the definition of finer-grained security policies more suited to executing untrusted mobile code. Compared with sandboxing and code signing, the access control model has the best of both worlds: mobile code actions can be restricted to a set of resources while the model allows to write and run really useful software at the same time. However, since it is performed dynamically at runtime the enforcement of the access scheme has a cost in performance.

Code Verification

Code verification provides further assurance on the code semantics through the analysis of the structure or behavior of the mobile code against a given security policy. Sandboxes have already exercised some rudimentary program checks, either statically or dynamically, for instance to ensure that operands of an instruction are of the correct type. A newer approach to host protection is to statically type-check the mobile code; the code is then run without any expensive runtime checks. Promising results were obtained in this area by the Proof-Carrying Code (PCC) work [Nec97] [NL98], and even to some extent by the Java virtual machine (for safety checks). In Proof-Carrying Code, the remote host first asks for proof that the code respects his security policy before he actually agrees to run it. The code owner sends the program and an accompanying proof, using a set of axioms and rewriting rules. After receiving the code, the host can then check the program with the guidance of the proof. This can be seen as a form of type checking of the program, since the proof is directly derived from it. In PCC, checking the proof is relatively simple compared to constructing it, thus this technique does not impose much computational burden on the execution environment. However, the proofs can be large and automating the proof generation is still an open problem.

1.2.2 Mobile Code Protection from Malicious Hosts

The problem of mobile code protection from a malicious host has only recently been studied, and it is intrinsically more difficult because the run-

time environment has total control over the mobile code. This problem has not yet attracted much attention from software manufacturers. Research is still in its infancy as well, and the existing mobile code platforms only provide solutions for host protection.

This category is also quite atypical as it does not rely on the security of the execution environment which has always been a basic assumption in classical reasoning for the security of cryptographic systems. This assumption is often denoted by the term "trusted computing base". Protecting mobile code from malicious remote execution environments may be rephrased as carrying out a trusted operation (defined by the mobile code), using an untrusted host as execution environment. Some authors [CGH+95] postulate that mobile code cannot be effectively protected against a malicious execution environment that has full access to both code and data segments.

The problems encountered by the execution of mobile code on untrusted and possibly malicious hosts are the focus of this dissertation. In the case of mobile code protection, the existing surveys [JK99][KP00] are limited to the description of the few approaches proposed to address this problem. A deeper analysis of the security issues and their solutions is needed to gain an insight into the problem of mobile code protection. Concretely, we start by classifying the security requirements according to the information carried by the mobile code. This information can be classified as follows:

- *Code*. The set of executable instructions;
- *Static data*. Data not modified during the trip;
- *Collected data*. Data collected during the trip performed by the mobile code;
- *State*. Dynamic data used as input to the computations performed on remote hosts during the trip.

The aim of mobile code protection is to protect the above information from the execution environments visited by the mobile code. Starting from this classification, possible violations originating from running a program in a potentially hostile environment may lead to the following different security requirements:

- A company might need to prevent the disclosure of certain sensitive algorithms implemented in its mobile code despite extensive code analysis and reverse engineering by potential intruders, including its customers;

- A mobile software agent acting on behalf of a person might need to assure the integrity of some critical operations performed on an untrusted remote host;

- A data collection agent might need to assure both the confidentiality and the integrity of the data gathered at various competing sites.

From the above discussion, the difference between requirements related to code execution and the ones related to data protection naturally comes up. Through this thesis, we show that there is a broad class of problems corresponding to each set of requirements. Therefore, we will clearly distinguish the problems of code protection from data protection.

Code Protection

We further classify the problems related to code protection (as in [BMW98]) into two categories, namely privacy of execution and integrity of execution:

- *Privacy of execution* aims at preventing the disclosure of the code semantics during its execution in a potentially hostile runtime environment. This is a hard problem because the disclosure of the code semantics to the runtime environment is considered a basic requirement for code execution;

- *Integrity of execution* assures that a program executed on a potentially hostile environment actually complies with its original semantics.

We analyze these two problems in Chapters 2 and 3, respectively. In Chapters 5 and 6, we present original solutions addressing these problems.

Data Protection

Mobile or roaming agents are a form of mobile code especially talked about in electronic commerce, but usual data protection techniques are not suited

to the protection of data that change dynamically during the code's trip. A good example might be the so-called "comparison shopping" application, where an agent looks for the lowest priced shopping item among several retailers. In this scenario, it is necessary to know whether the data collected has been changed or not. In a distributed auction scenario, it might also be necessary to prevent a host from exploiting the offers made by the previous bidders. These scenarios show that there is a need for data protection schemes outside the sphere of code protection. We focus on data protection in Chapters 7 and 8.

1.3 Contributions

This thesis focuses on problems related to mobile code protection from potentially malicious hosts. Our contributions may be summarized as follows:

- We elaborate an original list of requirements imposed by the problem of privacy of execution in Chapter 2;
- We introduce the concept of verifier in Chapter 3. This original definition is more suitable to tackle the problem of integrity of execution in the mobile code scenario;
- We present an original solution to the problem of code protection in Chapter 5. This solution applies to a simple model of computation (Boolean functions), based on error correcting codes and provides privacy and integrity of execution;
- We extend the previous solution to a more powerful model of computation in Chapter 6. This solution requires a limited Tamper-Proof Hardware, like a smartcard, acting on behalf of the code owner and located near the execution environment. A new set of requirements is presented as well as a solution that applies to sets of Boolean functions;
- We introduce the concept of off-line verifier in Chapter 6. We develop an off-line solution where the verification process is not performed by the TPH, but by the code owner. A small check value computed by the TPH is transmitted to the code owner, assuring the integrity of execution of several computations;

- We present techniques to address the problem of data storage in untrusted memory in Chapter 7;
- We present some guidelines for the development of an architecture for software protection in Chapter 7. This architecture is original in the sense that previous proposals had considered a trusted CPU and untrusted memory, while we extend the security of a limited Tamper-Proof Hardware to the computations performed on an untrusted CPU;
- We develop a new technique to protect the data collected by mobile code roaming through a set of competing hosts in Chapter 8.

1.4 Structure

This thesis focuses on the problem of mobile code protection from possibly malicious hosts as depicted at the top of Figure 1.1. The general problem of mobile code protection from untrusted execution environments is divided into two sub-problems as shown in Figure 1.1:

- Code Protection;
- Data Protection.

Chapter 2 describes the problem of privacy of execution and refers to related work. Privacy of execution is a hard problem that consists in hiding information about the code while giving a description that allows its execution. At the end of the chapter, we specify the requirements imposed by privacy of execution.

Chapter 3 focuses on the problem of integrity of execution and it discusses existing approaches to tackle this problem. Verifiers are defined, which make possible the design of more efficient solutions to the problem of integrity of execution.
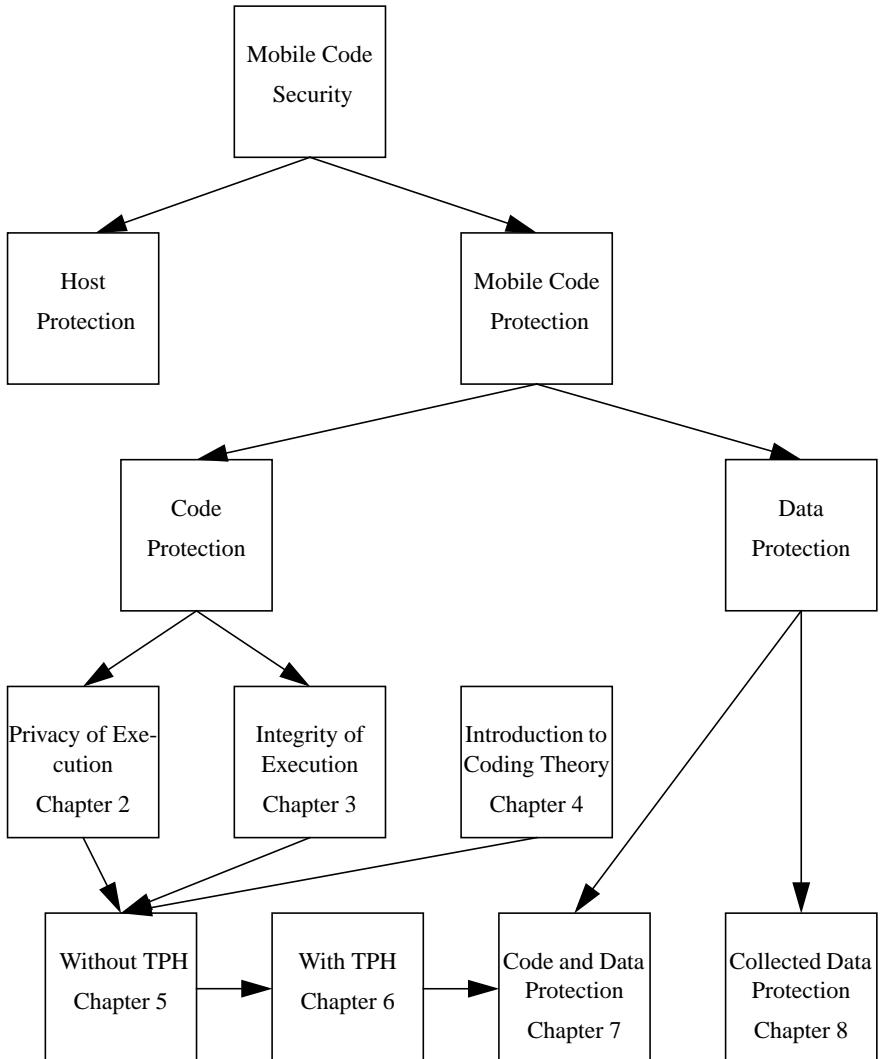
Figure 1.1 Scope and structure

Chapter 4 provides an introduction to the cryptographic tools used in subsequent chapters. These tools exploit the hardness of some problems found in coding theory. This chapter focuses on the complexity of the problems used to construct cryptosystems based on coding theory. At the end, the chapter includes a cryptoanalysis of these systems.

The remaining chapters (last row of blocks of Figure 1.1) present our solutions to the problems stated in the first three chapters. We further classify our solutions to mobile code protection, as:

- Solutions that do not rely on Tamper-Proof Hardware (Chapter 5);
- Solutions that rely on it (Chapter 6).

Chapter 5 presents solutions without Tamper-Proof Hardware (TPH). This chapter defines a framework for mobile code protection. We start by considering a simple example, and we present solutions to general Boolean functions afterwards. A simple model of computation based on Boolean functions is adopted in Chapter 5 to illustrate our solution in practical applications. We extend this model in the subsequent two chapters.

Chapter 6 builds solutions using an auxiliary trusted TPH acting on behalf of the code owner. The trusted TPH interacts with the untrusted execution environment in order to fulfill the security requirements of privacy and integrity of execution. The goal is not to execute the code on the trusted TPH, but to extend its inherent security to the more powerful untrusted environment. The solutions try to minimize the computational and storage complexities imposed on the trusted TPH.

The solutions described in Chapters 5 and 6 focus on the problem of code execution explained in Chapters 2 and 3. These chapters do not address the problem of data protection, specifically the protection of the dynamic data involved in the computations (state).

Chapter 7 focuses on the protection of the data used during the computations. In Chapter 6 we assumed that the limited TPH has enough memory to securely store all the values involved in the computations. In Chapter 7, several solutions are developed to address the protection of data stored in untrusted memory. Using solutions for the protection of code executions

and solutions for the storage of data in untrusted memory, we build an architecture for software protection that relies on a limited TPH.

Chapter 8 addresses the problem of collected data protection. This chapter describes a protocol to protect data collected by mobile agents roaming through a set of potentially malicious hosts. This protocol is based on a cryptographic technique that assures the integrity of a sequence of data segments, regardless of the order of each segment in the sequence. The protocol allows each host to update the data it previously submitted in a way that is suitable for free competition scenarios such as comparative shopping or distributed auction and for highly dynamic environments such as stock markets. The set of hosts can be visited several times in random order, and a short message digest allows for the integrity verification of all the collected data.

Chapter 9 ends this thesis with a conclusion and some unanswered questions.

1.5 References

[BMW98] Ingrid Biehl, Bernd Meyer, and Susanne Wetzel. Ensuring the integrity of agent-based computations by short proofs. In Kurt Rothermel and Fritz Hohl, editors, *Proc. of the Second International Workshop, Mobile Agents 98*, pages 183–194, 1998. Springer-Verlag Lecture Notes in Computer Science No. 1477.

[CGH+95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communication Systems*, 2(5):34–49, October 1995.

[CHK97] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? – update. In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GMM98] R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, June 1998.

[Gon98] Li Gong. Secure Java class loading. *IEEE Internet Computing*, 2(6):56–61, November- December 1998.

[HI99] Daniel Hagimont and Leila Ismail. A performance evaluation of the mobile agent paradigm. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 306–313, Denver-USA, November 1999.

[HLPS98] Brant Hashii, Manoj Lal, Raju Pandey, and Steven Samorodin. Securing systems against external programs. *IEEE Internet Computing*, 2(6):35–45, November-December 1998.

[JK99] Wayne Jansen and Tom Karygiannis. Special publication 800-19: Mobile agent security. Technical report, National Institute of Standards and Technology, August 1999.

[KLO97] Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for Aglets. *IEEE Internet Computing*, 1(4):68–77, July-August 1997.

[KP00] Günter Karjoth and Joachim Posegga. Mobile agents and telcos' nightmares. *Annales des Télécommunications*, 55(7-8):29–41, 2000.

[Kru97] Bruce Krulwich. Automating the internet: Agents as user surrogates. *IEEE Internet Computing*, 1(4):34–38, July-August 1997.

[KT98] Neeran Karnik and Anand Tripathi. Design issues in mobile-agent programming systems. *IEEE Concurrency*, 43(6):52–61, July-September 1998.

[LMR00] Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile code security. In *Proceedings of ISYPAR 2000 4ème Ecole d'Informatique des Systèmes Parallèles et Répartis*, Toulouse, France, February 2000.

[LO99] Danny Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[MGM99] Pattie Maes, Robert Guttman, and Alexandros Moukas. Agents that buy and sell. *Communications of the ACM*, 42(3):81–91, March 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[NL98] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents Security*, Lecture Notes in Computer Science N. 1419, pages 61–91. Springer-Verlag, 1998.

[Pic98] Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino, 1998.

[RG98] Aviel Rubin and Daniel Geer. Mobile code security. *IEEE Internet Computing*, 2(6):30–34, November-December 1998.

Privacy of Execution

2.1 Introduction

This chapter deals with the problem of privacy of execution. The goal of privacy of execution is to hide the actions performed by a piece of code from the environment that executes the code. We do not focus on the confidentiality of this code when transmitted over the network. There are several applications where privacy of execution is important:

- *Personal assistants*. With the increasing use of personal assistants for searching and shopping applications, someone who monitors the intentions of these agents can retrieve a large amount of information about the owner of the agents. In particular, the execution environment where each agent executes should be considered as a potentially malicious party from which privacy violations can originate;

- *Electronic Commerce - bargaining agents*. It is important to hide the strategy implicitly implemented within the code. In order to conduct a fair negotiation, each party's reasoning process about the negotiation should be kept secret from the other participant. The typical application of this scenario is a shopping agent that is able to bargain on behalf of its owner.

- *Valuable algorithms*. Privacy of execution is required to prevent the disclosure of the algorithms implemented by mobile code.

Privacy of execution is not a new problem, but research has not come up with many practical solutions. The emergence of the two first applications has renewed the need and motivation for research on this topic.

Confidentiality of data has been thoroughly studied and many encryption methods have been proposed to satisfy this requirement. Nevertheless, the use of data encryption techniques to address privacy of execution has to take into account that the ciphertext must be understood by the execution environment in order to be executed. The fact that the execution environment needs full access to both code and data segments in order to accomplish the execution has incited some authors [CGH+95] to hypothesize that privacy of mobile code cannot be effectively achieved.

In this chapter, an overview of the techniques developed to tackle this problem is provided. This overview tries to be as comprehensive as possible and attempts to cover both theoretical and heuristic settings. In order to better understand these techniques, the threats to the code execution in terms of privacy are enumerated. Then, we present the existing solutions to this requirement. At the end, a more formal definition of privacy of execution is presented and discussed.

2.2 Threats

The execution environment may threaten the privacy of execution of mobile code in several ways. Privacy of execution encompasses all the information the environment can retrieve from the execution of the code. Roughly, it is possible to classify these threats as follows:

- *Code interpretation*. In order to execute the code, the environment has to understand it. Therefore, the environment can inspect the code with the objective of disclosing the functions that the code encompasses. Due to the fact that source code is usually easier to understand than machine code, the first step of this attack often consists of performing the inverse

function of a compiler. The automatic tools that implement this first step are usually called de-compilers;

- *Code interpolation*. In this case, the code is regarded as a black-box and the analysis performed by an attacker focuses on the relationship between inputs and outputs, that intrinsically defines the functions implemented within the code. This attack is always possible if the input and output data are in cleartext form, and its complexity basically depends on the complexity of the implemented functions.

The privacy of execution requirement has been tackled in different ways, as follows:

- Tamper-proof hardware (TPH);
- Obfuscation;
- Secure function evaluation.

In the next three sections, we analyze these approaches in detail, and provide references to related work.

2.3 Tamper-Proof Hardware

The use of tamper-proof hardware aiming at mobile code protection was proposed in [WSB98] and [Yee97]. The approach suggested in [WSB98] requires code and data segments to be confined to trusted execution environments. In addition, providing privacy with that approach requires all the code to be transmitted over a private channel. This work focuses on implementation issues, such as the authentication of the code and its installation in the TPH.

The solution in [Yee97] considers a model with a trusted CPU and untrusted memory. The computation is therefore trusted, and the security concerns are shifted to the integrity of data stored in untrusted memories. The same setting is addressed in the more general proposal for software protection in [GO96]. However, the latter scheme is more complete because it hides all information about the software, i.e. the contents of the memory and the memory access pattern.

In all solutions referred to so far, the computation is executed inside the trusted TPH. Therefore, they all share the drawback of the limited capacities of the more practical and cheaper forms of Tamper-Proof Hardware such as smartcards.

It is not always efficient to rely on limited devices to perform complex tasks. This was the starting assumption for the proposals in [MKI88], [Fei93], and [Bla96] where the trusted device takes advantage of the superior computational power of an untrusted host without compromising security. These solutions focused on the specific applications of public-key cryptography ([MKI88] and [Fei93]) and symmetric key encryption, called Remotely Keyed Encryption [Bla96] (afterwards formalized in [BFN98]). In addition to server-aided RSA computation, [MKI88] provides some solutions to other applications, such as matrix multiplication and solving modular equations, but all the schemes presented in this paper were sucessfully attacked in [PW92]. The work presented in [Fei93] provides a comprehensive and formal treatment of host-assisted public-key cryptography when several servers are available.

Solutions for host-assisted computation focusing on more general functions can be found in the field of secure function evaluation discussed later in this chapter. The issues presented in this section will be revisited later, in our description of solutions using limited Tamper-Proof Hardware in chapter 6 and 7. One drawback of all approaches relying on Tamper-Proof Hardware is the additional cost of the TPH itself. Furthermore, these approaches must establish a private channel between the function owner and the TPH to transmit the function in order to achieve privacy.

2.4 Obfuscation

Obfuscation, as described in [CTL98] and [Hoh98], is a mechanism that transforms an application into another application which is functionally identical but more complex to understand. Functionally identical means that the two applications are equivalent concerning the relationship between inputs and outputs. There are many obfuscators proposed as automatic tools that encompass mostly heuristic and ad-hoc algorithms that scramble code

in order to render the code interpretation more complex. A taxonomy of obfuscation techniques can be found in [CTL97].

This approach does not address interpolation attacks. The goal of obfuscation is to render the interpretation of the code harder than learning its semantics by running it. One of the advantages of this technique is the low increase of the program size. Its major drawback is the lack of theoretical foundations in order to establish precise definitions of security, and accordingly to be able to quantify the security of the underlying transformations.

2.5 Secure Function Evaluation

This section focuses on the problem of preserving the privacy of functions executed on untrusted environments in a more theoretical way. Two parties are involved: the function owner and the environment that executes the function. The execution environment provides input data to the function. When getting the result of the function execution, the function owner does not want to reveal information about the function.

In other words: Alice has a function f and Bob has an input x. The function evaluation takes place in Bob's environment, but he should learn nothing significant about the function. Alice should learn the value $f(x)$.

The problem addressed here is sometimes called computing with encrypted functions and can be seen as an instance of the more general problem of secure function evaluation. Research on cryptography has addressed this problem from different points of view. This section provides an overview of these techniques and attempts to analyze them taking the mobile code scenario into perspective.

The problem of secure function evaluation was mainly addressed in two different scenarios, as follows:

- Secure multi-party computation;
- Instance hiding.

2.5.1 Secure multi-party computation

In this case, each party has one input to the public function to be evaluated. The goal of each player is to compute the result of the function revealing no information about his own data. More precisely, the information about each party's inputs that may be extracted from the execution of the scheme should be, at most, the information that can be retrieved from the overall result.

This problem was introduced by Yao in [Yao82] with the famous "Millionaires' Problem" where two millionaires want to know who is richer without revealing their wealth. The author proposed a solution based on the intractability assumption of factoring. In [Yao86], a more detailed formalization of the problem is presented. Later, Goldreich, Micali and Wigderson [GMW87] weakened the intractability assumption to the existence of any trapdoor permutation and extended the solutions to the multi-party case. However, the round complexity of the solution is at least linear in the depth of the underlying circuit. The work of [BMR90] focused on reducing the round complexity without significantly increasing the communication complexity, but both complexities continue to be high.

2.5.2 Instance Hiding

These schemes highlight the dichotomy between two parties that do not necessarily trust each other: the owner of the data and the owner of the circuit or execution environment. In this case, the data owner is a computationally limited player that wishes to use the help of a powerful but untrusted player (called an oracle) to compute a complex function. The limited player wishes to hide information about his data while retrieving the result of the function.

This problem was first introduced by [RAD78] in terms of "computing with encrypted data" (CED). The data owner wants to keep his data secret while the data is stored in an untrusted host. Moreover, the data owner wants to perform operations in the stored data without having to decrypt it for performing operations and then re-encrypting. The authors tried to find a homomorphic encryption scheme E with respect to an operation op. An

encryption scheme E is homomorphic concerning the operation op if there is an efficient operation op' to compute $E(x \; op \; y)$ from $E(x)$ and $E(y)$, that is $E(x \; op \; y) = E(x) \; op' \; E(y)$.

Nevertheless, the solutions proposed by the authors were sucessfully attacked in [BO92]. Later, [BL96] proved that all such deterministic encryption schemes are insecure.

The secure function evaluation problem was also referred to in the paper by Abadi, Feigenbaum and Kilian [AFK89], and it was called "hiding data from an oracle". In this case, a computationally limited player wants to use the computational power of a powerful player (the oracle) but without disclosing his data. The authors formalized the problem of computing with encrypted data in the two-party case. Based on this idea, Abadi and Feigenbaum [AF90] developed a two-party protocol for secure circuit evaluation, that is also similar to the protocol of [CDvG87], and which allowed a player to evaluate his data on another player's Boolean circuit. This protocol preserved the confidentiality of the data and also hid the circuit from the owner of the data. The major drawback of both protocols is the round complexity between the two players, which is proportional to the depth of the Boolean circuit.

Protocols for computing with encrypted data can be characterized by how they perform the evaluation in an operation-by-operation basis (gate-by-gate in the case of circuits). A natural way of designing these protocols is to use privacy homomorphisms, but this has been an open research issue since the seminal paper of [RAD78].

2.5.3 Computing with encrypted functions

The problems of secure multi-party computation and hiding instances are closely related to the problem of computing with encrypted functions. We may build solutions for computing with encrypted functions based on the two kinds of schemes described above. In the case of secure multi-party computation, if a universal circuit is used as the public function, then the description of the function is the private input of the function owner. Once again, using universal circuits, the encrypted data in instance hiding

schemes can represent the encrypted form of a circuit. The use of universal circuits incurs a cost of a polynomial expansion [CCKM00]. On the other hand, a thorough study about the characterization of more general functions (non-Boolean) that can be securely evaluated was begun in [CK91]. In general, secure two-party computation is not possible without some complexity assumption, as shown in [BGW88] for the case of an OR gate.

The major drawback of the solutions that achieve secure function evaluation is their complexity in terms of round and communication complexity. One of the main advantages of mobile code is the ability to execute tasks locally in an autonomous way, as seen in Chapter 1. Autonomy means that the code is able to perform its tasks with no interaction with the code owner. Therefore, non-interactivity is an important requirement. Next, we look at non-interactive solutions in greater detail.

Non-Interactive Solutions

Figure 2.1 describes the non-interactive setting for providing privacy of execution. There are three phases without interaction:

- *The scrambling phase*, where a transformation E is applied to f. The resulting function $E(f)$ should leak no information about f;
- *The evaluation phase*, where the function $E(f)$ is evaluated on the data x;
- *The result retrieval phase*, where the cleartext result $f(x)$ can be derived from the encrypted result $[E(f)](x)$ using a transformation D.

Sander and Tschudin [ST98] illustrated the concept with a method that allows to compute with encrypted polynomials, based on the Goldwasser-Micali [GM84] probabilistic encryption scheme. The authors took advantage of the additively and mixed multiplicatively homomorphic properties of the encryption scheme. The transformation E consists of the encryption of the coefficients of the polynomial. Thus, this technique does not hide the skeleton of the polynomial.

Recently, [SYY99] presented a non-interactive solution for secure evaluation of log-depth circuits. The evaluation is done in a gate-by-gate basis using a new privacy homomorphism for processing NOT and OR gates in a

secure way. The restriction on the depth of the circuit comes from the increase of the output size by a constant factor when computing an OR gate.
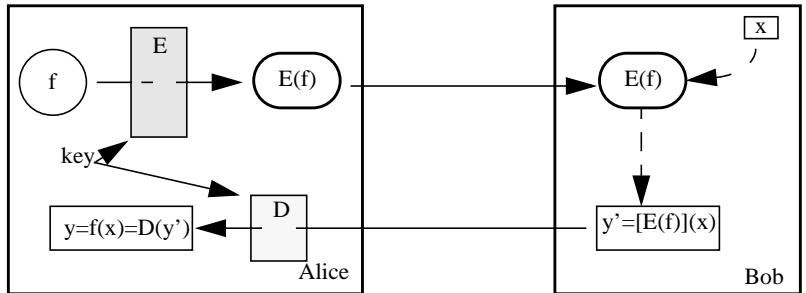
Figure 2.1 **Non-interactive protocol for privacy of execution**

In [CCKM00], a more powerful technique that combines the technique of "garbled" circuits by [Yao86] and one-round oblivious transfers (see for example [Kil89]) is presented. This work extended non-interactive secure computations to polynomial-size circuits. Moreover, the technique addressed the multi-hop scenario, where a mobile code may visit several (possibly malicious) hosts. Each visited host can use outputs computed on previously visited hosts without learning anything about the previously used inputs and obtained outputs.

The major drawback of the secure function evaluation approaches is their complexity in terms of computation, communication and rounds. This is the cost of having very strong security properties related to cryptographic or information-theoretic assumptions. There is a panoply of different schemes that fit different requirements, such as non-interactivity. However, non-interactivity is achieved with an additional cost in terms of computational and communication complexity. We invite the reader to refer to [Fra93] and [Gol98] for an in-depth analysis of the theoretical approaches to the problem of secure function evaluation.

2.6 Requirements

In this section, we focus on a scenario without TPH. The requirements imposed to solutions with TPH will be elaborated in chapter 6. Our goal is to address flexible models of computation, in an efficient way. The overhead brought up by the security primitives must not render the use of mobile code unattractive. We can now define an ideal set of requirements that should be imposed on techniques addressing privacy of execution in the mobile code scenario.

We consider the same non-interactive scenario as depicted on Figure 2.1. The privacy of f is assured by the algorithms {E, D} that should satisfy the following properties, stated in an informal way:

Correctness. There is a polynomial time algorithm D to retrieve the desired result $f(x)$ from the obtained result $y'=[E(f)](x)$;

Privacy of execution. The resulting function $f'=E(f)$ preserves the privacy of f if it is computationally unfeasible to derive f from f'. We noticed from related work that perfect privacy is hard to achieve. Therefore, the solutions may admit information leakage about the function f that should be minimized;

Remote host computational complexity. The complexity of f' should not be increased by more than a polynomial factor when compared with the computational complexity of f;

Communication complexity. The communication complexity should not be increased by more than a polynomial factor when compared with the scheme where f itself is transmitted and evaluated;

Round complexity. The scheme should be non-interactive, or the round complexity equal to two. The evaluation of f' should be performed without the help of Alice. Additionally, algorithms E and D should not need the help of Bob;

Privacy of data. The obtained result y' should not reveal more information about data x to the function owner than what the actual result y reveals.

This is an ideal set of requirements that will be analyzed during the security evaluation of our solutions in chapter 5. These requirements will be adapted in chapter 6 to the case where Tamper-Proof Hardware is available.

2.7 Conclusion

Two main approaches have been developed to address the problem of privacy of execution without Tamper-Proof Hardware. On one hand, there are more practical solutions like obfuscation with a lack of theoretical foundations. On the other hand, there are the stronger theoretical solutions of secure function evaluation, that apply to more limited models such as circuits and that have a large complexity associated to each bit of output. Solutions for code privacy against possibly malicious hosts are thus still in their infancy.

Our aim is to create new solutions that offer a trade-off between these two worlds: less complex and more flexible than secure function evaluation and stronger in security terms than obfuscation. The set of requirements defined is hard to fulfill in a complete way. Our objective is to achieve security in cryptographic terms while reducing the complexity of the solutions.

In addition, our objective is also to tackle the problem of integrity of execution in an efficient way. The function owner should have a way of verifying if the obtained result is a possible result of the function. Usually, the integrity problem is not considered in the scenarios illustrating privacy of execution. This fact relies on the trust assumption that if the host cannot get information about the function, then the remote host will not be able to tamper with the function in a meaningful way. In the next chapter, we focus on the problem of integrity of execution. In Chapters 5 and 6, we will show how to tackle privacy and integrity of execution in an efficient way.

2.8 References

[AF90] Martín Abadi and Joan Feigenbaum. Secure circuit evaluation. *Journal of Cryptology*, 2(1):1–12, 1990.

[AFK89] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On hiding information from an oracle. *Journal of Computer and System Sciences*, 39(1):21–50, August 1989.

[BFN98] Matt Blaze, Joan Feigenbaum, and Moni Naor. A formal treatment of remotely keyed encryption. In Kaisa Nyberg, editor, *Advances in Crytology - EUROCRYPT '98*, Lecture Notes in Computer Science, pages 251–265, Finland, 1998. Springer-Verlag.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for fault-tolerant distributed computing. In *ACM Symposium on the Theory of Computing 88*, pages 1–10, Chicago, 1988.

[BL96] Dan Boneh and Richard J. Lipton. Algorithms for black-box fields and their application to cryptography (extended abstract). In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 283–297. Springer-Verlag, 18–22 August 1996.

[Bla96] Matt Blaze. High-bandwidth encryption with low-bandwidth smartcards. In Dieter Grollman, editor, *Fast Software Encryption: Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 33–40, Cambridge, UK, 21–23 February 1996. Springer-Verlag.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, Maryland, 14–16 May 1990.

[BO92] E.F. Brickell and A.M. Odlyzko. *Cryptanalysis: A survey of recent results*, chapter Contemporary Cryptology - the Science of Information Integrity, pages 501–540. IEEE Press, 1992.

[CCKM00] C. Cachin, J. Camenisch, J. Kilian, and Joy Muller. One-round secure computation and secure autonomous mobile agents. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming-ICALP 2000*, Geneva, July 2000.

[CDvdG87] David Chaum, Ivan B. Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 87–119. Springer-Verlag, 1987.

[CGH+95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Paris, and Gene Tsudik. Itinerant agents for mobile computing. IBM Research Report RC 20010, IBM, March 1995.

[CK91] B. Chor and E. Kushilevitz. A zero-one law for Boolean privacy. *SIAM Journal of Discrete Mathematics*, 4:36–47, 1991.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences-The University of Auckland, July 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998.

[Fei93] Joan Feigenbaum. Locally random reductions in interactive complexity theory. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 13:73–98, 1993.

[Fra93] Matthew Keith Franklin. *Complexity and Security of Distributed Protocols*. PhD thesis, Columbia University, 1993.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest

majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[Gol98] Oded Goldreich. Secure multi-party computation. Working Draft, Version 1.1, 1998.

[Hoh98] Fritz Hohl. An approach to solve the problem of malicious hosts. In *4th ECOOP Workshop on Mobility: Secure Internet Mobile Computations*, 1998.

[Kil89] Joe Kilian. *Uses of Randomness in Algorithms and Protocols*. PhD thesis, Massachusetts Institute of Technology, 1989.

[MKI88] T. Matsumoto, K. Kato, and H. Imai. Speeding up secret computations with insecure auxiliary devices. In S. Goldwasser, editor, *CRYPTO88*, pages 497–506. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 403.

[PW92] Birgit Pfitzmann and Michael Waidner. Attacks on protocols for server-aided RSA computation. In R. A. Rueppel, editor, *Advances in Cryptology—EUROCRYPT 92*, volume 658 of *Lecture Notes in Computer Science*, pages 153–162. Springer-Verlag, 24–28 May 1992.

[RAD78] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In DeMillo, Dobkin, Jones, and Lipton, editors, *Foundations of Secure Computation*, pages 169–180. New York: Academic Press, 1978.

[ST98] Tomas Sander and Christian Tschudin. Towards mobile cryptography. In *Proceeding of the 1998 IEEE Symposium on Security and Privacy*, pages 215-224 Oakland, California, May 1998.

[SYY99] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC1. In *Proceedings of the IEEE FOCS*, October 1999.

[WSB98] Uwe G. Wilhelm, Sebastian Staamann, and Levente Buttyan. Protecting the itinerary of mobile agents. In *4th ECOOP Workshop on Mobility: Secure Internet Mobile Computations*, 1998.

[Yao82] A.C. Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science 82*, pages 160–164, Chicago, 1982.

[Yao86] A.C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science 86*, pages 162–167, Toronto, 1986.

[Yee97] Bennet Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC at San Diego, Dept. of Computer Science and Engineering, April 1997.

Integrity of Execution

3.1 Introduction

As stated before, this thesis deals with the protection of mobile code from the execution environment, a problem brought up by the mobile code paradigm. In this scenario, there is a conflict between two sides: the mobile code and the security requirements of its owner on one side, and on the other side the execution environment and its security requirements. We focus on mobile code protection from possibly malicious hosts, namely the problem of integrity of execution. The owner of the mobile code should be able to verify the correctness of the code execution. Our concern is not the integrity of the code itself, but the integrity of its execution.

Without integrity of execution, the number of possible applications for mobile code becomes smaller and the computations are restricted to trusted environments. The possibility of detecting misbehaviors from remote hosts is also fundamental for the enforcement of control policies.

This chapter presents a short but comprehensive survey of techniques addressing integrity of execution. Then, a definition for integrity of execution is given and analyzed.

3.2 Problem Statement

The aim of integrity of execution is to give the code owner the ability to verify the correctness of the execution of his code. This can be done at the end of the agent trip, or as suggested in [FGS96], the code can contain state appraisal functions that will verify the state of the agent on each host that the agent visits. The authors claimed that an agent can become malicious when its state is tampered with, based on the example of a shopping agent that searches for n seats on an airplane. A malicious host can augment the number of seats to be reserved in order to cause bogus reservations on competing hosts. In this case, it is important to check the state of the code at each host.

In our case, the code owner wishes to verify the integrity of the results at the end of the code's trip. The code owner should be able to detect attacks regarding the correctness of the code execution from the remote execution environments. It is possible to classify the solutions to this problem into four categories, as follows:

- Proof systems;
- Traces of execution;
- Fault tolerance;
- Result checking.

Moreover, the solutions called secure function evaluation schemes, presented in chapter 2, also prevent any player or groups of players from cheating. Therefore these solutions also achieve integrity of execution. Obviously, settings relying on tamper-proof hardware accomplish this goal as well. In the next sections, we analyze in more detail the existing approaches that deal with integrity of execution.

3.3 Proof Systems

Yee [Yee97] suggested the use of proof based techniques to address the problem of integrity of execution in the mobile code scenario. Each host

that executes the mobile code has to send a proof of the correctness of the execution with the result. The objective of proof systems is to make the verification of the proof easier than the construction of the proof.

Definition: A language L belongs to NP if there exists a polynomial time non-deterministic Turing machine V such that for all $x \in \{0, 1\}^n$:

- $x \in L$ implies that there exists a sequence of non-deterministic choices (or proof) π, such that V on choice π accepts the input x;

- $x \notin L$ implies that there is no proof π for which V will accept x.

Proof systems have been extended over the years with the possibility of a probabilistic verification procedure. The results include very well known techniques such as interactive proof systems and zero-knowledge proofs. There is extensive work on proofs, see for example [Gol99] for a comprehensive survey.

In [BC86], the authors illustrated this concept with a protocol where Alice can convince Bob of the good result of a Boolean circuit simulation, without revealing her inputs, under the Quadratic Residues Assumption. However, this protocol required the knowledge of the Boolean circuit by both parties. The protocol developed for the circuit model entailed a communication complexity dependent on the number of wires or gates of the circuit.

In [Yee97], the author pointed out the possible use of Probabilistic Checkable Proofs (PCP) [ALM+91] [AS98]. PCP proofs assure the correctness of a statement while checking only a subset of the proof. In more formal terms:

Definition: A language $L \in PCP(r, q)$ if there exists a probabilistic polynomial time oracle machine V, accessing a polynomial sized oracle π, such that for all $x \in \{0, 1\}^n$:

- $x \in L$ implies that $\exists \pi$ such that for all choices of random strings, the verifier V accepts x.

- $x \notin L$ implies that $\forall \pi$, V rejects x with probability of at least 1/2 (over internal coin flips).

Furthermore, V uses at most $r(n)$ coin flips for its probabilistic verification and queries the oracle π at most $q(n)$ times.

This concept is similar to the definition of transparent proofs introduced in [BFLS91] and [Bab93]. With transparent proofs, the input has to be provided to the verification process in an encoded form (using a specific error correcting code). In this way, membership proofs for NP may be verified in polylogarithmic time. One interesting result is that *NP=PCP(log n, 1)* [ALM+91], which means that there are proofs of membership for any NP language that can be checked by examining only a constant number of bits of the proof. However, the process of converting a proof into a PCP proof produces a super-quadratic increase in size [PS94]. Fortunately, the same authors presented an optimization in [PS94], and probabilistic proofs of nearly linear size were obtained.

In our case, it is important to analyze the round complexity of these approaches. One appealing idea is the use of Non-Interactive Zero Knowledge Proofs [BSMP91]. Based on some reasonable complexity assumptions, one may construct non-interactive zero-knowledge proofs for every NP language, but the prover and the verifier have to interact with a trusted random sequence of bits (which can be considered as being selected by a trusted third party [Gol99]).

PCP proofs may be used non-interactively, but the subset of verified bits has to be randomly determined by the verification process and the prover has to commit to the overall PCP proof. This commitment is necessary in order to prevent the prover from adapting the proof to the queries being made. In the mobile code scenario, the use of PCP proofs in a non-interactive way implies that the prover should send the complete PCP proof to the code owner. Therefore, the increased size of PCP proofs when compared with non-probabilistic proofs becomes a drawback in the mobile code scenario.

In [BMW98], the authors presented an interesting model for mobile computing as well as a solution that overcomes the problem of using PCP

proofs. The agent is modelled as a probabilistic Turing machine, and the set of all possible states of this machine constitutes an NP language. So, the verification process consists of checking whether an obtained state belongs to this language. In order to avoid the transmission of the overall PCP proof, the randomly chosen queries from the checker are encrypted using non-interactive Private Information Retrieval (PIR) techniques [CGKS98] [CMS99]. These private queries are then sent to the remote host. In a few words, single database PIR consists of retrieving a bit from a string (the database), without revealing the index of the desired bit. Therefore, the remote host has only to send the answers to the encrypted queries and not the complete PCP proof.

We may use more practical and attractive proof systems in which the prover is restricted in computational terms, such as cryptographic CS-proofs [Mic94] and arguments [Kil92]. Considering intractability assumptions, it is then possible to build more communication efficient proof systems. With cryptographic CS proofs, the programs may be transformed in order to supply a very short proof which is easy to inspect in addition to the result [Mic94]. Nevertheless, it is an open problem whether such cryptographic proofs can be constructed, for example for NP. In the case of CS-proofs and argument systems, we need either three messages of interaction or the assumption of the existence of a random oracle [Gol99], consequently they are not really non-interactive.

Recently, [ABOR00] improved the above results to one-round proofs for NP with poly-logarithmic communication complexity in the length of the proof. The authors extended the idea presented in [BMW98] in that they used a combination of PCP proofs and PIR techniques. Nevertheless, the solution proposed in [ABOR00] achieves stronger security properties.

3.4 Traces of Execution

In a more practical sense, one could think about storing the entire trace of execution in the mobile code. Then, the code owner would re-execute the code and compare it with the trace. Such a process is so cumbersome that it renders the use of mobile code unattractive.

A more efficient way of implementing this idea was proposed in [Vig97]. Firstly, the author divided the code statements into white and black statements. The black statements are the critical ones because they result from interactions with the environment. The trace of execution of the black statements has to be signed and transmitted to the code owner. Secondly, in order to avoid the transmission of the complete trace, the author proposed that the remote host send a hash of the trace. The host hereby commits to the trace. If the code owner suspects that the code was not correctly executed, the remote host has to send the trace to prove his correct behavior. A Trusted Third Party may arbitrate conflicts at the moment they occur, thus enforcing an optimistic control of integrity of execution. The drawbacks of this approach are:

- Each host has to keep the traces of the executions it performs in order to prove its innocence in case of dispute;
- Each time the code owner wants to perform a verification, he has to re-execute the code.

In order to increase the power of the traces, we may use the techniques described in [KV98] where the authors used fault injection to simulate possible malicious host systems. The objective is to build assertions, that will be included in the code, therefore increasing the observability of the traces. The traces of execution become more powerful and an oracle may be automatically generated to check these traces. Once more, the traces have to be sent to the code owner in order to be verified with the help of the oracle.

3.5 Fault Tolerance

Fault tolerance approaches are based on techniques for software reliability. The distinguishing feature of these approaches is the redundancy of executions, which means that the result of the computation comes from the co-operation between different execution environments.

In [Rot98], the problem of integrity of execution is mentioned in the case of the protection of a mobile agent's itinerary. Preventing the tampering of routing decisions by the agent is not easy, if these decisions are to be taken

in a possibly malicious host. This problem is a good illustration of the need for integrity of execution. Without integrity of execution, the mobile code can be sent where the execution environment wants, rather than to the place it was supposed to go otherwise. The author proposed using a backup agent executed on a trusted environment. This backup agent verifies the routing decisions taken by the agent executed on a possibly malicious host.

Without recurring to previously known trusted environments, [MvRSS96] presented an approach where several instances of the same code are executed on different hosts and the right result is elected by a voting mechanism. The very efficient solution for voting proposed in [Nou96] can be used if the voting procedure is performed on a trusted environment. The security of the scheme in [MvRSS96] relies on the fact that a bigger number of executions should occur in non-malicious hosts. Thus, the approach assumes that attackers cannot have access to all the execution environments, or that the execution environments do not cooperate. However, this model is not realistic because it supposes that there exists replicated servers that are not controlled by the same entity.

3.6 Result Checking

Program result checking, first introduced in [Blu88] and further developed in [BK89], deals with the problem of software reliability. The authors defined a program checker in the following way:

Definition: Given a program P that should implement a function f, a program checker C aims at catching with high probability that for an arbitrary input x, $P(x) \neq f(x)$. Moreover, the checker should be efficient in the sense that its computation time should be shorter, i.e. $o(T(n))$, than the smallest known computation time $T(n)$ for f.

The latter condition prevents the checker from just re-computing the function at the given point. The methodologies used in program checking are usually linked to the algebraic properties of the function, namely random self-reducibility:

Definition: A function f is k-random self-reducible if $f(x)$ can be expressed as an efficient function of $f(x_1), f(x_2), ..., f(x_k)$, where the x_i are uniformly and randomly distributed.

A similar concept called coherent functions was defined by [Yao90], which allows an adaptive choice of points x_i. The self-reducibility property can be used by querying the program P on several random inputs and then checking a known relationship between the outputs. Furthermore, result checking techniques consider program P as a non-malicious adversary. Result checking aims at achieving reliability (refer to [BW97] for a survey) without taking into account a possible malicious behavior from the execution environment. Being resilient to a malicious behavior is equivalent to having a program P that attempts to defeat the checker. However, this requirement is not included in the definition of a checker.

In addition, if the program owner does not trust the checker, then we may use witnesses [FGY96]. In order to address this requirement, the program owner constructs the answers to the checker in such a way that the checker can infer no information about the program from these answers. For the same reasons, witnesses do not tackle the possible maliciousness of program P.

3.7 Verifiers

We now present the definition of a new concept meant to address the problem of integrity of execution. This definition is presented in the same non-interactive scenario of chapter 2.

Using the same notation as in chapter 2, Alice receives the result y' of the execution of function f'. With this value, Alice should be able to retrieve the result y of the execution of the function f (using algorithm D) and to perform a polynomial time verification of this result, using a verifier V.

Definition: Given a program that should implement a function f and considering Y as the set of all possible outputs of the function $f(x_i)$, with $x_i \in \{0, 1\}^l$, a verifier V satisfies the following condition:

if $(y = D(y')) \notin Y$ then $P(V(y') = Accept) < \delta$.

Where P represents a probability and thus δ is the error probability of the verifier. Additionally, as already stated in chapter 2, the computation and communication complexities of transmitting and calculating y' should not exceed the complexities associated with y by more than a polynomial factor.

The verifier concept has a common point with CS Proofs [Mic94] in that there may exist invalid proofs, but they should be hard to find. In short, the integrity of execution property relies on the difficulty of finding a value y' accepted by the verification process, that do not correspond to a valid output y.

To our knowledge, the verifier definition is original. In spite of being similar to the checker definition [BK89], there are two main differences between the two definitions:

- A checker focuses on the correctness of a computation for specific inputs x_i, whereas a verifier aims at detecting that the output is a possible output of the function f. The computations performed by the verifier do not rely on the knowledge of the input x_i;

- Maliciousness of the program (or its execution environment) is not taken into account in the checker definition. A checker needs the computation of a set of outputs of the function and checks the relationship between these outputs. A malicious program may build a set of outputs that satisfy the relationship but that are not outputs of the program.

Note that a verifier only ensures that the cleartext result is a possible output of the function. However, the remote host is able to identify all the possible outputs.

3.8 Conclusion

Detecting wrong executions of a mobile code is a fundamental issue for the success of the mobile code paradigm. We reviewed the existing approaches to the problem. The solutions inspired by software reliability techniques (fault tolerance and result checking) do not cope with the maliciousness of the execution environment. The solutions satisfying stronger security properties, such as proofs and traces of execution, suffer from their high complexity.

We defined the new concept of verifier to address the requirement of integrity of execution. We believe this concept is more suitable for the design of efficient solutions and for expressing the possible maliciousness of the execution environment. However, a verifier does not prevent re-executions of the function and selection of the best result. Furthermore, it does not prevent attempts to invert the function in order to retrieve an input that corresponds to a desired output. However, when integrated with privacy of execution, the two attacks mentioned above are defeated.

Briefly, a verifier only guarantees that it should be computationally intractable to a malicious execution environment, to find values accepted by the verifier for bogus outputs. The implementation of verifiers should be efficient, that is the remote hosts should have a limited increase in computational and communication complexity and the verification process should be easy.

We have searched for implementations in the field of coding theory. The goal is to build programs that are resilient to errors of execution, such as data is resilient to errors of transmission when encoded by an error correcting code.

3.9 References

[ABOR00] W. Aiello, S. Bhatt, R. Ostrovsky, and S. Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In *Proocedings of the 27th International Colloquium*

on Automata, Languages and Programming-ICALP 2000, Geneva, July 2000.

[ALM+91] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Foundations of Computer Science*, pages 14–23, October 1991.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998.

[Bab93] László Babai. Transparent (holographic) proofs. In *10th Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 525–534, Würzburg, Germany, 25–27 February 1993. Springer.

[BC86] Gilles Brassard and Claude Crépeau. Zero-knowledge simulation of Boolean circuits. In A. M. Odlyzko, editor, *Advances in Cryptology— CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 223–233. Springer-Verlag, 1987, 11–15 August 1986.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 21–31, New Orleans, Louisiana, 6–8 May 1991.

[BK89] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 86–97, Seattle, Washington, 15–17 May 1989.

[Blu88] Manuel Blum. Designing program to check their work. Technical Report TR-88-009, International Computer Science Institute, December 1988.

[BMW98] Ingrid Biehl, Bernd Meyer, and Susanne Wetzel. Ensuring the integrity of agent-based computations by short proofs. In Kurt Rothermel and Fritz Hohl, editors, *Proc. of the Second International Workshop, Mobile*

Agents 98, pages 183–194, 1998. Springer-Verlag Lecture Notes in Computer Science No. 1477.

[BSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, December 1991.

[BW97] Manuel Blum and Hal Wasserman. Software reliability via runtime result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[CGKS98] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, November 1998.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *Advances in Cryptology: EUROCRYPT '99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[FGS96] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, September 1996. Springer-Verlag Lecture Notes in Computer Science No. 1146.

[FGY96] Yair Frankel, Peter Gemmel, and Moti Yung. Witness-based cryptographic program checking and applications (an announcement). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, page 211, Philadelphia, Pennsylvania, USA, 23–26 May 1996.

[Gol99] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Springer, 1999.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 723–732, Victoria, British Columbia, Canada, 4–6 May 1992.

[KV98] Lora Kassab and Jeffrey M. Voas. Agent trustworthiness. In *4th ECOOP Workshop on Mobility: Secure Internet Mobile Computations*, 1998.

[Mic94] Silvio Micali. CS Proofs (extended abstract). In *IEEE Proceedings of Foundations on Computer Science*, pages 436–453, 1994.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 109–114, Connemara, Ireland, September 1996.

[Nou96] Guevara Noubir. *Nouvelles Techniques pour la Tolérance aux Pannes Basées sur l'Algèbre des Polynômes*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1996.

[PS94] Alexander Polishchuk and Daniel Spielman. Nearly-linear size holographic proofs. In *ACM Symposium on the Theory of Computing 94*, pages 194–203, 1994.

[Rot98] Volker Roth. Secure recording of itineraries through co-operating agents. In *4th ECOOP Workshop on Mobility: Secure Internet Mobile Computations*, 1998.

[Vig97] Giovanni Vigna. Protecting mobile agents through tracing. In *3rd ECOOP Workshop on Mobile Object Systems*, Jyvälskylä, Finland, June 1997.

[Yao90] Andrew Chi-Chih Yao. Coherent functions and program checkers (extended abstract). In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 84–94, Baltimore, Maryland, 14–16 May 1990.

[Yee97] Bennet Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC at San Diego, Dept. of Computer Science and Engineering, April 1997.

CHAPTER 4 *Introduction to Coding Theory*

4.1 Introduction

This chapter focuses on the cryptographic tools used to tackle the problems stated in chapters 2 and 3. The tools presented here are used in the solutions described in the following three chapters. The security of these solutions relies on intractability assumptions found in the field of coding theory.

Coding consists of adding redundant information to the data exchanged during a communication in order to allow the receiver to recover the original data even in the presence of transmission errors. Basically, an error correcting code transforms a data block of k symbols into a coded block of n symbols. The n-k redundant symbols allow for the detection and correction of transmission errors, if the number of error symbols is bounded.

The complexity of some instances of coding theory problems has been used to construct cryptosystems and our solutions also exploit the hardness of such instances.

In this chapter, we provide a brief introduction to coding theory. We focus on linear block codes and on the complexity of the decoding process in the presence of errors. We then show how it is possible to construct hard

instances of the decoding problem. The most well-known cryptosystems are described and an introduction to possible attacks is finally given.

4.2 Linear Block Codes

A *q*-ary linear block code *C* of length *n* over $GF(q)$ and of dimension *k* is a linear subspace of $GF(q)^n$ with dimension *k*.

The elements of the code are called codewords and the Hamming weight *w* of a word is the number of non-null bits of the word. The Hamming distance *d(a,b)* between two words *a* and *b* in the set $\{0, 1\}^n$ is defined as:

$$d(a, b) = |\{1 \leq i \leq n | a_i \neq b_i\}| .$$

The minimum distance *d* of a code *C* is equal to:

$$d = min\{d(a, b) | a, b \in C \wedge a \neq b\} .$$

It is straightforward to see that the minimum distance of a code *C* is equal to the minimum non-zero weight codeword of *C*. A linear code with length *n*, dimension *k* and minimum distance *d* is referred to as an [n,k,d] code. The information rate of a code is expressed as k/n. Since the code is a linear subspace of dimension *k*, there exist *k* linearly-independent codewords that form a basis of the subspace. When *k* independent codewords are the rows of a $k \times n$ matrix *G*, *G* is called a generator matrix of the code.

A vector *m* is encoded into a codeword *c* as follows: $c = mG$. A generator matrix is in systematic form when $G = (I_k | A)$, where I_k is the $k \times k$ identity matrix and A is a $k \times (n - k)$ matrix (| means concatenation). Another important matrix in coding theory is the parity-check matrix *H* of a code. This matrix has size $(n - k) \times n$ and it defines the code in the following way:

$$C = \{c \in GF(q)^n | cH^T = 0\}$$

which implies $GH^T = 0_{k, n-k}$, where $0_{k, n-k}$ is the $k \times (n-k)$ all-zero matrix. When the generator matrix is in systematic form $(I_k | A)$, then the parity check matrix assumes the form $(A^T | I_{n-k})$. Furthermore, there is always a polynomial time algorithm to derive G from H and vice-versa.

4.2.1 Error Correction

Consider $v = c + e$, where c is a codeword and $e \in GF(q)^n$ is an error vector. The syndrome of v is defined by $s = vH^T$. Therefore, the syndrome is zero if and only if v is a codeword. Hence:

$$s = vH^T = cH^T + eH^T = eH^T$$

The expression $s = eH^T$ defines *n-k* linear equations, with n unknowns (the n bits of the vector e). Thus, an error correcting scheme is a method of solving the *n-k* equations [LC83]. For example in the binary case, there are 2^k solutions, which means that there are 2^k error patterns that result in the same syndrome. Then, the most probable error pattern should be chosen. This pattern is the one that has minimum weight. In other words, the received vector should be decoded to the nearest codeword.

One can show that if a code has minimum distance $d = 2t + 1$, then there is only one error pattern satisfying the *n-k* equations and having weight inferior or equal to t. Therefore, the code is capable of correcting all the error patterns of t or fewer errors. Let c and v be the transmitted codeword and the received vector respectively. Let w be any other codeword in C. The Hamming distances satisfy the triangle inequality:

$$d(c, v) + d(w, v) \geq d(c, w)$$

Let us suppose that an error pattern of t' errors occurs. Then $d(c, v) = t'$ and since $d(c, w) \geq d \geq 2t + 1$, the following inequality is obtained:

$d(w, v) \geq 2t + 1 - t'$.

If $t' \leq t$, then $d(w, v) > t$. If an error pattern of t or fewer errors occurs, the received vector v is closer to the transmitted codeword c than any other codeword w in C. Thus, the value t represents the maximum number of errors the code can correct.

4.3 Complexity

In their seminal paper, Berlekamp, McEliece and van Tilborg [BMvT78] proved that the following problem, stated in the style of [GS79], is NP-complete:

Problem: Maximum-Likelihood Decoding

Instance: Let H be a $m \times n$ binary matrix, s a binary vector of length m and p a nonnegative integer.

Question: Is there a binary vector x of length n, such that $xH^T = s$ and $w(x) \leq p$?

This problem was originally termed Coset Weights in [BMvT78] and syndrome decoding in [Ste93] but we use the denomination proposed in [Var97]. This problem is more general than the one relevant to decoding because it is not stated that the matrix has maximum rank. However, this does not affect NP-completeness [BMvT78]. The problem stated in terms of the generating matrix G instead of the parity check matrix H is also NP-complete [BMvT78]. This problem was called G-Syndrome Decoding [Ver95a]:

Problem: G-Syndrome Decoding

Instance: Let G be a $n \times k$ binary matrix, y a binary vector of size n and p an integer.

Question: Is there a binary vector x of size k, and a vector e with $w(e) \leq p$, such that $xG + e = y$?

NP-completeness ensures that there is no polynomial time algorithm for solving the problem in the worst case. However, many NP-complete problems can be attacked after a preprocessing phase. Namely the beforehand knowledge of the parity check matrix can be used, but [BN90] showed that the problem remains complex even with a large amount of preprocessing on the matrix. Furthermore, the possibility of approximating the problem was also proven to be hard [ABSS97].

Nevertheless, there may exist easy instances of the problem. In the section on cryptanalysis, we will focus on this particular problem. The decision problems mentioned above are connected to the optimization problem of complete decoding:

Problem: Given a $k \times n$ matrix G of rank k and a vector x of size n, find a vector m of size k such that $d(x, mG)$ is minimum.

This problem is NP-hard. However, instances used in cryptosystems may be attacked with algorithms for decoding up to a given bound, as described later in this chapter. This problem is called Bounded Distance Decoding [Var97], and it is not necessarily NP-hard [Can96][vT94].

Recently, [Var97] proved that the Minimum Distance problem is also NP-complete. To end this section, we state the Minimum Distance problem:

Problem: Minimum Distance

Instance: Let H be a $m \times n$ binary matrix, and p a nonnegative integer.

Question: Is there a binary non-zero vector x of length n, such that $xH^T = 0$ and $w(x) \leq p$?

The author [Var97] (agreeing with [Bar97]) conjectured that the Bounded Distance Decoding is also NP-hard.

4.4 Cryptosystems

Cryptosystems based on coding theory take advantage of the hardness of the problems mentioned above. Next, the specific implementations in terms of public key cryptosystems are described and analyzed, namely the McEliece [McE78] and Niederreiter [Nie86] cryptosystems. Gabidulin [GPT91] also proposed a public key cryptosystem using rank distance codes. The cryptoanalysis of systems using rank distance codes can be found in [CS96], [Gib95a], and [Gib96] but our discussion will be restricted to Hamming distance codes.

One of the major obstacles to the widespread use of the above cryptosystems is certainly the impossibility of performing digital signatures. Several implementations have been proposed but they do not withstand cryptanalysis. It is currently believed [Ste94][vT94] that such a scheme will be hard to find.

Secret-key variants of the McEliece scheme appeared in [Jor81], [RN86] and [HR89] just to cite a few. Basically, they are very similar to the public key versions but the codes used are smaller. A description and cryptanalysis of these schemes can be found in [vT94]. We will revisit these kind of systems later in this chapter.

Some identification schemes that take advantage of these problems were also proposed, for example in [Gir90], [Ste93] (formalized in [Ste96]), and [Ver95a]. They have interesting properties including zero-knowledge but the major drawback is their communication complexity. Veron [Ver95b] studied the optimization of this kind of authentication schemes and he achieved very interesting results, including the possibility of an implementation on a very limited device such as a smartcard.

4.4.1 The McEliece Public-Key Cryptosystem

Let C be a q-ary linear $[n, k, d]$ code. Let G be a $k \times n$ generator matrix of the code C for which an efficient decoding algorithm exists. The encryption matrix is $E = SGP$, where S is a random $k \times k$ invertible matrix over $GF(q)$ and P is a random $n \times n$ permutation matrix.

Encryption: A plaintext message represented by the vector $x \in GF(q)^k$ is encrypted into the ciphertext $y \in GF(q)^n$ by $y = xE + e$, where e is a randomly chosen error vector that belongs to the set:

$$\{e \in GF(q)^n \,|\, w(e) = (d-1)/2\} \,.$$

Decryption: A ciphertext y is decrypted by $yP^{-1} = xSG + eP^{-1}$, and yP^{-1} is decoded with the decoding algorithm of C to retrieve xS (eP^{-1} is correctable since $w(eP^{-1}) = w(e)$). The plaintext x is then obtained by $x = (xS)S^{-1}$.

Keys: The public key is the matrix E and $w(e)$. The secret key consists of the matrices S and P, and of the decoding algorithm for the code C.

4.4.2 The Niederreiter Public-key Cryptosystem

Let C be a q-ary linear $[n, k, d]$ code. Let H be a $(n-k) \times n$ parity check matrix of the code C for which an efficient decoding algorithm exists. The encryption matrix is $E = SHP$, where S is a random $(n-k) \times (n-k)$ invertible matrix over $GF(q)$ and P is a random $n \times n$ permutation matrix.

Encryption: A plaintext message represented by the vector $x \in GF(q)^n$ has to satisfy the condition: $w(x) \le (d-1)/2$. The message is encrypted into the ciphertext $y \in GF(q)^{n-k}$ by $y = xE^T$.

Decryption: A ciphertext y is decrypted by $y(S^T)^{-1} = xP^T H^T$. Next, xP^T is obtained by applying the decoding algorithm of C. The plaintext x is then easily obtained: $x = (xP^T)P$.

Keys: The public key is E and d. The secret key consists of the matrices S and P, and of the decoding algorithm for C.

The McEliece scheme uses a generator matrix while the Niederreiter scheme uses a parity-check matrix, but they were proven to be equivalent in terms of security for the same code [LDW94]. The code initially proposed by Niederreiter was too small and this version was broken in [BO92]. Nevertheless, for the same code, the Niederreiter cryptosystem reveals some advantages in terms of size of the public key, the number of operations to encrypt and the transmission rate [CC98]. On the other hand, the McEliece scheme is easier to use because it does not put any restriction on the weight of the message. Moreover the complexity of the decryption algorithm is lower [CC98].

In comparison with the RSA public key cryptosystem, the disadvantages of coding theory based cryptosystems are the lack of a signature scheme, the size of the public key and the transmission rate. However, the operations of encryption and decryption are much faster (see [Can96] for a detailed analysis).

4.5 Cryptanalysis

Generally speaking, the secret key to this kind of cryptosystems is the code itself, for which an efficient decoding algorithm is known, and the public key is a transformation of the generator or parity-check matrices. In other words, the efficient decoding algorithm is the trapdoor to the public key transformation. There are two main attacks:

- Retrieving the secret code from the public code;
- Finding an efficient algorithm to decode the public code.

4.5.1 Retrieving the Secret Code

This first attack depends on the class of codes used in the instance of the cryptosystem. We focus on the McEliece scheme, which uses Goppa codes.

The complexity of the brute force attack on the original McEliece public key cryptosystem can be measured by searching exhaustively for all the possible combinations of permutations ($n!$), Goppa codes ($\sim 2^{mt} / t$, with

$m=logn$), and invertible matrices ($\sim0.29*2^{k^2}$)[MS77]. Using the code size proposed by McEliece ([1024, 524, 50]), this attack is obviously not feasible [Til88].

The other possibility is to explore the characteristics of the code transformation in order to identify its building blocks. The hardness of identifying a Goppa code from a permutation has been thoroughly analyzed.

Heiman [Hei87] was the first to study this attack and stated that the random matrix *S* used in the original McEliece scheme serves no security purpose concerning the protection of the code, because it does not change the codewords of the original code. However, Canteaut [Can96] showed that the matrix *S* may be important to hide the systematic structure of the Goppa code therefore having an important security role. Adams and Meijer [AM87] showed that the likelihood of identifying a Goppa code is small and that there is usually only one trapdoor to a given public code. Later, [Gib91] challenged this result and proved that each permutation applied to Goppa codes can be regarded as a possible trapdoor and there are at least *m.n.(n-1)* trapdoors. This results from the fact that non-equivalent Goppa polynomials can generate permuted equivalent codes. However, the number of trapdoors is very small when compared with the *n!* possible permutations. The number of trapdoors is still open for large codes, but calculated lower bounds [Gib95b] showed that an exhaustive search remains unfeasible.

Furthermore, there were efforts to find an efficient algorithm to retrieve the characteristics of the code from a permuted code represented by the generator or parity check matrix using techniques that try to identify invariants among the classes of codes [Sen94]. However, the results were negative for the case of Goppa Codes. There is currently no efficient algorithm to retrieve the characteristics of a code from a permuted generator matrix for Goppa Codes [CS98].

4.5.2 Finding a Decoding Algorithm

In spite of being conjectured as NP-hard [Bar97][Var97], many probabilistic algorithms have been developed to tackle the problem of Bounded Dis-

tance Decoding. The better known algorithms include the ones of Lee-Brickell [LB88], Leon [Leo88], Stern [Ste89] and Canteaut-Chabaud [CC98].

We refer the reader to [Cha96], [vT94] and [Can96] in order to have a more detailed view of their possible implementations and actual performances. Nevertheless, their running time increases exponentially with the length of the code. In practical terms, the mentioned algorithms are efficient to solve the problem for small codes and for errors with low weight, but the probability of success is negligible for large codes (n>1024 [CS98]) when the weight of errors is close to the maximum correction capability of a code.

4.5.3 Classes of Codes

The security of the cryptosystem is highly dependent on the class of codes used, especially concerning the complexity of the first attack, described in section 4.5.1. Some classes of codes reveal their characteristics even when they go through the permutation used to construct the public key.

Niederreiter's initial proposal used concatenated codes, which were proven to be insecure [Sen94]. Reed-Solomon codes were also proven to be insecure [SS92]. McEliece proposed Goppa codes that proved to be secure. Nevertheless, Goppa codes generated by a Goppa polynomial with binary coefficients are also insecure [Loi98].

The properties that a code should have in order to be an eligible candidate for these cryptosystems, which result from the experience gained from successful attacks against this kind of cryptosystems, are the following [CC98]:

- The type of codes must be large enough to avoid any enumeration, which is important to defeat a brute force attack. This attack can use Sendrier's algorithm [Sen97] that determines if codes are equivalent given two generator matrices. If the answer is positive, then the permutation can be retrieved. In brief, equivalence means that one matrix is a permutation of the other, because matrix S does not change the code, but only performs a modification on the basis of the linear subspace;
- An efficient decoding algorithm should exist for this class in order to build efficient systems;

- The generator or parity-check matrix of a transformation of the code must not give any information about the characteristics of the code in order to be resilient to the attack described in section 4.5.1.

If the class of codes fulfills these requirements, as in the case of Goppa codes, then the security of the cryptosystem is equivalent to the problem of bounded distance decoding any linear code [CS98].

Goppa Codes

This is the most interesting subclass of alternant codes. Concretely, Goppa codes result from the restriction to $GF(q)$ of the Generalized Reed-Solomon (GRS) codes [MS77]. Goppa codes are specified in terms of a generator polynomial $g(z)$, called a Goppa polynomial, with coefficients from $GF(q)^m$. An important property is that the minimum distance is easily estimated: $d \geq degree(g(z)) + 1$. Let $L = \{\alpha_1, ..., \alpha_n\}$ be a subset of $GF(q)^n$, such that $g(\alpha_i) \neq 0, 0 < i \leq n$. L is called the support of the code. The Goppa code $\Gamma(L, g)$ consists of all vectors $C_i \in GF(q)^n$ satisfying:

$$\sum_{i=1}^{n} \frac{C_i}{z - \alpha_i} = 0 \, mod(g(z))$$

In the binary case, if $g(z)$ is irreducible (consider $degree(g(z)) = t$) then the parameters are:

$$d \geq 2t + 1, \; n = 2^m \text{ and } k = n - mt.$$

There are approximately $2^{mt} / t$ different irreducible binary polynomials of degree t over $GF(2)^m$. A different Goppa code corresponds to each polynomial. The decoding of Goppa codewords can be done using the extended Euclidean algorithm (see for example [Sen91]) or the Berlekamp-Massey algorithm (see for example[Bla83]). For more information on Goppa Codes, see [MS77].

Without loss of generality, we focus on binary codes, because they are suitable for tackling Boolean functions, but the technique can be used for algebraic functions, where the elements are defined in GF(q). This implies the use of *q*-ary Goppa codes and the security analysis of this kind of codes can be found in [JM96].

4.5.4 Linearity

In the case of the McEliece cryptosystem, there is one weakness due to the linearity of the codes, as well as the non-deterministic nature of the system. Non-deterministic means that one plaintext results with strong probability in two different plaintexts. Furthermore, the error vectors do not completely hide the linearity of the transformation due to their small Hamming weight. Therefore, it is possible to easily identify that two ciphertexts (y_i, y_j) result from the same plaintext because the distance $d(y_i, y_j)$ will be small (inferior to *2t*).

In the case of public key cryptosystems, this attack is described in [Can96] and [Ber97] and it allows the plaintext to be recovered when the attacker gets two or more ciphertexts corresponding to the same plaintext (with high probability the ciphertexts use different error vectors). In order to avoid this attack, several techniques were suggested:

- Introducing randomness as proposed in [BR94]. For example, [Sun98] proposed to change the encryption algorithm to: $y = (x + h(e))E + e$, where *h* is a collision free hash function. The ciphertext *y* is decrypted by decoding yP^{-1}, which gives *e* and $(x + h(e))S$. The plaintext *x* is obtained by applying S^{-1} and then adding *h(e)* to the result;
- Using patterns of errors with higher weight [Loi00].

4.5.5 Secret Key Cryptosystems

The secret-key variants of the McEliece scheme (for example [Jor81], [RN86] and [HR89]) are similar but the code used is kept secret, thus the length of the codes used can be made smaller.

The security of such schemes relies on the secrecy of the code. Therefore, the hardness of the problem of disclosing the secret code from a set of ciphertexts is further analyzed. Each ciphertext is a codeword with errors. In another context this problem was called the Maximum Likelihood Code Recognition Problem [Val99].

Maximum Likelihood Code Recognition

Imagine an eavesdropper catching several encoded messages with errors but without knowing the error correcting code used. The goal is to perform the recognition of the code. This problem was reduced to the decision problem of Rank Reduction, which was proved to be NP-complete [Val99]:

Problem: Decision Rank Reduction

Instance: Let Y be an $N \times n$ matrix formed by N words of length n and k, p nonnegative integers.

Question: Is there a matrix E satisfying $rank(Y + E) = k$ and $w(E) \leq p$? Where the Hamming weight of a matrix denotes the number of its non-null elements.

In the same paper, a probabilistic algorithm to address the problem was devised, but its probability of success decreases heavily with the Hamming weight of the matrix E, being negligible for more than a few errors in the overall matrix.

Linearity

The linearity of the code can be exploited to obtain more efficient solutions to the code recognition problem. This fact is used on the attacks to symmetric key cryptosystems based on the McEliece scheme called Majority Voting and its extensions [MvT91] [SvT87], whose goal is to recover the secret code from the knowledge of several ciphertext values.

Due to the small weight of the error vectors and with several ciphertexts of the same plaintext, it is possible to determine the codeword corresponding

to the plaintext with high probability. By applying this technique a number of times equivalent to the dimension of the code, it is possible to retrieve the secret code. The probability of success of this attack depends on getting a high number l of different ciphertexts for each plaintext. Moreover, this probability decreases exponentially with the length of the code, and the attack has a work factor of $O(knl)$. Hence, the attack is only an obstacle to the use of small codes. For large codes, the attack is unfeasible unless more information about the error patterns can be obtained.

In the case of secret key cryptosystems, in addition to the solutions that use higher weight error patterns to break the linearity of the system [Loi00], we can as well use pattern of errors in a different coset such as in [RN86]. Both alternatives can defeat the majority voting attacks for recovering the secret code.

4.6 Conclusion

This chapter reviewed some hard problems brought up by coding theory and it analyzed their use in cryptography. We showed that after thorough studies, cryptosystems based on coding theory appear as an alternative to cryptography based on number theory. We should emphasize the high level of efficiency of these cryptosystems.

Cryptanalysis revealed some weaknesses of the existing systems that have to be taken into account when designing protocols based on the same intractability assumptions. Our solutions presented in the next three chapters take advantage of the security properties revealed in this chapter.

4.7 References

[ABSS97] Sanjeev Arora, László Babai, Jacques Stern, and Z. Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. *Journal of Computer and System Sciences*, 54(2):317–331, April 1997.

[AM87] Carlisle M. Adams and Henk Meijer. Security-related comments regarding McEliece's public-key cryptosystem. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 224–228, University of California, Santa Barbara, 16–20 August 1987. Springer-Verlag.

[Bar97] A. Barg. Complexity issues in coding theory. Technical Report TR97-046, Electronic Colloquium on Computational Complexity, 1997.

[Ber97] Thomas A. Berson. Failure of the McEliece public-key cryptosystem under message-resend and related-message attack. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 213–220. Springer-Verlag, 17–21 August 1997.

[Bla83] Richard E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, 1983.

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Trans. Information Theory*, IT-24(3):384–386, May 1978.

[BN90] Jehoshua Bruck and Moni Naor. The hardness of decoding linear codes with preprocessing. *IEEE Transactions on Information Theory*, 36(2):381–385, March 1990.

[BO92] E.F. Brickell and A.M. Odlyzko. *Cryptanalysis: A survey of recent results*, chapter Contemporary Cryptology - the Science of Information Integrity, pages 501–540. IEEE Press, 1992.

[BR94] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *Advances in Cryptology—EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995, 9–12 May 1994.

[Can96] Anne Canteaut. *Attaques de Cryptosystèmes à Mots de Poids Faible et Construction de Fonctions t-Résilientes*. PhD thesis, Université Paris VI, October 1996.

[CC98] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 1998.

[Cha96] F. Chabaud. *Recherche de performance dans l'algorithmique des corps finis. Applications à la cryptographie*. PhD thesis, École Polytechnique, 1996.

[CS96] F. Chabaud and J. Stern. The cryptographic security of the syndrome decoding problem for rank distance codes. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology—ASIACRYPT '96*, volume 1163 of *Lecture Notes in Computer Science*, pages 368–381, Kyongju, Korea, 3–7 November 1996. Springer-Verlag.

[CS98] A. Canteaut and N. Sendrier. Cryptanalysis of the original McEliece cryptosystem. In *Advances in Cryptology - ASIACRYPT'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[Gib91] J. K. Gibson. Equivalent Goppa codes and trapdoors to McEliece's public key cryptosystem. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 517–521. Springer-Verlag, 8–11 April 1991.

[Gib95a] J. K. Gibson. Severely denting the Gabidulin version of the McEliece public key cryptosystem. *Designs, Codes and Cryptography*, 6(1):37–45, July 1995.

[Gib95b] Keith Gibson. *Algebraic Coded Cryptosystems*. PhD thesis, University of London- Royal Holloway and Bedford New College, 1995.

[Gib96] Keith Gibson. The security of the Gabidulin public key cryptosystem. In Ueli Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 212–223. Springer-Verlag, 12–16 May 1996.

[Gir90] Marc Girault. A (non-practical) three-pass identification protocol using coding theory. In J. Seberry and J. Pieprzyk, editors, *Advances in*

Cryptology—AUSCRYPT '90, volume 453 of *Lecture Notes in Computer Science*, pages 265–272, Sydney, Australia, 8–11 January 1990. Springer-Verlag.

[GPT91] E. M. Gabidulin, A. V. Paramonov, and O. V. Tretjakov. Ideals over a non-commutative ring and their application in cryptography. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 482–489, Brighton, UK, 8–11 April 1991. Springer-Verlag.

[GS79] M. R. Garey and D. S.Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completness*. San Francisco CA, Freeman, 1979.

[Hei87] R. Heiman. On the security of cryptosystems based on error correcting codes. Master's thesis, Feinberg Graduate School of the Weizman Institute of Science, 1987.

[HR89] Tzonelih Hwang and T. R. N. Rao. Private-key algebraic-code cryptosystems with high information rates (extended abstract). In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology—EUROCRYPT 89*, volume 434 of *Lecture Notes in Computer Science*, pages 657–661. Springer-Verlag, 1990, 10–13 April 1989.

[JM96] Heeralal Janwa and Oscar Moreno. McEliece public key cryptosystems using algebraic-geometric codes. *Designs, Codes and Cryptography*, 8(3):293–307, June 1996.

[Jor81] John P. Jordan. A variant of a public key cryptosystem based on Goppa codes. In Allen Gersho, editor, *Advances in Cryptology: A Report on CRYPTO 81*, pages 25–30. Department of Electrical and Computer Engineering, U. C. Santa Barbara, 24–26 August 1981.

[LB88] P. J. Lee and E. F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In Christoph G. Günther, editor, *Advances in Cryptology—EUROCRYPT 88*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer-Verlag, 25–27 May 1988.

[LC83] Shu Lin and Daniel Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[LDW94] Yuan Xing Li, Robert H. Deng, and Xin Mei Wang. On the equivalence of McEliece's and Niederreiter's public key cryptosystems. *IEEE Trans. on Information Theory*, 40(1):271–273, January 1994.

[Leo88] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. on Information Theory*, 34(5):1354–1359, 1988.

[Loi98] P. Loidreau. Some weak keys in McEliece public-key cryptosystem. In *IEEE International Symposium on Information Theory, ISIT'98, Boston, USA*, 1998.

[Loi00] Pierre Loidreau. Large weight patterns decoding in Goppa codes and application to cryptography. In *IEEE International Symposium on Information Theory, ISIT 2000, Sorento, Italy*, 2000.

[McE78] R. McEliece. A public-key cryptosystem based on algebraic coding theory. In *Jet Propulsion Lab. DSN Progress Report*, 1978.

[MS77] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[MvT91] Joost Meijers and Johan van Tilburg. Extended majority voting and private-key algebraic-code encryptions. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology—ASIACRYPT '91*, volume 739 of *Lecture Notes in Computer Science*, pages 288–298, Fujiyoshida, Japan, 11–14 November 1991. Springer-Verlag. Published 1993.

[Nie86] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[RN86] T. R. N. Rao and Kil-Hyun Nam. Private-key algebraic-coded cryptosystems. In A. M. Odlyzko, editor, *Advances in Cryptology—*

CRYPTO '86, volume 263 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 1987, 11–15 August 1986.

[Sen91] Nicolas Sendrier. *Codes Correcteurs d'Erreurs à Haut Pouvoir de Correction*. PhD thesis, Université Paris VI, December 1991.

[Sen94] Nicolas Sendrier. On the structure of a randomly permuted concatenated code. In *EUROCODE 94*, pages 169–173, Abbaye de la Bussière sur Ouche, France, October 1994.

[Sen97] Nicolas Sendrier. Finding the permutation between equivalent binary code. In *IEEE Symposium on Information Theory ISIT'97*, Ulm, Germany, June 1997.

[SS92] V. Sidelnikov and S. Shestakov. On cryptosystems based on generalized Reed-Solomon codes. *Diskretnaya Math*, 4:57–63, 1992.

[Ste89] J. Stern. A method for finding codewords of small weight. In G. Cohen and J. Wolfmann, editors, *Coding Theory and Applications*, number 388 in Lecture Notes in Computer Science, pages 106–113. Springer-Verlag, 1989.

[Ste93] Jacques Stern. A new identification scheme based on syndrome decoding. In Douglas R. Stinson, editor, *Advances in Cryptology—CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 13–21. Springer-Verlag, 22–26 August 1993.

[Ste94] Jacques Stern. Can one design a signature scheme based on error-correcting codes? In Josef Pieprzyk and Reihanah Safavi-Naini, editors, *Advances in Cryptology—ASIACRYPT '94*, volume 917 of *Lecture Notes in Computer Science*, pages 424–426, Wollongong, Australia, 28 November–1 December 1994. Springer-Verlag.

[Ste96] J. Stern. A new paradigm for public key identification. *IEEE Transactions on Information Theory*, 42(6):1757–1768, November 1996.

[SvT87] René Struik and Johan van Tilburg. The Rao-Nam scheme is insecure against a chosen-plaintext attack. In Carl Pomerance, editor, *Advances*

in Cryptology—CRYPTO '87, volume 293 of *Lecture Notes in Computer Science*, pages 445–457. Springer-Verlag, 1988, 16–20 August 1987.

[Til88] H.C.A. van Tilborg. *An Introduction to Cryptology*. Kluwer Academic Publishers, Boston, 1988.

[Val99] A. Valembois. Recognition of binary linear codes as vector-subspaces. In *Workshop on Coding and Cryptography'99, Book of abstracts*, pages 43–51, Paris, France, January 1999.

[Var97] Alexander Vardy. The intractability of computing the minimum distance of a code. *IEEE Transactions on Information Theory*, 43(6):1757–1766, November 1997.

[Ver95a] P. Veron. A fast identification scheme. In *IEEE Symposium on Information Theory*, Canada, 1995.

[Ver95b] P. Veron. *Problème SD, Opérateur Trace, Schémas d'Identification et Codes de Goppa*. PhD thesis, Université de Toulon et du Var, 1995.

[vT94] Johan van Tilburg. *Security-Analysis of a Class of Cryptosystems Based on Linear Error-Correcting Codes*. PhD thesis, Technische Universiteit Eindhoven, 1994.

Code Protection

5.1 Introduction

This chapter presents solutions for the protection of privacy and integrity of code executed on possibly malicious environments. The solutions we present here do not use Tamper Proof Hardware. The next chapter shows how these solutions can be extended, concretely in terms of computational model, if an auxiliary trusted hardware is available.

The chosen computational model consists in Boolean functions. On one hand, it can be easily related to other models, like Random Access Machines [AHU74]. On the other hand, the model allows for the establishment of a formal framework. The model also makes it possible to link the security of the solutions to known cryptographic assumptions.

The design of solutions places a special emphasis on the efficiency requirement. Apart from security issues, one of the most solid arguments against the use of mobile agents is performance. The mobility property in mobile code platforms often causes the increase not only in the code size but also in communication complexity. Therefore, solutions that inflate the computational and communication complexity are not likely to be adopted in practice. Our solutions are based on the composition of functions. The

properties of functions used in coding theory are exploited in order to efficiently address both privacy and integrity of execution.

The definition of the computational model and the definition of a security framework for mobile code protection are the starting sections of this chapter. Then we provide solutions to Boolean functions. One example focusing on the function representation is also given. Finally, a security evaluation of the solution is described.

5.2 Computational Model

Let $F_{l,k}$ represent the set of all functions from $\{0,1\}^l$ into $\{0,1\}^k$. In the mobile code scenario, a function $f \in F_{l,k}$ owned by Alice is evaluated by Bob on his input data $x \in \{0,1\}^l$. In a protocol that additionally provides privacy of execution, Alice obtains the result $y = f(x) \in \{0,1\}^k$, and Bob provides the input data $x \in \{0,1\}^l$, while preventing the disclosure of f to Bob.
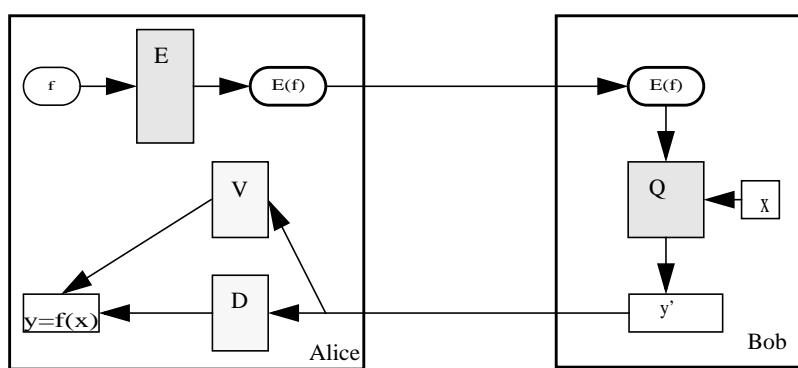
Figure 5.1 Framework for Code Protection

A function $f \in F_{l,k}$ does a conversion between inputs from the set $X = \{0, 1\}^l$ and outputs belonging to the set $Y = \{f(x) | x \in X\}$. Therefore, Y defines a subset of $\{0, 1\}^k$. Integrity of execution allows the function owner to verify in an efficient way that the obtained output y indeed belongs to the set Y.

5.3 Security Framework

A mobile function protection framework is a set of efficient algorithms {E, Q, D, V}, as depicted in Figure 5.1. For the sake of completeness, an algorithm Q was included to represent the function evaluation on the remote host. Such a framework should fulfill the following properties:

Correctness. There is a polynomial time algorithm D to retrieve the result $f(x)$ from y';

Privacy of execution. It is computationally hard to get information about the function f from the function $f'=E(f)$. The execution of algorithm Q should not leak any additional information about f;

Remote host computational complexity. The complexity of evaluating algorithm Q with inputs f' and x should be $O(T(n))$, where $T(n)$ is the worst case running time for f;

Communication complexity. The communication complexity associated with the evaluation of f' should be $O(S(n))$, where $S(n)$ is the worst case communication complexity associated with the evaluation of f;

Round complexity. The scheme should be non-interactive, or should have round complexity equal to two;

Integrity of execution. Using the same notation as defined in chapter 3, Alice receives the result y' of the evaluation of function f' at the point x. With this value, Alice should be able to perform a polynomial time verification of this result, using a verifier V:

if $y \notin Y$ then $P(V(y') = Accept) < \delta$.

Privacy of data. The result y' should not reveal more information about data x to the function owner, than what the actual result y reveals.

5.4 Example

Let us start by considering a set of easy functions from $\{0,1\}^l$ into $\{0,1\}^k$. Concretely, we consider functions where each bit of the output may be represented as a linear combination of the inputs. Such a function f can be represented by a $l \times k$ matrix M.

This matrix will be evaluated by Bob using the data $x \in \{0,1\}^l$. This is equivalent to saying that Bob will perform the vector by matrix multiplication $y = xM$.

We present a simple protocol to address privacy of execution. The aim is to introduce the techniques used later in this chapter.

5.4.1 A Simple Protocol for Privacy of Execution

Let G be a generating matrix for an [n,k,d] Goppa code C and t the number of errors the code is able to correct. Let P be an $n \times n$ random permutation matrix.

Algorithm E. The goal is to achieve the privacy of the function, which means hiding M. Alice calculates another matrix M' as follows:

$M' = MGP$

The M' matrix has size $l \times n$, thus there is an increase in the matrix size inversely proportional to the information rate of the code C.

Algorithm Q. Bob performs the operation:

$$z = xM'.$$

The result z is protected against eavesdroppers by adding a random error vector $e \in \{0, 1\}^n$, with $w(e) = t$. The result $y' = z + e$ is then sent to Alice.

Algorithm D. Alice decrypts the result by computing $y_a = y'P^{-1}$, which is $y_a = yG + eP^{-1}$, and uses C's secret decoding algorithm to retrieve the cleartext result $y = xM$ from y_a. The error vector eP^{-1} is a correctable error vector since $w(eP^{-1}) = t$.

5.4.2 Security Analysis

In this example we focus on the privacy of execution. That is why algorithm V is not mentioned.

Concerning the privacy of execution, one can see that from M', Bob cannot distinguish M from another matrix N built in the following manner:

$N = MR$ and then $M' = NR^{-1}GP$, where R is a $k \times k$ random invertible matrix.

However, the transformation E does not hide the rank of the matrix M. Furthermore, Bob may identify inputs that result in the same output, even if he cannot disclose the output. The case of matrices whose ranks are inferior to the maximum is not very interesting for either player, because they are obliged to handle redundant equations (other than those generated by the code).

Alice does not get more information about x. The number of information bits about x is equal to the rank of the matrix M' which is the same as the matrix M. More important is that Bob can easily check the rank of matrix

M'. Therefore, our protocol satisfies the property of privacy of data from the function owner.

The increase in communication and remote host computational complexity is inversely proportional to the information rate of the code.

5.4.3 Discussion

Although not being mentioned in the requirements section, our protocol addresses confidentiality of data during the transmission of the result back to Alice. The security of this property is equivalent to decoding the code generated by M', which is hard as shown in the previous chapter. The possibility of protecting the data during transmission, with just an addition of an error vector is an advantage in comparison with schemes that use composition with random matrices.

The transformation on the original function (algorithm E) consists of a composition used to create a well defined subspace from the subspace of the solutions of the function. Next, we use the error detecting capability of the code to implement efficient Verifiers (rather than to protect the data during transmission).

5.5 Protocol Description

In this section, we redraw the restrictions that we have imposed on the function in the previous section, and we consider general functions. Each bit y_i of the output y can be represented as a Boolean function f_i on the bits x_i of the input x. For example, in the Algebraic Normal Form [MS77]:

$$y_i = f(x_0, ..., x_{l-1}) = a_o \left(\prod_{i=0}^{l-1} x_i^{u_i} \right) \oplus ... \oplus a_{2^l-1} \left(\prod_{i=0}^{l-1} x_i^{u_i} \right)$$

where $a_i \in GF(2)$ and $u \in GF(2)^l$.

The function $f \in F_{l,k}$ may be seen as a set of k Boolean functions $\{0,1\}^l \to \{0,1\}$. Most part of the Boolean functions have very large combinational complexity [Sav87]. For the sake of simplicity, we consider the number of Boolean outputs as our measure of complexity, rather than the size or depth of the Boolean circuits representing each output bit.

Therefore, a function $f \in F_{l,k}$ is a set of k equations. These k equations are the inputs to algorithm E, and they can be represented in every form, and furthermore they can be optimized. Algorithm E performs linear combinations between these equations, which is easy to compute if the equations are in Algebraic Normal Form.

5.5.1 Error Function - Function e

Algorithm E introduces some errors in the function in order to detect integrity attacks. In order to generate the errors we need to define an error function.

Let us assume that there exists a function $r \in F_{l,m}$ that outputs random n-bit vectors with a given weight t.

Let $e \in F_{l,n}$ be a function, which given an l-bit argument x, returns an n-bit string with weight t, denoted $e(x)$, such that it is computationally unfeasible to distinguish the responses of e from the responses of r.

This means that the function e should be computationally indistinguishable from the set of functions satisfying the weight restriction. This definition tries to adapt the definition of pseudorandom functions [GGM86] to the case where the outputs of the functions have a given weight.

We further assume that is possible to efficiently build functions satisfying these requirements. In our case, this function has to be expressed in terms of each output bit e_i.

In [Sen95], the author proposed an efficient construction for functions that output words of a given weight. However, the indistinguishability from other functions that satisfy the weight requirement was not proven.

5.5.2 Function Transformation - Algorithm E

Alice constructs a new function $f' \in F_{l,n}$, that is a set of n Boolean equations. The y_i' represents the equation corresponding to the *ith* bit of the function f', as follows (all the operations are performed over GF(2)):

$$\begin{bmatrix} y_0' & \cdots & y_n' \end{bmatrix} = \begin{bmatrix} y_0 & \cdots & y_{k-1} \end{bmatrix} GP + \begin{bmatrix} e_o & \cdots & e_{n-1} \end{bmatrix}$$

The new function $f' \in F_{l,n}$ is sent to Bob. G and P are kept secret by Alice.

The construction of f' can be seen as the composition of f with a transformation similar to the one used to construct the public-key cryptosystem of McEliece [McE78]. However, we further include the error function. The set Y' of all the possible outputs of $f' \in F_{l,n}$ defines a subset of $\{0,1\}^n$.

5.5.3 Remote Function Evaluation - Algorithm Q

Bob evaluates f' on his data $x \in \{0,1\}^l$ and gets the result $y' = f'(x) \in Y'$. There is an increase in the number of Boolean outputs while the number of inputs is kept unchanged. The result is then sent to Alice.

5.5.4 Result Retrieval - Algorithm D

Alice decrypts the result by computing $y_a = y'P^{-1}$ or $y_a = yG + z$, and uses C's secret decoding algorithm to retrieve the cleartext result $y = f(x) \in Y$ and the error vector z from y_a. The error vector $z = e(x)P^{-1}$ is a correctable error vector since $w(e(x)) = t$.

5.5.5 Result Verification - Algorithm V

Alice performs the following verification:

If $w(z) = t$ then Accept.

5.6 Function Representation

A small example is given to show how algorithm *E* works. We only perform the multiplication of the function by an error correcting code (matrix *G*). Consider the function $f \in F_{3,4}$ defined as follows:

- $y_0 = \bar{x}_0\bar{x}_1\bar{x}_2 + x_1x_2$

- $y_1 = \bar{x}_0\bar{x}_1x_2 + x_0\bar{x}_1\bar{x}_2 + x_0x_1x_2$

- $y_2 = x_0$

- $y_3 = \bar{x}_0\bar{x}_1x_2 + \bar{x}_0x_1x_2 + x_0x_1\bar{x}_2$

or in the form of truth table:

x0	x1	x2	y0	y1	y2	y3
0	0	0	1	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	1	0	0	1	0
1	1	0	0	0	1	1
1	1	1	1	1	1	0

If we use the following *G* matrix (in systematic form for the sake of simplicity):

$$G = \begin{bmatrix} 1\ 0\ 0\ 0\ 1\ 1\ 0 \\ 0\ 1\ 0\ 0\ 0\ 1\ 1 \\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \end{bmatrix}$$

Then the following $f' \in F_{3,7}$ is defined as follows (after some manual optimizations):

- $y_0 = \bar{x}_0 \bar{x}_1 \bar{x}_2 + x_1 x_2$

- $y_1 = \bar{x}_0 \bar{x}_1 x_2 + x_0 \bar{x}_1 \bar{x}_2 + x_0 x_1 x_2$

- $y_2 = x_0$

- $y_3 = \bar{x}_0 \bar{x}_1 x_2 + \bar{x}_0 x_1 x_2 + x_0 x_1 \bar{x}_2$

- $y_4 = \bar{x}_1$

- $y_5 = \bar{x}_0 \bar{x}_1 + \bar{x}_0 x_1 x_2 + x_0 \bar{x}_1 x_2 + x_0 x_1$

- $y_6 = \bar{x}_0 x_1 x_2 + x_0 \bar{x}_1 x_2$

or in truth table format:

x0	x1	x2	y0	y1	y2	y3	y4	y5	y6
0	0	0	1	0	0	0	1	1	0
0	0	1	0	1	0	1	1	1	0
0	1	0	0	0	0	0	0	0	0
0	1	1	1	0	0	1	0	1	1
1	0	0	0	1	1	0	1	0	0
1	0	1	0	0	1	0	1	1	1
1	1	0	0	0	1	1	0	1	0
1	1	1	1	1	1	0	0	1	0

The first four outputs are identical due to the systematic form of the code. Then, we have to permute the expressions and to add the error function. One

drawback of our solution is that the function must be expressed in a per output bit form.

5.7 Security Evaluation

5.7.1 Complexity

We consider the number of Boolean outputs as our measure of complexity, rather than the size or depth of the Boolean circuits representing each output bit. This simplification relies on the fact that the majority of the possible Boolean functions have very large combinational complexity [Sav87]. Taking this simplification into account, the proposed framework fulfills the requirements in terms of communication and remote host computational complexity.

Algorithm *D* consists of applying a permutation and decoding (knowing the secret code). Thus it is a polynomial time algorithm and our framework satisfies the correctness property.

The privacy of data property is not satisfied by our protocol. Actually, the multiplication by the matrix *GP* just adds linearly dependent bits, therefore not giving more information about *x*. But the error function gives the function owner the possibility of revealing $\log\binom{n}{t}$ bits about *x* (logarithm is base 2).

5.7.2 Privacy of Execution

Privacy of execution relies on the hardness of retrieving the function *f* from its composition with the matrix *GP* and an addition with an error function *e*. We show that for functions and codes of large dimensions, there are many possible solutions to this problem.

For a given *f'*, there are several possible combinations of functions, Goppa codes, permutations and error functions, so an enumeration attack is unfeasible for large codes.

Next, we analyze the hardness of identifying the blocks involved in the composition. Let us start by retrieving the error function *e* from the function *f'*. In another context, the problem of retrieving a subspace from a set of codewords with errors has been called Decision Rank Reduction and its hardness has already been discussed in chapter 4. The best probabilistic algorithm to address the problem has a probability of success that decreases heavily with the weight of the errors. In our solution, we use errors with Hamming weight equal to the maximum correction capability of the code. Therefore, the matrix considered as the input of the algorithm proposed by [Val99] has a large Hamming weight and the probability of success of the algorithm is negligible.

Nevertheless, transformation *E* does not hide everything about the function *f*. Error function *e* does not completely hide the linearity of the transformation due to the small Hamming weight of its outputs. Therefore, Bob can identify inputs (x_i, x_j) that have the same output $y_i = y_j$ because the distance $d(y_i', y_j')$ will be small (inferior to *2t*). Even in the case where an efficient algorithm could be devised for searching this specific case, only the size of *Y* and information about the distribution of the function outputs may be obtained, but not the actual outputs. The complexity of this algorithm depends heavily on function *f*. We discuss in more detail how to avoid information leakage about *f*:

- There is information revealed about the fact that two given inputs have the same output, but retrieving the cleartext value of the output using the attacks described in [Can96] and [Ber97] is not possible because the code is unknown;

- For applications where the information about *f* showing that two inputs have the same output must be hidden, we have to use the more complex transformations described in section 4.5.4. Namely, the proposals of [Sun98] are effective to hide this information, but they imply an increase in the computational complexity of the code owner (algorithm *E* and algorithm *D*);

- The majority voting attack described in section 4.5.5 that exploits the non-deterministic nature of the cryptosystem is not effective for large codes and it requires a lot of different ciphertexts of the same cleartext. In the case of a function f that has many collisions among its outputs, the majority voting attack may be feasible. In order to be resilient to the majority voting attack, we may use higher weight error patterns as described in section 4.5.5.

5.7.3 Integrity of execution

The verifier can accept an invalid y' in the case where $Y \subset \{0, 1\}^k$. On the other hand, if the outputs of function f do not generate the space $\{0, 1\}^k$, the outputs of function f' do not completely define the code (this happens if linear combinations of the output vectors of the function f cannot generate all the space $\{0, 1\}^k$). To our knowledge, there is no efficient algorithm to find a codeword that is not defined by a subpart of a Goppa code. Moreover, the code is scrambled with the errors introduced by the error function. The fact that it was not possible to find an invariant seems to be good evidence that such an algorithm will be difficult to find.

Due to the definition of the error function given in section 5.5.1, we assume it is hard to build relationships between inputs to the error function and the error patterns. Thus, picking a random value $r \in \{0, 1\}^n$ has a probability of success δ that can be easily calculated as:

$$\delta < 2^{k-n} \cdot \binom{n}{t}. \text{ For a code } [1024,524,101] \ \delta < 2^{-215}.$$

Nevertheless, there is a better attack that explores the linearity of the transformation. If an adversary uses a sum of two outputs of the function f', then the probability of success is:

$$\delta = \frac{\binom{t}{t/2} \cdot \binom{n-t}{t/2}}{\binom{n}{t}} \quad , \text{for the same code} \quad \delta < 2^{-73} \;.$$

Using linear combinations between more than two outputs gives a smaller probability of success.

5.8 Conclusion

The aim of our solution was to address the issue of secure evaluation of functions in potentially hostile environments. Both integrity and privacy of execution were addressed, satisfying the requirements described in the framework definition. It must be noted that if an attacker cannot disclose the original function, and if the final result is encrypted, he will not be able to tamper with the function for his benefit.

The security of our solution was related to the hardness of coding theory problems described in the previous chapter. The novelty of the approach consists of using error correcting codes to hide functions instead of encrypting data vectors. On the other hand, the overhead on the communication and on the remote host computational complexity introduced by our solutions is linear.

A drawback of our solution is the limited computational model. In addition, the result is only meaningful for the function owner. In order to perform further computations based on the previous computed results, the help of the function owner is needed. Nevertheless, the function owner can use the technique presented here to address a sequence of functions. This is possible by cascading several transformations. However, the functions have to be executed in a pre-established order. Furthermore, this construction is very cumbersome.

We will show in the next chapter how to address in a more efficient way a model with several functions, but we need a limited Tamper Proof Hardware acting on behalf of the function owner.

5.9 References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Ber97] Thomas A. Berson. Failure of the McEliece public-key cryptosystem under message-resend and related-message attack. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 213–220. Springer-Verlag, 17–21 August 1997.

[Can96] Anne Canteaut. *Attaques de Cryptosystèmes à Mots de Poids Faible et Construction de Fonctions t-Résilientes*. PhD thesis, Université Paris VI, October 1996.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[McE78] R. McEliece. A public-key cryptosystem based on algebraic coding theory. In *Jet Propulsion Lab. DSN Progress Report*, 1978.

[MS77] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[Sav87] John E. Savage. *The complexity of computing*. Krieger, 1987.

[Sen95] Nicolas Sendrier. Efficient generation of binary words of given weight. In Colin Boyd, editor, *Cryptography and Coding; proceedings of the 5th IMA conference*, number 1025 in Lecture Notes in Computer Science, pages 184–187. Springer-Verlag, 1995.

[Sun98] Hung-Min Sun. Improving the security of the McEliece public-key cryptosystem. In *Proceedings of Asiacrypt 98*, pages 200–213, 1998.

[Val99] A. Valembois. Recognition of binary linear codes as vector-subspaces. In *Workshop on Coding and Cryptography'99, Book of abstracts*, pages 43–51, Paris, France, January 1999.

Code Protection with TPH

6.1 Introduction

The solution presented in the previous chapter lacks the possibility of re-using the obtained result in subsequent computations without the help of the code owner. The originality of the model presented here is the delegation of the code owner functionalities (integrity verification and cleartext result retrieval) to tamper proof hardware available at the remote host.

We take advantage of limited Tamper-Proof Hardware (TPH) acting on behalf of the code owner to perform multi-step executions and giving the cleartext result to the remote host. Although the new solution is based on the technique described in the previous chapter, it does not consist of a straightforward substitution of the code owner of the previous scheme for a trusted party located at the remote site. Our solution takes into account the limitations of the TPH, in terms of storage and computational power, while in chapter 5 these requirements were not important to the code owner.

The new solution requires a TPH with limited capacity, such as a smartcard, and it assures the security of the functions executed on untrusted runtime environments by means of some interactions between the remote host and

the trusted hardware. Unlike the scheme described in the previous chapter, the proposed technique allows multi-step execution, as well as the delivery of the cleartext output at the remote site.

This chapter describes the extended computational model and the new requirements imposed by the new scenario. Then, a first solution is described and its security evaluation is discussed. Next, the concept of off-line verifiers is introduced. An off-line verifier allows the integrity verification to be performed after the execution of several functions, instead of at the end of each individual function execution. Finally, an off-line solution is presented and its security is analyzed.

6.2 Computational Model

Our model is a set of Boolean functions $\{f_i | 0 \leq i \leq p - 1 \}$, such that $f_i \in F_{l,k}$, as depicted in the left side of Figure 6.1. The computation of each function f_i depends on the input x_i received from the host and on the input y_i which is the result of the functions that were previously evaluated.

For the sake of simplicity, we assume that every individual function f_i has the same number of inputs and outputs. Without this assumption, we may add bogus equations and variables. We represent the set of functions sequentially, but each function f'_i is independent of the others. Therefore, the order of execution of the functions may change between different executions. So, this computation model allows conditional jumps.

As in chapter 5, the goal of the transformation E is to achieve the privacy of each function f_i. The output of the algorithm E is a set of functions $\{f'_i | 0 \leq i \leq p - 1 \}$, where each $f'_i \in F_{l,n}$. The computation of each function f'_i depends on the input x_i received from the host and on the input m_i calculated from the result y'_i of the functions that were previously evaluated.

For each output of the transformed function f'_i, the Tamper-Proof Hardware should be able to check the integrity of the result and to get the cleartext result in an efficient way as shown in the right side of Figure 6.1.
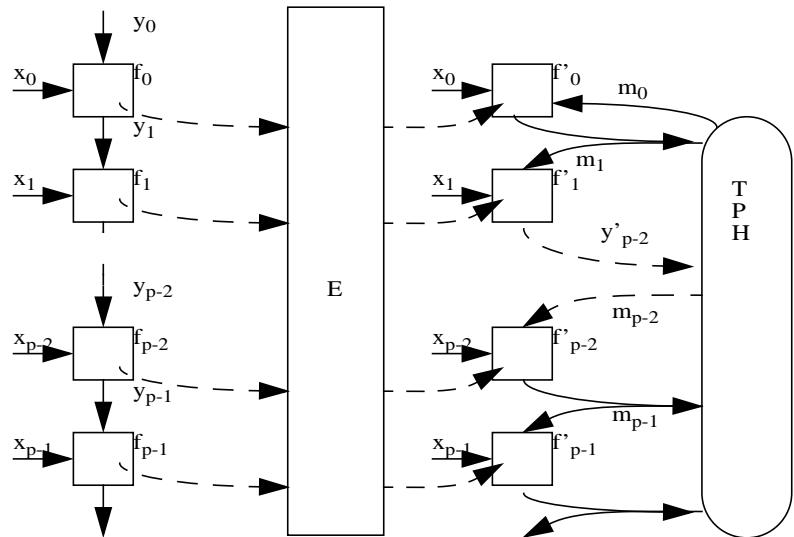
Figure 6.1 Extended model and Algorithm *E*

6.3 Requirements

The requirements defined in chapter 5 have to be adapted to the new scenario. The computational and storage complexity of the algorithms that are executed on the TPH have to be taken into account. Basically, the cleartext result retrieval (algorithm *D*) and result verification (algorithm *V*) will be performed by the TPH. Therefore, we need to reduce the complexity of these algorithms. In order to establish the difference, we will call the new verification algorithm *EV* and the result retrieval algorithm *ED* as depicted in Figure 6.2.
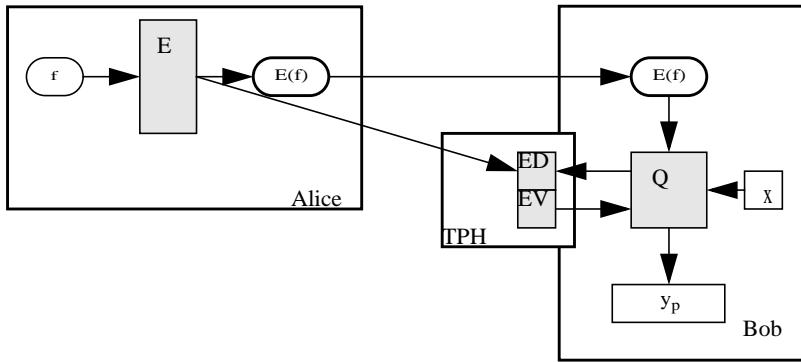
Figure 6.2 Code Protection with TPH

We define the new requirements as follows:

TPH computational complexity. The computational complexity of the algorithms evaluated by the TPH (*ED* and *EV*) must be smaller than evaluating each function f_i itself in the TPH.

Otherwise, we may not justify our solutions, in comparison to the case where the functions $\{f_i | 0 \le i \le p - 1 \}$ are downloaded over a private channel and executed on the TPH. When comparing these two approaches, we should also include the computational complexity related to the transmission of the functions over a private channel between the function owner and the TPH. However, we adopted the more stringent requirement.

The algorithms *D* and *V* as defined in the previous chapter do not fulfill the computational complexity requirement. The same discussion applies to the communication and storage complexity:

Communication complexity. The communication complexity over a private channel implied by the solution should be smaller than the transmission of the functions f_i to the TPH.

TPH storage complexity. The storage complexity imposed to the TPH by the solution should be smaller than the storage of the functions f_i in the TPH.

Each function $f_i \in F_{l,k}$ is a set of k Boolean functions $\{0,1\}^l \rightarrow \{0,1\}$. We consider the number of non-null Boolean outputs as our measure of complexity as in the previous chapter. This simplification relies on the fact that the majority of the possible Boolean functions have very large combinational complexity [Sav87].

Based on the assumption that privacy of the data x from the TPH is not crucial, we use data x as an input to the ED and EV algorithms in order to achieve lower complexity. In addition, we adopt a stronger integrity of execution property that uses the data x. The compromise of the secrets distributed to the TPH results in the very cumbersome task of changing these secrets. This is the reason for the use of a stronger definition.

Integrity of execution. Integrity relies on the hardness of creating a valid input x_i, for an invalid output y'_{i+1}. Using the same terminology as before, we can state this probability as follows:

if $y_{i+1} \neq f_i(x_i|m_i)$ then $P(EV(x_i, m_i, y'_{i+1}) = Accept) < \delta$,

where $y_{i+1} = D(y'_{i+1})$.

6.4 Protocol Description

6.4.1 Error Function

Let q be a security parameter with $0 < q \leq (n - k)$, and let $r \in F_{l,q}$ be a pseudorandom function [GGM86] (r_a represents the *ath* bit of the output of r). We may choose a family of efficient functions, where each function is identified by a seed s.

Furthermore, consider a set R_i constructed by randomly taking q elements from the set *{0,1,...,n-1}*. We define a function $e_i \in F_{l,n}$, $0 \le i \le p - 1$, where $e_{i,j}$ represents the Boolean function defined by the jth bit of the output of e_i ($0 \le j \le n - 1$):

if $j \in R_i$ then ($e_{i,j} = r_a \wedge a = a + 1$) else $e_{i,j} = 0$; With $0 \le a \le q - 1$.

This construction gives $\forall x \in \{0, 1\}^l$, $w(e_i(x)) \le q$. Each function e_i is defined either by q equations (because *(n-q)* are null) or by the seed s_i, and their positions (set R_i).

6.4.2 Function Transformation - Algorithm E

Alice applies the following transformation to the expressions representing each output $y_{i,j}$ $0 \le j \le k - 1$ of each function f_i (all the operations are performed over GF(2)):

$$\left[y_{i,0}' \quad \cdots \quad y_{i,n}' \right] = \left[y_{i,0} \quad \cdots \quad y_{i,k-1} \right] SGP + \left[e_{i,o} \quad \cdots \quad e_{i,n-1} \right]$$

This transformation builds a new set of functions $\{f'_i | 0 \le i < p \}$ that are sent to Bob. Over a private channel, Alice transmits the error functions $\{e_i | 0 \le i < p \}$ to the TPH. *G, P*, and *S* are kept secret by Alice.

6.4.3 TPH Storage Requirements

If we consider the *G* matrix as $(I|A)$, the TPH has to permanently store *A*, *S* and *P*. The matrix *A* has size $k \times (n - k)$. Furthermore, the TPH receives *p* error functions.

Each error function e_i may be represented by a seed s_i. In this case, the TPH must store a pseudorandom function, and the communication complexity

consists of a seed s_i and the set of the error positions for each error function e_i. Otherwise, the functions $\{e_i | 0 \le i < p\}$, where each function consists of q equations and their positions (the set R_i) have to be transmitted.

The construction of the error functions tries to minimize the complexity imposed to the TPH.

6.4.4 Remote Function Evaluation - Algorithm Q

In order to evaluate each function f'_i the following procedure applies:

- The TPH sends m_i to the host;

- The host evaluates the function f'_i at the inputs $(x_i | m_i) \in \{0, 1\}^l$;

- The host sends back the result $y'_i \in \{0, 1\}^n$ and the input data x_i to the TPH.

The initial value m_0 given by the TPH, as depicted in Figure 6.1 is a random seed.

We did not discuss what the relationship between the values y and m was, nor the problem of the storage of these values in memory. These two problems will be the focus of the Chapter 7. The simplest case is when the values y and m are identical and when the TPH has enough memory to securely store the values involved in the computations.

Compared with the case without privacy, there is no modification concerning the inputs x_i given by the host (Bob). There is an increase in the number of outputs in comparison with the original function f_i.

6.4.5 Result Retrieval - Algorithm ED

For each evaluation of a function f'_i the TPH starts by receiving x_i and consecutively computing the following expressions:

- $z_i = e_i(x_i | m_i)$;

- $y_a = y'_i + z_i$;

- $y_b = y_a P^{-1}$;

- y_c becomes the first k bits of y_b (the G matrix is in systematic form);

- $y_i = y_c S^{-1}$.

6.4.6 Result Verification - Algorithm EV

The TPH verifies if $y_b \in C$. In order to perform this, the TPH only requires that the last $n - k$ bits of y_b, called y_d satisfy the equation defined by the generator matrix of the code, as follows:

Is $y_c A = y_d$?

If the answer is affirmative, then the result y'_i is accepted since it is a word at distance z_i from a codeword of the code C.

6.5 Security Evaluation

6.5.1 Complexity

We consider the number of non-null Boolean outputs as our measure of complexity. In our approach, the communication complexity over a private channel is equivalent to the parameters that define the several error functions e_i. Each function is defined by a seed s_i (or q equations) and a set R_i of q elements. In comparison, if the functions f_i are executed on the TPH, the communication complexity is equivalent to the size of all these functions. Therefore, our solution satisfies the communication complexity requirement.

Similar reasoning applies to the computational complexity imposed on the TPH. Nevertheless, the algorithms *ED* and *EV* do not consist of just evaluating the function e_i. Algorithm *ED* has one addition, one permutation and one vector by matrix multiplication. Algorithm *EV* has one vector by matrix multiplication. These operations are less complex than evaluating f_i.

The TPH storage complexity must take into account the secrets required to perform the *ED* and *EV* algorithms (the matrices *S*, *P* and *A* and the description of the pseudorandom function). However, the cost associated with these secrets is amortized during the evaluation of the *p* executions.

6.5.2 Privacy of Execution

The security evaluation of the privacy property is built from the same principles as in chapter 5. In the previous chapter, the security evaluation considered only one function and no cleartext result was given back. Here we will focus on the impact of these modifications:

- We suggested the use of the same code (defined by matrices *S*, *G* and *P*) for all functions. Using the same code with several functions does not significantly impact the security of the overall scheme in terms of enumeration attacks (section 4.5.1) since the number of combinations between all possible functions, codes, permutations and error functions is very high;

- The fact that the result is given back to the untrusted environment is crucial. With sufficient cleartext pairs of inputs/outputs, the remote host is able to interpolate the function f_i. In general, 2^l pairs of inputs and outputs are needed to completely define the function. However, if it is possible to identify inputs that give the same output, as referred in section 5.7.2, then fewer pairs are needed. We may then apply the techniques mentioned in section 5.7.2 to avoid the information leakage about f;

- Because of the importance of having a matrix to hide the systematic form of the code, the additional matrix *S* is used. The functions f_i are partially revealed. Therefore as was shown in [Can96], the matrix *S* is important to avoid the disclosure of the code;

- The fact that, for a given function f_i, the errors always have the same positions can decrease the complexity of the problem of identifying the code. However, the best known probabilistic algorithm presented in [Val99] (discussed in section 4.5.5) cannot take advantage of this fact since the success probability depends on the weight of errors. This success probability is negligible in our case;

- The Majority Voting attacks described in section 4.5.5 are more efficient in disclosing the secret code, because the errors have the same positions for a given function. The complexity of this attack depends heavily on the function f_i. This is the reason why we used different positions for each error function. Furthermore, this attack may be defeated by using higher values for the security parameter q, but the drawback of this solution is its additional complexity.

Nevertheless, the execution sequence of the different functions is disclosed. This calls for techniques that hide the sequence of accesses to a given set of cells, such as the ones described in [GO96]. This issue is further discussed in the next chapter.

6.5.3 Integrity of Execution

We adopted a new implementation of the result verification algorithm *EV*. In our solution, the verification is done using the actual error pattern and not the weight of errors as those described in chapter 5.

The algorithm *EV* accepts a result if y_b is a codeword. This implies that y_a should also be a codeword of the permuted code *GP*. The decomposition of y_a yields:

$$y_a = y_i' + r_i$$

In order for y_a to be a codeword, y_i' has to be at distance r_i of a codeword of the code *GP*. If the host randomly picks a word, then the probability δ is equivalent to the inverse of the number of codewords, i.e. 2^{-k}.

A better attack is to pick a valid output of f_i'. The attack would take advantage of the probability that a valid output $f_i'(x)$, along with an input $x' \neq x$ are accepted by the integrity verification process as a valid pair. Therefore, the integrity of execution can be reduced to:

$$P(e_i(x) = e_i(x')) < \delta$$

The error function e_i is constructed from a pseudorandom function with q bits of output. Therefore this probability is equal to 2^{-q} if we assume that is unfeasible to get information about the error positions. With a reasonable code size (code [1024, 524, 101] and $q=t=50$ for example), this probability is negligible.

However, some of the inputs are provided by the TPH. In fact the verification consists of:

If $y_i' + e_i(x_i'|m_i) = codeword$ then accept

As mentioned before, the best attack is to choose an output of the function, called $f_i'(x_a|x_b)$. Thus, the integrity of execution property is equivalent to:

$$P[e_i(x_a|x_b) = e_i(x_i'|m_i)] < \delta$$

Under the assumption of pseudorandomness of the function used to construct the function e_i, this probability is the same as before.

6.6 Off-line Verifiers

In the previous solution, the verification algorithm *EV* is called each time a result is computed by the untrusted environment. Using an analogy with memory checkers [BEG+94] we call this verifier *on-line*. In order to reduce the computational burden associated to the verification task, we try to

develop solutions which verify a set of results with a single call to the verifier. We call these verifiers *off-line*.

The possibility of doing the off-line verification at the end of all computations in the function owner environment is specially interesting. We present a solution for this scenario where the integrity of execution is verified by Alice afterwards, as shown in Figure 6.3. Our solution provides a simple check value with all the results computed at the remote host.
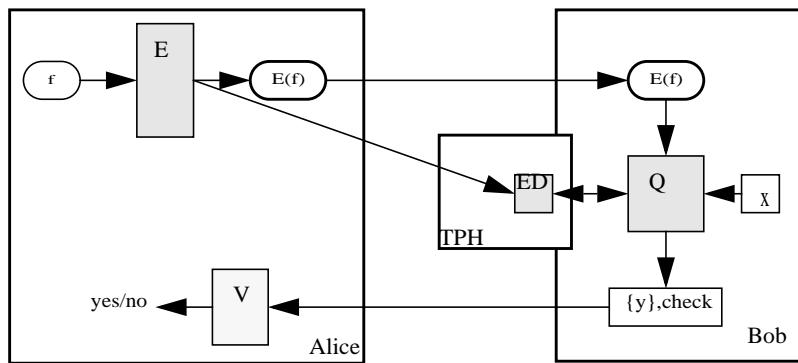
Figure 6.3 Off-line Verifier with verification performed by Alice

6.6.1 Function Transformation - Algorithm E

Let e_i be a function as defined in section 6.4.1, namely satisfying:

$$\forall x \in \{0, 1\}^l, w(e_i(x)) \leq q$$

Alice applies the following transformation to the outputs $y_{i,j}$ $0 \leq j \leq k - 1$ of the function f_i (all the operations are performed over GF(2)):

$$\left[y_{i,0}{}' \; \cdots \; y_{i,n}{}'\right] = \left[y_{i,0} \; \cdots \; y_{i,k-1}\right] SGP + \left[e_{i,o} \; \cdots \; e_{i,n-1}\right]$$

This transformation builds a new set of functions $\{f'_i | 0 \leq i \leq p-1\}$ that is sent to Bob.

Over a private channel, Alice transmits a nonce h_0 and the error functions $\{e_i | \; 0 \leq i \leq p-1 \;\}$ (seeds and positions like in the previous solution) to the TPH. *G, P*, and *S* are kept secret by Alice.

6.6.2 Remote Function Evaluation - Algorithm Q

Each function f'_i is evaluated as follows:

- The TPH transmits m_i to Bob;

- Bob evaluates the function f'_i at the inputs $(x_i | m_i) \in \{0, 1\}^l$;

- Bob sends back the result y'_{i+1} and the input data x_i to the TPH.

6.6.3 Cleartext Result Retrieval - Algorithm ED

For each function execution, the TPH starts by consecutively computing the following expressions:

$$r_i \; = \; e_i(x_i | m_i), \; y_a = \; y'_i + r_i \; \text{ and } \; y_b \; = \; y_a P^{-1}.$$

The first k bits of y_b, called y_c, are multiplied by the matrix S^{-1} revealing the cleartext result y_i.

6.6.4 Off Line Verification

The TPH computes $h_{i+1} \; = \; h(h_i | y_d)$ for each computed function. The last $n-k$ bits of y_b are called y_d. h is a one way collision free hash function [Sti95] and the first hash h_1 is calculated using the nonce h_o provided by Alice.

At the end of the executions, the TPH sends the cleartext results $\{y_i | 0 \le i \le p - 1\}$, and the value h_p to Alice.

Considering the G matrix as $(I|A)$, Alice performs the following operations: $h_1 = h(h_0|y_0 SA)$ and after $h_{i+1} = h(h_i|y_i SA)$ (p times).

At the end, the calculated h_p should be equal to the received h_p.

6.6.5 Security Evaluation

The TPH stores a small value h_i that assures the integrity of all the results. As a result, the TPH does not have to store A, but it has to store the description of the hash function h.

We concentrate on the off-line verification. The addition of the y_d values as the check value does not work due to the linearity of the algorithm E, apart from the addition with the error function. Even though a malicious host may not calculate the error vector r_i, it knows that by giving the correct x_i to the TPH, it will sucessfuly cancel the error r_i included in y'_{i+1}. If the overall integrity value consists of the addition of y_d values, then an attack that adds a pattern of values whose addition is null to the set of the values y'_i would not be detected. This happens because the relationship between the y_d and y_c is not verified for each value, as in our solution. This is the reason for using a collision-free hash function.

If there is a value y_d that does not satisfy the integrity of execution property, then this attack will be detected due to the fact that the received hash value h_p is going to be different from the one calculated with overwhelming probability. The goal of the nonce h_0 is to avoid replay attacks where an overall set of previous values with a valid check value is resent.

We assumed that each function f_i is only calculated once on a sequential order and where all the p functions are executed. Otherwise, the intermedi-

ary results and the sequence of executions have to be transmitted back to Alice in order for her to be able to perform the verification.

6.7 Conclusion

Our solutions provide a fundamental building block for secure computing in potentially hostile environments. The solutions presented apply to a more general computational model, when compared with the previous solutions presented in chapter 5. The difference between the computational models of Chapters 5 and 6 is equivalent to the difference between combinational and sequential circuits. The enhancement of the computation model is possible due to the limited TPH.

The scenario with TPH raises new security requirements. Due to the limitations imposed by a TPH, such as a smartcard, our solutions require only a small function to be downloaded and executed on the TPH, while extending the TPH's intrinsic security to the untrusted environment where the main computations are executed. Furthermore, we presented an off-line Verifier, where a single check value accounts for the integrity of execution of a set of functions.

In our solutions, the error correcting code is only used to create a structure suitable for the integrity verification, while the decoding algorithm is not used at all. The fact that the algorithms *ED* and *EV* are highly coupled is one of the reasons for the high level of efficiency achieved by our solutions.

An advantage of the two solutions presented is that the different functions may be executed in any order because each transformed function is independent of the others. Therefore, our solutions apply to computational models that include conditional jumps.

Nevertheless, the execution sequence of the different functions is disclosed to the remote host (Bob). In addition, we supposed that the TPH has enough memory to store all the cleartext results. These two points will be the focus of the next chapter.

6.8 References

[BEG+94] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, August/September 1994.

[Can96] Anne Canteaut. *Attaques de Cryptosystèmes à Mots de Poids Faible et Construction de Fonctions t-Résilientes*. PhD thesis, Université Paris VI, October 1996.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[Sav87] John E. Savage. *The complexity of computing*. Krieger, 1987.

[Sti95] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 1995.

[Val99] A. Valembois. Recognition of binary linear codes as vector-subspaces. In *Workshop on Coding and Cryptography'99, Book of abstracts*, pages 43–51, Paris, France, January 1999.

Code and Data Protection

7.1 Introduction

In the previous chapters, we focused on the problem of the computation of functions on untrusted environments. In Chapters 5 and 6, we presented solutions that take advantage of interesting properties of error correcting codes.

The solutions presented in Chapter 6 took into account the integrity and privacy of execution, while neglecting the problem of data storage. Therefore, we implicitly considered that secure storage was available to store the data involved in the computations. A limited TPH, such as a smartcard, may not have enough memory to store all the results. Thus we envisage using the memory of a more powerful component. Therefore, we address the problem of secure storage of data in untrusted memories.

We first consider a scenario where the CPU is trusted and the memory is considered untrusted. We define a Secure Memory Manager (SMM) to handle the CPU requests for data storage and retrieval. The SMM performs integrity checks on memory accesses in order to detect data tampering attacks occurring while data resides in memory.

We present solutions for memory protection based on coding theory techniques, since the goal is to later combine these solutions with the solutions developed in the previous chapter for code protection. Next, we consider a model with untrusted memory and untrusted CPU, and we combine the SMM with the solution for code protection presented in Chapter 6 to address the problems raised by this model. We succinctly analyze the characteristics and the weaknesses of this combined solution.

This chapter introduces a technique based on error correcting codes to deal with memory protection. Then, we use this technique to address the protection of the memory space where the data is stored. We present an architecture for code and data protection that combines solutions for code protection from Chapter 6 and memory protection, at the end of the chapter.

7.2 Memory Protection

In this section, the main emphasis is placed on data protection as opposed to the previous chapters that focused on code protection. In the solution in Chapter 6, we implicitly assumed that a secure storage medium was available. Secure storage is needed to save the intermediary results and data structures used during the code execution.

Due to the inherent limitations of storage capacity in most TPH implementations, we suggest an approach for secure storage of data in the untrusted host. In the first solution, we only address the secure storage problem assuming that security of the computations can be assured by other means such as a trusted CPU or the solutions presented in chapter 6. We then present a comprehensive solution that assures both secure storage and secure computation in untrusted environments.

7.2.1 Requirements

Data storage in untrusted memory raises several security requirements, as follows:

- *Privacy.* The data stored in the memory should be protected against unauthorized disclosure;
- *Integrity.* Unauthorized modification of data stored in the memory should be detectable;
- *Access Pattern Protection.* Unauthorized monitoring of memory access operations concerning the location of memory cells should be preventable.

7.2.2 Model

Our memory protection scheme is based on a new component called a secure memory manager (SMM) that is an extension of the memory checker concept introduced by [BEG+94]. The main difference between a memory checker and an SMM is that the checker only addresses integrity, while an SMM also provides privacy. The model consists of a powerful trusted CPU that interacts with an untrusted memory, as depicted in Figure 7.1.
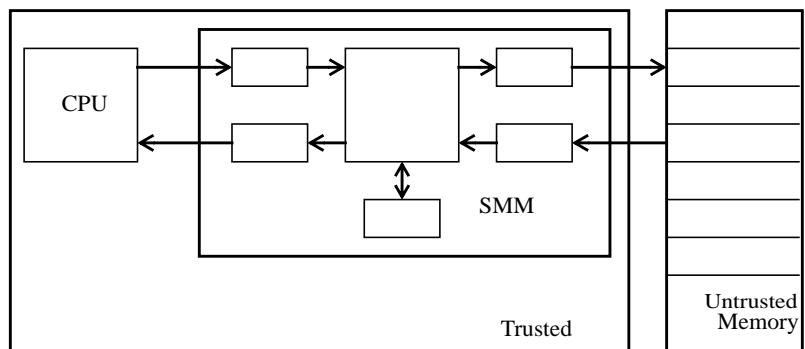
**Figure 7.1 Secure Memory Manager - SMM**

Inspired by the memory checker model, we define a SMM as a probabilistic Turing machine that performs the interface between the trusted CPU and the memory. All the interactions between the CPU and the memory are handled by the SMM. The SMM uses two pairs of tapes to communicate with the CPU and the memory. Each pair of tapes consists of a read-only tape and a

write-only tape. In addition, the SMM includes an internal read/write work-tape of limited capacity that is considered reliable and secret.

The SMM translates the operations requested by the CPU in a way that ensures the protection of the memory contents. The memory is considered as being controlled by a powerful attacker, so the secure memory manager should withstand possible attacks from the memory. As in the case of memory checkers [BEG+94], after reading the operation requested by the CPU, the SMM should answer (with a high probability) with the correct output or "BUGGY" if an error is found in the memory.

If a memory checker verifies the integrity of each operation, it is called *on-line*. Alternatively, if the checker waits until the end of a sequence of operations to report an error, then the checker is called *off-line*. In addition, checkers which introduce operations that imply the storage of checking information in the untrusted memory are called *invasive*. Otherwise, they are called *non-invasive*. We consider the same definitions for the case of the SMM.

The important characteristics of a SMM are:

- The computational complexity of each memory operation performed by the SMM;
- The amount of trusted and secret memory required by the SMM;
- The space overhead introduced by the SMM in the untrusted memory.

7.2.3 Related Work

The proposals for software protection [Gol86][Ost90][GO96] tackle a setting where a very powerful adversary controls the memory. These schemes address all the requirements stated in section 7.2.1. However, the overhead introduced by the techniques proposed to address each of the requirements is high. More efficient solutions that only address the integrity requirement are called memory checkers [BEG+94].

Methods to address the problem of integrity of messages are called Message Authentication Codes (MACs) [Sti95]. It is possible to construct very effi-

cient MACs using hash functions. These techniques are suitable when the message is not dynamic as will be discussed in the next chapter. Nevertheless, when the access pattern of modifications is simple, as in the case of stacks and queues, very efficient solutions for memory checkers exist as described in [DS98]. These solutions rely on the difficulty of finding collisions among the outputs of the hash function.

For the more general case of Random Access Memories (RAMs), the task of designing a memory checker becomes harder. One solution is to use incremental hashing [BGG94]. This technique allows for the hash of the message to be modified when a block of the message changes without having to recalculate the hash from scratch. The incremental hashing and signing technique was designed to withstand virus attacks on files or memory [BGG95]. The technique requires no reliable or private memory. In order to design a checker for RAMs, we need hash functions that are incremental with respect to block replacement, which is simpler and very efficient using XOR MACs [BGR95]. More complex incremental schemes were designed to support delete, insert or even cut and paste operations [BGG95][BM97].

Nevertheless, it is possible to design more efficient checkers [Fis97] [BEG+94], taking advantage of a limited private memory. On the other hand, the technique of fingerprinting [KR87] can also be used as a cryptographic checksum that detects modifications to the data. The fingerprint scheme is incremental with respect to single character replacement and the security of the scheme relies on the assumption that the fingerprint cannot be identified. The performance of this algorithm was analyzed in [Yee94].

The solutions referred to above are suitable for off-line checking because the verification process is cumbersome. The verification implies reading all memory cells. However, the re-calculation of the new integrity check value subsequent to a write operation is very efficient. Conversely, on-line checkers for RAMs verify the integrity each time the data is accessed. Therefore, the integrity check mechanism must be made efficient at the expense of increased complexity to re-calculate the integrity check values.

The main idea behind on-line checkers consists of using individual integrity check values for each position. Then the problem is that previous valid values continue to be valid and thus they can replace more recent values. The

solution presented in [BEG+94] builds a complete binary tree on top of the memory. In order to authenticate a position, all the nodes on the path from the root to the position and their children must be accessed. This is an improvement over the off-line solutions that access all the memory positions (we have to access *2 log n* positions, where *n* is the number of cells). However, when one cell is modified, all the nodes on the path to the root have to be modified.

7.2.4 Message Authentication

We develop solutions for memory protection using coding theory techniques, with the objective of combining these solutions with the solutions developed in the previous chapter. Our solutions are inspired in message authentication techniques.

In the coding theory context, message authentication is based on the idea that the error vector is used to transmit authentication information, instead of being a random string. This approach has been used on coding theory based cryptosystems in [LW91]. The authors described a secret key cryptosystem, where the ciphertext *z* corresponding to message *m* has the following form:

$$z = mSGP + e(m) \, .$$

The function *e* is a secret shared between the two parties involved in the communication. The error vector $e(m)$ provides Message Authentication, due to the relationship between the message and the error vector established by function *e*. A message *m* and an error vector *r* are obtained by the decoding process. The verification consists of evaluating the function *e* with the decoded message *m* as input and checking if *r=e(m)*. Nevertheless, the scheme proposed in [LW91] was broken due to the linearity of the function *e* that generated the error vectors [vT94].

This technique for message authentication may create a potential exposure since some information about the plaintext can be recovered through the error vector, which is not random. In order to avoid this flaw, the error func-

tions should belong to a family of functions satisfying the properties enumerated in sections 5.5.1 and 6.4.1.

Our solutions address both privacy and integrity of RAMs under computational assumptions. The off-line and on-line solutions are original techniques based on the Message Authentication technique described above. The off-line solution is efficient in the sense that the verification process consists of simple XOR operations. Both memory managers are invasive considering the same definition as stated in the case of memory checkers.

7.3 Off-Line Secure Memory Manager

In our solution, the SMM stores in secret memory a check value z, that assures the integrity of all the memory cells. This value is updated each time the CPU requests an operation. Therefore, the SMM translates each memory operation to a set of operations, necessary to keep an updated check value. The verification may be done any time but it is a cumbersome task because it implies accessing all memory cells.

7.3.1 Protocol Description

Let G be a generating matrix for an $[n,k,d]$ Goppa code C and t the number of errors the code is able to correct. Let P be a $n \times n$ random permutation matrix and S be a $k \times k$ random invertible matrix.

Let $f_s \in F_{l,n}$ be a function as defined in section 5.5.1. We assume that this function belongs to a family of functions, therefore each function is identified by a seed s. The seed s is stored in the secret and reliable memory of the SMM. The function f_s satisfies the weight constraint: $\forall x \in \{0,1\}^l, w(f_s(x)) = t$.

The untrusted memory *M* is a set of *N* memory cells *M[i]*, where i represents the address $0 \leq i < N$. Each cell has length *n*. The check value *z* stored in secret memory is initialized to 0, corresponding to an empty memory.

Write Operation

A write operation has two operands: the value *v* to be stored and the address *i* where the value *v* should be stored.

The encryption is performed like in the McEliece scheme, however the error vector e_i is generated in a special way, as follows:

$$e_i = f_s(i, v).$$

The SMM computes the ciphertext: $y = vSGP + e_i$.

In order to write a value *v* in location *i*, the SMM performs the following operations:

• Read the existing value $M'[i]$ stored in the memory location *i*;

• Decrypt $M'[i]$ as in the McEliece scheme obtaining v' and e'_i;

• Xor e'_i to *z*;

• Write *y* to *M[i]*;

• Xor e_i to *z*.

Read Operation

A read operation has only one operand, the address *i*. On a read operation the SMM performs the following operations:

• Read the value $M'[i]$ stored at the address *i*;

• Decrypt $M'[i]$ as in the McEliece scheme obtaining v' and e'_i;

• Verify that $e'_i = f_s(i, v')$.

7.3.2 Off-line Verification

In order to check the integrity of the memory, the SMM performs the following operations for each memory location:

- Read the memory cell $M'[i]$;
- Decrypt $M'[i]$, obtaining v' and e'_i ;
- Verify that $e'_i = f_s(i, v')$.
- Xor e'_i to z.

The verification completes with success if $z = 0$ after performing the above operations over all the memory cells.

7.3.3 Security Evaluation

In a write operation, it is also possible to verify if $e'_i = f_s(i, v')$. Additionally, as in the memory checkers of [BEG+94] a time stamp t_i may be appended to the stored value y and included in the pseudorandom function in order to check that the timestamp read is older than the current time. However, these additional verifications can help detect other attacks but they do not solve the replacement of memory cells by outdated valid values.

The integrity of the memory cells is assured by the difficulty of creating a set of cells that have the same check value z. This value is kept in reliable and secret memory. In addition, the patterns of errors are unknown and it is computationally hard to get any information from them (apart from the low Hamming weight and length). Therefore, the attack described in [BM97] on XOR MACs does not apply to our case because the description of the function f_s and z are secrets.

The integrity attack that consists of substituting a current value $M[i]$ by a value $M'[i]$, such that $M[i] \neq M'[i]$, will require that:

$$f_s(i, v) = f_s(i, v') \text{ with } v \neq v'.$$

The probability of the above statement is $1/\binom{n}{t}$, assuming that all the error patterns are possible outputs of the function f_s and that it is computationally unfeasible to get information about the outputs of f_s. For example, using a code [1024, 524, 101] this probability is negligible. This is also the probability of an integrity attack consisting of switching positions between two different cells.

There is the possibility of exploiting the low weight of the outputs of f_s, that is performing the substitution of *M[i]* by a linear combination of a subset of the other positions. The success probability is equivalent to the probability of finding a subset $U \subset \{0, ..., N-1\}$ such that:

$$\sum_{j \in U} f_s(j, v_j) = f_s(i, v).$$

This is a hard problem because the attacker has to build a specific error pattern. We showed in chapter 5 that matching a given Hamming weight of errors is hard. In chapter 6, we proved that even with limited positions, matching an error pattern has a low probability. Hence the success probability of the problem considered here may be considered negligible.

The same reasoning applies for the deletion or insertion of a set of memory cells. Then, the success probability is equal to finding a set where the sum of the secret error vectors would be null.

The privacy of the data stored in the memory relies on the assumptions defined in chapter 4 concerning the encryption with a secret code. The linearity of the transformation raises the problems discussed in section 4.5.5. If the same value v is stored in two different positions, then information about the error pattern is disclosed. The techniques already mentioned in section 4.5.4 to withstand this attack may be used at the expense of an increased complexity in the SMM.

7.4 On-Line Secure Memory Manager

On-line SMMs for RAMs verify the integrity each time the data is accessed. As a result, the integrity verification must be efficient. The main idea consists of using individual integrity check values for each position.

Our solution for on-line SMM is a combination of the replay detection technique from [BEG+94] and the generic integrity verification technique used with the off-line SMM. The solution from [BEG+94] constructs a binary tree of sums of the timestamps t_i on top of the memory to detect replay attacks. The root of this tree is stored in the secure memory of the checker. Along with the value *y*, a timestamp t_i is also stored in *M[i]*. The timestamp can be a discrete value, like a counter incremented on each operation.

The timestamp tree should be authenticated but without a timestamp. We can just append the value of a pseudorandom function to the timestamp in cleartext as in [BEG+94] due to the fact that the confidentiality of the timestamps is not required.

7.4.1 Write Operation

The CPU asks for a value *v* to be stored in position *i*. The error vector e_i is generated as follows:

$$e_i = f_s(i, t_i, v) \, ;$$

The SMM computes the ciphertext: $y = vSGP + e_i$.

Then, the SMM performs the following operations:

- Read the existing value $M'[i]$ stored in the memory location *i*;
- Verify that t'_i is earlier than the current time;
- Verify that the sum of the values stored in the children leaves on the path from the *i* position to the root is equivalent to the value of the root stored in secure memory;

- Decrypt $M'[i]$ as in the McEliece scheme obtaining v' and e'_i;

- Verify if $e'_i = f_s(i, t'_i, v')$;

- Write y and the current time t_i to *M[i]*;

- Update the sums of the timestamp tree, on the path from the position i to the root.

7.4.2 Read Operation

In order to read address i, the SMM performs the following operations:

- Read the existing value $M'[i]$ stored in the memory location i;

- Verify that t'_i is earlier than the current time;

- Verify that the sum of the values stored in the children leaves on the path from the i position to the root is equivalent to the value of the root stored in secure memory;

- Decrypt $M'[i]$ as in the McEliece scheme obtaining v' and e'_i;

- Verify if $e'_i = f_s(i, t'_i, v')$.

7.4.3 Security Evaluation

The integrity attacks described in the previous section are subject to the same conclusions drawn in section 7.3.3. The main problem with on-line SMM is that previous valid values continue to be valid and thus can replace more recent values. This is known as a replay attack. As stated in [BEG+94], the binary tree is immune to replay attacks due to the fact that replay attacks can only decrease the timestamps stored in the tree. Therefore, the value of the root that is stored in secure memory and that contains the overall sum cannot be satisfied by using previously valid values.

7.5 Code and Data Protection

In the previous chapters, we considered untrusted computations using an implicitly trusted memory. In the previous three sections, we considered

memory protection using a trusted processor. We used coding theory techniques in order to build an efficient architecture for code and data protection.

In this section, we suggest an architecture for code and data protection as depicted in Figure 7.2 by combining the features of the solutions for secure computation in chapter 6 and the SMM. This section mainly presents guidelines and open issues concerning the architecture. To our knowledge, our architecture is the only architecture for code and data protection using an untrusted CPU. The existing proposals ([Bes79],[Ken80],[Yee94] and [GO96]) rely on a trusted CPU.

7.5.1 Architecture

In chapter 6, we extended the inherent security of a limited trusted TPH to the computations performed in a powerful execution environment. Concretely, we addressed the problems of privacy and integrity of execution.

As in the case of the SMM, we model the trusted CPU as a probabilistic Turing machine that uses two pairs of tapes to interact with an untrusted CPU and the SMM. Each pair of tapes consists of a read-only tape and a write-only tape. In addition, the trusted CPU uses an internal read/write worktape of limited capacity. The code is transformed in order be executed on the untrusted CPU, by an algorithm E as defined in chapter 6. The trusted CPU receives the error functions over a private channel and stores them in the internal worktape. The knowledge of the error functions allows the trusted CPU to retrieve the cleartext result and to efficiently verify the integrity of the computation performed in the untrusted CPU, as shown in chapter 6.

Now, the cleartext result that may be used in subsequent computations will be transmitted to the SMM in order to be stored in untrusted memory. The trusted CPU must receive the execution sequence of the functions and the memory operations from the code owner. This information must be protected against integrity attacks and is stored in the internal worktape of the trusted CPU. The SMM translates the operations requested by the CPU in a

way that ensures the privacy and integrity of the data stored in untrusted memory, as defined in the previous two sections.
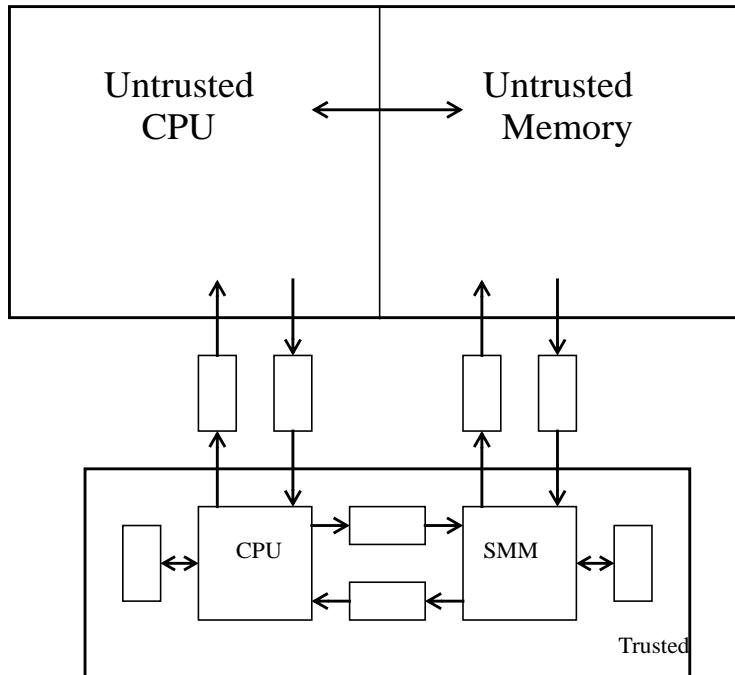

Figure 7.2 Code and Data Protection Architecture

Up to now, we have presented the problems related to the execution on untrusted CPUs and to the storage in untrusted memories separately. Therefore, there is room for enhancement, as follows:

- The secret error-correcting code (matrices S, G and P) used by the trusted CPU and SMM may be the same;
- Algorithm E can take into account the positions where the results are to be stored. By building the error vectors according to this information, we may avoid decryption and re-encryption in order to store a computed value in the untrusted memory. This is not possible if the SMM uses timestamps;

- The integrity verifications performed by the SMM and the trusted CPU may be merged. The integrity of execution property may be verified when the cleartext result is required as the input of a subsequent computation (and after being stored in untrusted memory), rather than immediately after the computation.

The precise evaluation of these enhancements is left open.

7.5.2 Characteristics

The characteristics of such an architecture are:

- Computational complexity of the trusted processor and of the secure memory manager;
- Size of the secure and reliable memory of the trusted processor and of the secure memory manager;
- Increase in the size of the code executed on the untrusted CPU;
- Increase in data stored in the untrusted memory;
- Communication complexity between the trusted and untrusted processors;
- Communication complexity between the secure memory manager and the untrusted memory;
- Information leakage about the program;
- Information leakage about the data stored in untrusted memory.

7.5.3 Security

We presented efficient solutions to code protection (chapter 6) and data protection (chapter 7). However, these solutions conceal information about the code and data stored in the memory, namely the sequence of execution and the access pattern to the memory.

The information concealed by both code protection and data protection can be exploited by an integrated attack. Note that cleartext data is given to the untrusted CPU. The security analysis of the SMMs did not take into account the fact that cleartext data is revealed. The main problem is that information

about the secret code can be disclosed. However, this issue was already discussed in section 6.5.2, where we showed that retrieving the secret code remains a hard problem, even with the knowledge of cleartext data.

Nevertheless, the proposed architecture conceals the access pattern to the memory and the sequence of execution of the functions. In order to hide this information, we must break the link between cleartext data given to the untrusted CPU and ciphertext stored in untrusted memories (without compromising the efficiency of the solutions). In this thesis, we do not present solutions for dealing with the above requirements. However, we give some hints to address this problem, as follows:

- The program structure can be hidden by requesting several computations of functions using bogus data, or by using bogus functions;
- Performing bogus accesses may also be applied to the memory;
- A set of cleartext values can be maintained in the secure and reliable memory, in order to increase the combinations of cleartext data that can be given to the untrusted CPU, without corresponding to a memory read.

These techniques break the relationship between the values m_i and y'_i as mentioned in chapter 6, and illustrated by Figure 6.1. The goal is not to completely hide the sequence of execution and the memory access pattern, but to create a large number of combinations in order to counter attacks exploiting the correlation between ciphertext outputs, values stored in the memory and cleartext data given to the untrusted CPU.

The implementation of these techniques may be easily included in the already defined solutions for code protection and for the SMM. This is possible due to the fact that the protection mechanisms address each function independently. In other words, the protection technique does not rely on a specific sequence of execution of the several functions. In the case of the SMM, the protection mechanisms do not impose any restrictions on the access pattern.

7.6 Conclusion

We have built efficient solutions for the problem of secure storage in untrusted memories. The concept of Secure Memory Manager was defined to address this problem. Our goal was to use techniques that allowed for easy integration with the techniques developed in chapter 6 that address code protection. Therefore, we used memory protection mechanisms based on coding theory. We should emphasize that the secrets used for code protection and memory protection are the same.

The solutions for memory protection are very efficient from the point of view of the required length of the secure worktape. However, we may implement more computationally efficient SMMs using a larger secure memory. For example, we may store the error patterns in secure memory. In this case, the decoding operation is avoided. Such a solution still requires less private storage than the straightforward solution based on storing all the memory cells in secure storage.

Furthermore, we combined the solutions for code protection in chapter 6 and memory protection in order to build an efficient architecture for code and data protection. To our knowledge, the presented architecture is original. It should be added that there are new features such as the integrity and the privacy of execution with the help of a "small" trusted processor.

We mainly presented guidelines and open issues concerning the architecture. The problems of preventing the disclosure of the sequence of execution of the functions and the memory access pattern were not addressed. We just proposed some guidelines to include these features in our architecture. In addition, we showed that providing these features does not imply modifications in the protection mechanisms already developed.

Our architecture can be applied to the following scenarios:

- *Software distribution*. In this scenario, each copy of the software is distributed with a limited TPH. The TPH acts on behalf of the software owner and only customers possessing a valid TPH device are able to run the software. This solution is an alternative to the application hosting

scenario, where a trusted centralized server runs the applications in order to prevent disclosing information about the software;

- *Security management of a network of untrusted CPUs and memories.* One trusted CPU including a limited amount of memory may perform the security management of a network of untrusted computers and memories. The trusted CPU assures code and data protection while the computations and data are distributed among a set of untrusted components (CPUs and memories).

7.7 References

[BEG+94] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, August/September 1994.

[Bes79] Robert Best. Microprocessor for executing encrypted programs. US Patent No. 4,168,396, September 1979.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo G. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer-Verlag, 21–25 August 1994.

[BGG95] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 45–56, Las Vegas, Nevada, 29 May–1 June 1995.

[BGR95] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *Advances in Cryptology—CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, 27–31 August 1995.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *Advances in Cryptology—EUROCRYPT'97*, number 1233 in Lecture Notes in Computer Science, pages 163–192, Konstanz, Germany, May 1997. Springer-Verlag.

[DS98] P. Devanbu and S. G. Stubblebine. Stack and Queue Integrity on Hostile Platforms. In *Proceedings of IEEE Symposium on Security and Privacy*, 1998.

[Fis97] Marc Fischlin. Incremental cryptography and memory checkers. In Walter Fumy, editor, *Advances in Cryptology—EUROCRYPT'97*, number 1233 in lncs, pages 393–408, Konstanz, Germany, May 1997. Springer-Verlag.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[Gol86] Oded Goldreich. Towards a theory of software protection (extended abstract). In A. M. Odlyzko, editor, *Advances in Cryptology—CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 426–439, University of California, Santa Barbara, 11–15 August 1986. Springer-Verlag.

[Ken80] Stephen Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, MIT Laboratory for Computer Science, 1980.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31(2), March 1987.

[LW91] Y. X. Li and X. M. Wang. A joint authentication and encryption scheme based on algebraic coding theory. In H. F. Mattson, T. Mora, and T. R. N. Rao, editors, *Applied Algebra, Algebraic algorithms and Error Correcting Codes 9*, number 539 in Lecture Notes in Computer Science, pages 241–245. Springer-Verlag, 1991.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 514–523, Baltimore, Maryland, 14–16 May 1990.

[Sti95] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 1995.

[vT94] Johan van Tilburg. *Security-Analysis of a Class of Cryptosystems Based on Linear Error-Correcting Codes*. PhD thesis, Technische Universiteit Eindhoven, 1994.

[Yee94] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, School of Computer Science - Carnegie Mellon University, May 1994.

Collected Data Protection

8.1 Introduction

In this chapter, we address a special area of mobile code protection which is the security of the data collected by mobile agents. A typical illustration for such a scenario is a software agent that does product and merchant brokering on behalf of a customer when visiting various merchant systems at different locations. There are several agent-mediated electronic commerce systems already deployed as described in [GMM98]. Secure data collection schemes can also be used to carry out a distributed auction where the mobile agent collects bids from several bidders. Another example is the collection of data in stock markets where data are available at different locations. In these examples the emphasis on security is shifted from the mobile program to the data that is collected from the visited hosts. The collected data is subject to disclosure, modification, and repudiation by single hosts, a group of colluding hosts, or network intruders.

We suggest an original integrity scheme to protect the data submitted to mobile agents by competing hosts visited by the agent. The integrity scheme ensures the protection of each piece of data against tampering by parties other than those at the origin of the data (the host that previously

submitted the data). The integrity of the entire set of data collected by the mobile agent is assured based on an original secure cryptographic mechanism using discrete exponentials. Thanks to the properties of our mechanism, the data collection protocol offers three advantages:

- Each host can update the data it previously submitted without additional memory space;
- The integrity verification mechanism is independent of the sequence of individual data submissions by different hosts;
- The integrity verification is not computationally intensive and does not depend on the number of updates. Moreover, the complexity of the verification does not increase significantly with the number of visited hosts.

The update facility is suitable for commercial competition as required during an auction, and it allows for the collection of large quantities of information that change frequently as in the case of dynamic scenarios such as stock exchange markets. Unlike techniques that rely on appending data, in our case there is no increase in space requirements. The second property of the protocol allows for the collection of data from hosts without any constraint on the itinerary performed by the mobile agent.

The chapter is organized as follows. Section 2 and 3 present the data collection scenario and provide a model for such a scenario. Section 4 formalizes the security requirements. Related work is presented in section 5. The generic cryptographic technique for data integrity is presented in section 6. The data collection protocol based on this technique is described in section 7. Finally, section 8 is devoted to the evaluation of the security properties previously defined.

8.2 Data Collection System

The purpose of a data collection system is to allow mobile agents to travel among hosts of a network to collect individual data segments from these hosts and to return the set of data segments to the originator of the agent. Each data segment collected by the agent can either be the result of some computation by the agent, based on some local input, or simply the input of

some data by the visited host without any processing by the agent. Our security scheme assures the integrity of data segments against tampering and deletion attacks that might originate from a host visited by the agent, a set of colluding hosts or an intruder on the network. The security of the process used to generate the data segments at each host is out of the scope of our scheme, based on the assumption that, even though each host might behave maliciously against other hosts, each host can be trusted with respect to the generation of its own data.

The migration process is another important aspect of the data collection scheme with respect to the security of the collected data. By controlling the migration process, malicious hosts can have a significant impact on the set of data segments collected by the agent. Our data integrity scheme does not address the security of the agent's itinerary. Again, this calls for techniques focusing on the integrity of code execution in untrusted environments as described in chapter 3.
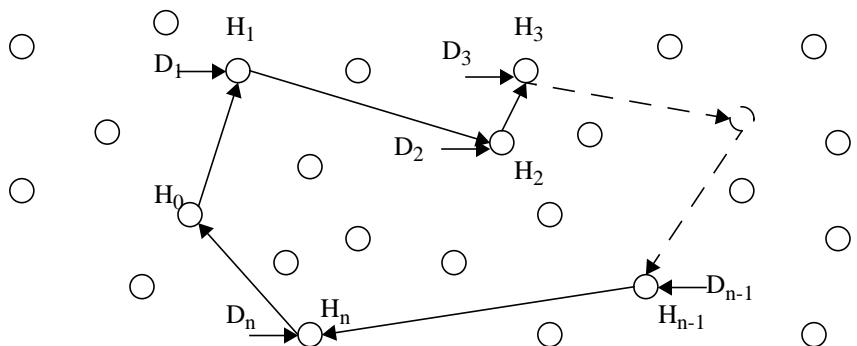

Figure 8.1 Data collection scheme

8.3 Model

In a typical data collection scenario, the mobile agent is generated by its owner H_0 and visits intermediate hosts H_i based on certain migration decisions which are beyond the scope of this thesis. Each intermediate host H_i submits a piece of data D_i as depicted in Figure 8.1. In addition, each host computes an integrity proof value P_i as part of the secure data collection scheme. This value will be integrated on the overall integrity proof value $\Gamma(P)$, included in the agent and computed by the previous host H_{i-1}. The new integrity value $\Gamma(P \cup \{P_i\})$ and the set of data segments collected from the previously visited hosts $D=\{D_0, D_1,..., D_i\}$ are transmitted to the next host.

The data collection scheme allows the agent to visit hosts that were already visited and allows these hosts to update the data pieces they have previously submitted. When the data collection process terminates (or the agent is called back), the agent returns to the agent owner H_0. At this point the agent returns to its originator the set D of data segments collected from all the visited hosts and the final integrity proof value $\Gamma(P)$. The agent owner can then verify the integrity of the data segments in D using $\Gamma(P)$. Table 1 summarizes the components involved in the secure data collection process.

H_0	agent owner
H_i, $1 \leq i \leq n$	visited hosts
D_i, $1 \leq i \leq n$	data collected from H_i
P_i, $1 \leq i \leq n$	integrity proof associated with D_i
D	set of data collected, i. e. $\{D_0, D_1,..., D_i\}$
P	set of integrity proofs, i. e. $\{P_0, P_1,..., P_i\}$
$\Gamma(P)$	integrity proof associated with all the elements of set P

Table 8.1 Data collection components

8.4 Security Requirements

The data collection process is exposed to a number of attacks from network
intruders and legitimate hosts behaving maliciously with respect to compet-
ing parties as explained in [KAG98]. These attacks raise a number of secu-
rity requirements as follows:

- **Data Integrity.** D_i cannot be modified or updated by parties other than
  H_i;

- **Truncation Resilience.** Only the data segments $D_j, i < j < k$, submitted
  between the first malicious host H_i and another malicious host H_k can be
  truncated from the set of data pieces;

- **Insertion Resilience.** No data segment can be inserted unless explicitly
  allowed;

- **Data Confidentiality.** D_i cannot be disclosed to parties other than H_i and
  H_0;

- **Non-Repudiation of Origin.** H_i cannot deny having submitted D_i once
  it was actually included in the set of collected data.

Our definition of the data integrity requirement expands the previous defini-
tions in [KAG98] and [Yee97] in the sense that a host can update the data it
previously submitted. We believe that the update facility is required in free
competition and dynamic commercial environments, like stock markets and
auctions. The insertion resilience property aims at restricting the number of
hosts that can participate thus enabling elementary access control.

Data confidentiality and non-repudiation are not mandatory in all scenarios.
In some real life scenarios like auctions, the confidentiality service might
even be conflicting with the free competition model. It is still an open
research problem how to keep a secret from the execution environment
when this secret has to be used during the agents' trip comprising only
untrusted environments. Therefore, not being able to carry any secret keys,
agents cannot access previously encrypted data during the trip. This implies
that the collected data cannot be used as an input for the negotiation process
for example. Nevertheless, like the protocols proposed in [KAG98], our
data collection scheme can be easily enhanced with data confidentiality

based on public key encryption and with non-repudiation of origin based on digital signatures.

A limitation of data collection schemes based on data integrity mechanisms is the degree of truncation resilience that can be offered. As pointed out in [Yee97], the collusion between two malicious hosts allows for the deletion of the data collected on the path between them by substituting the data set with a copy recorded prior to the beginning of the truncated path. It seems to be an inherent limitation regardless of the way the integrity function is computed in each scheme. Solutions based on time stamps suffer from the probabilistic nature of the network transmission delays. One could also think of assuring the integrity of the migration path in addition to the integrity of the collected data as the solution for the truncation attack. Nevertheless, the integrity of the migration process does not avoid possible collusions between hosts visited to perform a truncation attack, but this can be alleviated by keeping the hosts anonymous. On the other hand, a possible solution is the publication of results after the trip in order to allow visited hosts to verify and complain if needed, as suggested in [KAG98]. This requires hosts to maintain databases with all the data submitted to agents. Alternatively a pre-defined list of hosts with a mandatory submission scheme could also solve the truncation problem. This may be done with explicit empty offers, and the mandatory signalization of the end of participation on the process.

8.5 Related Work

Prior work addressing the integrity of collected data [KAG98][Yee97] uses a technique called hash chaining as the basic mechanism to achieve the integrity of the data pieces submitted by visited hosts. The result of the chained hash computations is the proof of integrity for all the collected data. This technique is suitable to create secure audit logs [BY97] due to its efficiency in terms of computational complexity and size of the integrity proof. In the case of audit logs, it is important to keep a tamper-proof record of all the operations performed in a resilient way for further analysis, so it is important to keep the order of the events.

The data collection scheme suggested by [Yee97] and [KAG98] differs from the solution presented in this thesis regarding various aspects. One of the objectives of the data collection schemes presented in [Yee97] and [KAG98] is to prevent hosts from updating the data they previously submitted. These schemes allow closed bids and include data confidentiality and host anonymity as a basic requirement in order to assure fairness among competing hosts. Host anonymity is a solution to avoid that hosts cannot query the previously visited hosts in order to have access to the submitted data. Our solution in contrast addresses a dynamic scenario including several rounds of competing offers between bidders. As a result, unlike the solutions in [Yee97] and [KAG98], our scheme allows for multiple updates of each data piece by the submitting host. Therefore, data confidentiality and host anonymity are not mandatory requirements for building a fair competition scenario.

Furthermore, hash chaining as a basic data integrity mechanism does not meet the requirements of our data collection scheme. Hash chaining is tied to an implicit sequence among the various data pieces that are protected and the knowledge of the sequence is mandatory for the verification process, that is, the verification process has to compute the hash chain in the same order as the data collection process. In dynamic scenarios such as auctions and stock markets where each data piece may be updated several times, the hash chaining mechanism would require to keep track of all the past values for each data piece. Our scheme is thus based on a novel data integrity technique called a set authentication code that allows for the verification of the most recent value of each data piece without keeping track of past values. With our set authentication code, the computation of the integrity check value and its verification can be performed in random order with respect to the data pieces.

Our scheme aims at a scenario that fosters competition by keeping the information from competing sources in cleartext and by authorizing frequent updates, whereas [KAG98] and [Yee97] assume a rigid scenario based on widespread confidentiality. While our scheme can be easily enhanced with classical confidentiality mechanisms, confidentiality can hardly be suppressed from the solutions in [KAG98] and [Yee97] because of the fairness property between hosts.

On the other hand, on its itinerary from host to host, the agent carries an increasing amount of data that might be exploited by a malicious host. Their confidentiality can be achieved at each step of the agent itinerary by a simple enciphering on the current host, before the agent moves elsewhere. With RSA-based encryption, security will be ensured but the data carried by the agent can be very small: the size of the data padding will then be excessive. Sliding encryption [YY97] aims at retaining equivalent security, using a large key, while at the same time, taking into account the limited storage capacity of an agent. Sliding encryption is aimed at conserving space, which might be of importance for agents that collect small amounts of data on many different hosts. Therefore, this solution addresses a different problem.

8.6 Set Integrity

This section defines an original cryptographic mechanism at the core of the proposed data collection algorithm. We build a "*set authentication code*" using the difficulty of solving the discrete logarithm problem in a finite field, in combination with a classical cryptographic hash function. This mechanism provides a method to authenticate together a set of data segments in an order-independent fashion.

8.6.1 Generator sequences

Let p be a large Sophie Germain prime (also called a strong prime), that is $p = 2q + 1$ where q is also prime. Define g as a generator of the cyclic group *GF(p)*. Then for all x in $X = \{1...(q-3)/2\}$, $g' = g^{2x+1} \bmod p$ is also a generator of the cyclic group GF(p). Hence the following sequence $(G_i)_{i \geq 0}$ is a sequence of generators in GF(p), as depicted in figure 8.2:

$$G_0 = g$$

$$G_{i+1} = (G_i)^{2x_{i+1}+1} \bmod p$$

where $(x_i)_{i > 0}$ is a sequence in X [Kob94].
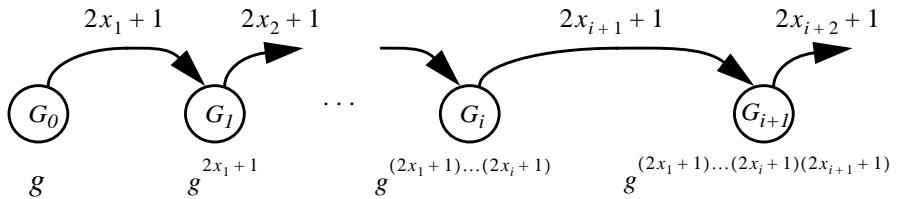
Figure 8.2 Generator sequence

The advantage of this construction is that the result is always a generator, which has the properties referred in the next section.

8.6.2 Properties

- **Property 1 - Security**. With the knowledge of any G_i and G_{i+1}, it is computationally Unfeasible to compute x_{i+1}. Solving for x_{i+1} would require an adversary to solve a discrete logarithm to the base G_i in GF(p) [DH76];

- **Property 2 - Commutativity.** If $G_{i>0}$ is defined by the sequence $(x_j)_{0 < j \leq i}$ then for any $G'_{i>0}$ defined from a permutation of $(x_j)_{0 < j \leq i}$, we have $G_i = G'_i$. This second property is based on the commutativity of the exponent field GF(p-1);

- **Property 3 - Cancellation.** With the knowledge of G_{i+1} and x_{i+1}, it is possible to compute G_i as:

$$G_i = (G_{i+1})^{\frac{1}{2x_{i+1}+1}} \bmod p$$

- **Property 4 - Computational complexity.** With the knowledge of the set $(x_i)_{0 < i \leq n}$, the computation of G_i requires *2n* multiplications, *n* addi-

tions and only *1* exponentiation since:

$$G_i = g^{(2x_1 + 1)(2x_2 + 1)\ldots(2x_n + 1)} \bmod p$$

8.6.3 Set Integrity Function

The combination of *property 2* and *3* allows us to work on sets instead of sequences, thus we define a set function $\Gamma : X \rightarrow GF(p)$ which takes an unordered set of elements in *X* and produces an element in GF(p):

- $\Gamma(\varnothing) = g$

- $\Gamma(P \cup \{x\}) = (\Gamma(P))^{2x+1} \bmod p$

- $\Gamma(P - \{x\}) = \Gamma(P)^{\frac{1}{2x+1}} \bmod p$ *(property 3)*

where $P \subset X$ and $x \in X$.

Let *h* be a cryptographic hash function and $(K_i)_{0 < i \leq n}$ a set of secret keys along with a set $(D_i)_{0 < i \leq n}$ of data segments. Then if we apply Γ to $P = \{P_i = h(D_i | K_i), 0 < i \leq n\}$, we form a set authentication code $\Gamma(P)$, with the following core properties:

- $\Gamma(P)$ cannot be computed without the knowledge of $(K_i)_{0 < i \leq n}$.

- Removing or modifying a data segment D_i from $\Gamma(P)$, while maintaining the integrity of the set, requires the knowledge of the secret value K_i.

This combination of cryptographic techniques allows us to compute a "*set authentication code*" that we will use to securely collect data segments from a set of hosts.

8.7 Data Collection Protocol

In this section we describe our data collection protocol using the set authentication mechanism. Unless otherwise indicated, this section uses the notation of the previous section, whereby h denotes a collision-free hash function (for example MD5 [Riv92]), p is a public prime and "|" denotes concatenation.

Our protocol does not rely on a public key infrastructure. However, a shared key between the source and each of the participant hosts is needed. The generation of the individual integrity proof for each data segment by a visited host requires the knowledge of a secret key shared between the source and the visited host. In order to perform the verification of the global integrity proof, the source has to know all the individual secrets shared with the hosts visited by the agent.

8.7.1 Setup

Each host $H_{i>0}$ exchanges a secret shared key K_i, $i < 0 \le n$ with the source H_0. For example, K_i can be exchanged using the Diffie-Hellman protocol [DH76].

The source H_0 sends an agent to visit a set of hosts $\{H_1, ..., H_n\}$ with an initial *set authentication* value Γ and an empty data collection list D:

- $\Gamma(P) = \Gamma(\varnothing) = g \bmod p$
- $D = \varnothing$

8.7.2 First visit

Each host H_i visited by the agent for the first time, receives:

- $\Gamma(P) = \Gamma(\{P_1, ..., P_{i-1}\})$
- $D = \{D_1, D_2, ..., D_{i-1}\}$

It computes $P_i = h(D_i|K_i)$ and then sends:

- $\Gamma(P) = \Gamma(\{P_1, P_2, ..., P_{i-1}, P_i\})$

- $D = \{D_1, D_2, ..., D_{i-1}, D_i\}$

The submission of an offer is not mandatory for each visited host.

8.7.3 Update

An offer D_j is updated by host H_j to a new value D'_j in 3 steps:

- The old offer D_j is replaced by the new offer D'_j in D;

- An intermediate set authentication value $\Gamma(P')$ is derived from $\Gamma(P)$ by cancelling out P_j;

- The new set authentication value $\Gamma(P'')$ is computed taking into account P'_j.

The first step is straightforward. In the second step, *property 3* is used to compute a new set authentication value $\Gamma(P')$ that does not include $P_j = h(D_j|K_j)$:

$$\Gamma(P') = \Gamma(P - \{P_j\}) = (\Gamma(P))^{\frac{1}{2P_j+1}} \bmod p$$

In the third step, we use the new value $\Gamma(P')$ and update it with $P'_j = h(D'_j|K_j)$:

$$\Gamma(P'') = \Gamma(P' \cup \{P'_j\}) = (\Gamma(P'))^{2P'_j+1} \bmod p$$

8.7.4 Verification

Once the agent goes back to the source H_0, the source can verify the integrity of the set of offers by using *property 4* to check that:

$$\Gamma(P) = g^{(2P_1 + 1)(2P_2 + 1)...(2P_n + 1)} \bmod p$$

where $\Gamma(P)$ is the integrity check value received by the source.

If this condition fails, none of the offers are considered as valid. It should be noted that the cost of verification is much lower than the cost of generation of $\Gamma(\{P_i | (0 < i \leq n)\})$. During the data collection process, the submission of each data piece D_i requires the computation of a discrete exponentiation whereas the verification of the integrity value for the entire set of data requires only a single exponentiation.

8.8 Security Evaluation

This section focuses on the evaluation of the security properties defined before for secure data collection schemes.

8.8.1 Data Integrity

Due to the shared secret, each segment of collected data can only be modified by its originator. Tampering with a data segment or unauthorized modification thereof by intruders will be detected by the source. Data modification attempts by an intruder may consist of the update of a data segment D_i with a new value D'_i generated by the intruder or of the replacement of the current data segment with an old value that was previously submitted by its legitimate origin. Both types of modification attempts would require the intruder to first cancel out the integrity proof P_i of the current data segment from the global integrity proof $\Gamma(P)$. Computing P_i from the actual data segment D_i requires the knowledge of the secret K_i shared

between the origin of the data segment and the source. Appending a new data segment computed by the intruder similarly would require the knowledge of the shared secret K_i. The intruder is thus unable to either get an old value of P_i or to compute a new one that is valid without knowing the shared secret. Another possible attack consists of retrieving past values of P_i computed by the origin of the data segment. A possible modification attack would consist of deriving P_i from $\Gamma(P)$ and $\Gamma(P \cup \{P_i\})$. That would require the computation of a discrete logarithm which is known to be as computationally unfeasible.

8.8.2 Truncation Resilience

Truncation of one or several data segments from a valid offer by a single intruder can be reduced to the data integrity problem. Thus truncation resilience relies on the data integrity property. As already discussed in section 8.4, collusion can result in the truncation of data submitted by all the hosts visited on the path between two colluding hosts.

8.8.3 Insertion Resilience

No data can be inserted by unauthorized parties because only hosts H_i sharing a secret key K_i with the source can generate a valid integrity proof P_i.

8.8.4 Confidentiality and Non-Repudiation

As explained in section 8.4, data confidentiality and non-repudiation are not considered part of mandatory requirements since the main purpose of our security scheme is data integrity in a free competition environment. Nonetheless, these missing features can easily be retrofitted in the data collection scheme using classical data encryption and digital signature mechanisms. For example, data confidentiality can easily be ensured, encrypting the data with the shared key.

8.9 Conclusion

In this chapter, we described an original protocol to protect dynamic data collected by mobile agents when roaming through a set of hosts. We only considered perfectly autonomous agents, i. e., without any communication with the source or with some kind of trusted party.

Unlike prior work, our protocol allows hosts to update their own submissions without keeping track of past values and to submit data in a random order thanks to the original set authentication technique. Our protocol does not rely on a public key infrastructure. However, a shared key between the origin and each host is needed.

This technique also allows the source to verify the integrity of all the collected data segments without knowing the sequence of data submissions by each host. Moreover, the size of the integrity proof is small and independent of the number of hosts or updates, and the verification is not computationally intensive.

8.10 References

[BY97] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, UC at San Diego, Dept. of Computer Science and Engineering, nov 1997.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transaction on Information Theory*, 22(6):644–654, November 1976.

[GMM98] R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, June 1998.

[KAG98] G. Karjoth, N. Asokan, and C. Gulcu. Protecting the computation results of free-roaming agents. In Kurt Rothermel and Fritz Hohl, editors,

Proc. of the Second International Workshop, Mobile Agents 98, pages 195–207, 1998. Springer-Verlag Lecture Notes in Computer Science No. 1477.

[Kob94] Neal Koblitz. *A course in number theory.* Springer-Verlag, 1994.

[Riv92] Ronald L. Rivest. The MD5 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1321.

[Yee97] Bennet Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC at San Diego, Dept. of Computer Science and Engineering, April 1997.

[YY97] A. Young and Moti Yung. Sliding encryption: a cryptographic tool for mobile agents. In *Proc. 4th International workshop fast software encryption 97*, 1997. Springer-Verlag Lecture Notes in Computer Science No. 1267.

CHAPTER 9 *Conclusion*

Mobile code is generally justified on the grounds of greater efficiency and increased flexibility even if these features have not been fully exploited yet. However, this flexibility does not come without a price: there is an increased exposure to security threats.

Among various security concerns raised by mobile code, we addressed the ones related to malicious interactions between mobile code and the runtime environment. We considered two different types of protection mechanisms to counter possible malicious behaviors:

- *Host protection* aiming at preventing harmful operations caused by a malicious mobile code on the resources of the runtime environment;
- *Mobile code protection* required to assure the privacy and integrity of mobile code against possible attacks from a malicious execution environment.

Based on recent research developments, we believe that a high level of host protection is achievable. There is currently a trend towards enabling finer grained access control schemes. The deployment of these schemes will probably enable widespread applications of mobile code in the near future.

The protection of a mobile code against a malicious host is an open research topic. The problem as a whole is considered hard and viewed as unfeasible by some researchers. *We distinguish the problems related to code protection and those related to data protection*.

The problems related to mobile code execution on an untrusted runtime environment are quite atypical. We analyzed two aspects of this problem: *privacy of execution*, treated in Chapter 2 and *integrity of execution* treated in Chapter 3. Privacy of execution aims at preventing the disclosure of the code semantics during the execution of the code on an untrusted run-time environment. Integrity of execution focuses on the correctness of code execution. An exhaustive study of possible approaches to these two problems was explored. In Chapters 2 and 3, we provided comprehensive surveys of the above problems. These surveys are comprehensive in the sense that they include approaches ranging from theoretical computer science to practical solutions.

Concerning *privacy of execution*, there are solutions offering strong security in the field of secure function evaluation and multiparty computations. However, these solutions are far from practical use due to their complexity and their limited coverage. Additionally, the overhead imposed by these solutions may compromise the fragile advantages of the mobile code paradigm. Namely, the complexity of non-interactive solutions for privacy is prohibitive for practical applications. Non-interactivity is a mandatory feature of autonomous mobile code. On the other hand, practical solutions such as code obfuscation suffer from the lack of solid assumptions on which security can be based. *Solutions for protecting the privacy of mobile code against the execution environment are still in their infancy.* The definition of a set of requirements for the design of solutions tackling privacy of execution was the final contribution of Chapter 2.

Integrity of execution is another crucial requirement for the affirmation of the mobile code paradigm. Without having a way of preventing or at least detecting the correctness of the execution of mobile code, applications appear to be restricted to those involving only trusted parties. Once again, the field of theoretical computer science has developed several techniques to address this problem. The idea behind these techniques is to provide a proof of correctness together with the result, with the objective of making

the integrity verification easy when compared to the complexity of the computation of the proof. More practical approaches provide a log of execution with the result. In the field of result checking, we found very interesting solutions. However, they did not take into account the malicious behavior of the execution environment. Another approach is to perform redundant computations and to elect the result on a voting basis. *We proposed a different way of expressing the requirement of integrity of execution* in order to achieve more efficient solutions.

We believe that integrity of execution is necessary but not sufficient without privacy for certain applications. When a malicious host can re-execute the code an unlimited number of times, even without tampering or reverse engineering the code, it can choose the best result from the set of results. Applications such as bargaining motivated this reflection. In order to develop solutions for these applications *we need privacy and integrity of execution.*

We used the class of Boolean functions as a *computational model* to design our solutions for privacy and integrity of execution. Based on this model the security of the solutions could be related to well known computational assumptions. Unlike other models that are suitable for formal security evaluation, Boolean functions allow for the representation of realistic computational models, such as Random Access Machines.

The *cryptographic tools* chosen to address the code protection problem were found in the field of coding theory. The main idea was to construct programs that were resilient to execution errors, as data may be protected against transmission errors using error correcting codes. On the other hand, coding theory includes hard instances of problems that are used to build cryptosystems. The study of the advantages and weaknesses of these systems in Chapter 4 revealed interesting properties while providing guidelines for the design of secure solutions.

We defined the properties that a framework for mobile code protection should satisfy. Namely, we need non-interactive and efficient solutions. *We developed original solutions to mobile code protection without TPH.* These solutions, presented in Chapter 5, addressed both privacy and integrity of execution. Nevertheless, the solutions did not allow multi-step applications

to be performed in a non-interactive scenario. Thus, the solution was bound to a single function execution (without interaction with the code owner).

We extended the solution to a more powerful computational model using a limited TPH in Chapter 6. The possibility of using a limited TPH in the remote host acting on behalf of the function owner was considered as one way to overcome the limitation of the solution developed in Chapter 5. Furthermore, *we considered the limited capacity of the TPH*. The objective was not to execute the functions on the TPH, but to perform the computations on the untrusted host while having a "small" fragment of the computation performed in the TPH in order to verify and "complete" the untrusted computation. In other words, the aim is to extend the intrinsic security of the TPH to the overall environment. In addition, the solutions must be proved less complex when compared to the case of downloading the code into the TPH and executing the code there. In the solutions presented in Chapter 6, a small piece of code is downloaded into the TPH that allows the integrity verification and the calculation of the cleartext result to be performed while the main computation remains on the untrusted environment. *In the first scheme, the TPH verifies the integrity of the result each time a computation is done. We proposed another solution where the verification of the results is done off-line* (at the end of a number of computations) in order to enhance efficiency. This solution is especially interesting to construct an overall verification value transmitted back to the function owner. This verification value can be considered an efficient proof of computation.

In Chapter 6, we implicitly assumed that a secure storage medium in the trusted TPH was available. Due to the limited storage capacity of a TPH, such as a smartcard, we thought of achieving secure storage within the untrusted host. In Chapter 7, *we suggested a scheme based on the memory checker concept for the protection of the data stored in untrusted memory.* Memory checkers deal with the problem of integrity of the data stored in untrusted memories. We extended this concept by including data privacy in the set of requirements. *We defined the concept of a secure memory manager*, which is another contribution of this dissertation. In addition, *we proposed an original architecture for code and data protection* that combined the techniques of memory protection (Chapter 7) and of code protection (Chapter 6). As opposed to our architecture, the existing solutions for software protection have always considered the computation as trusted. How-

ever, our architecture is not fully defined, leaving some security requirements as open research topics.

Finally in Chapter 8, we turned our attention to the problem of data protection in a specific scenario. W*e proposed a solution for data protection in a scenario where mobile code gathers data among a set of hosts*. The goal is to protect collected segments against attacks carried out by competing hosts. Our solution has the interesting feature of allowing updates of the data segments already submitted by a given host, while preserving the integrity of all the data segments with a small authentication value. Therefore, our solution is very efficient in competitive scenarios, such as distributed auctions and comparative shopping. *This last solution shows that it is possible to address security of specific applications in an efficient way using more traditional cryptographic tools*.

Mobile code protection raises a large number of new and atypical security issues that are far from being solved in practical terms, but we believe to have given some interesting contributions and we are optimistic about a bright future for the field of mobile code protection.

9.1 Unanswered Questions

In this section, we present some possible directions for further research, as follows:

- An implementation of the error function, which outputs words of a given weight was not presented. The construction of efficient error functions offering strong security properties would be an important addition to our solutions;

- We envisage using more complex function transformations in order to achieve stronger security properties. For example, using several error correcting codes;

- The proposed architecture for code and data protection was not fully defined. We did not address the issue of hiding the memory access pattern, and that of hiding the sequence of computations;

- We envisage extending the proposed solutions to practical programming languages. We are thinking about automatic tools such as obfuscators, but where the security of the transformations applied to the code are related to cryptographic problems, as in our schemes for code protection.

Remark: At the end of writing this thesis, the problem of giving the cleartext result back to the remote host was addressed in:

J. Algesheimer, C. Cachin, J. Camenisch and G. Karjoth. Cryptographic Security for Mobile Code. To appear in 2001 IEEE Symposium on Security and Privacy, May 13-16, 2001.

Publications

S. Loureiro, R. Molva
Mobile Code Protection with Smartcards
Proceedings of the ECOOP 2000 workshop on Mobile Object Systems
Sophia Antipolis, France, June 13th, 2000

S. Loureiro, R. Molva, Y.Roudier
Mobile Code Security
Proceedings of the ISYPAR 2000
(4ème Ecole d'Informatique des Systèmes Parallèles Répartis)
Toulouse, France, Feb 1-3, 2000

S. Loureiro, R. Molva, A.Pannetrat
Secure Data Collection with Updates
Special issue "Agents in Electronic Commerce" of Electronic Commerce
Research Journal. Yimming Yee and Jiming Liu, editors. To appear.
(Preliminary version in Proceedings of the Workshop on Agents in Electronic Commerce, First Asia Pacific Conference on Intelligent Agent Technology, pages 121-130, Hong-Kong, December 1999)

S. Loureiro, R. Molva
Privacy for Mobile Code
Proceedings of the OOPSLA'99 Workshop on Distributed Object Security, Denver, US, November 1999

S. Loureiro, R. Molva
Process for securing the execution of a mobile code in an untrusted environment. European Patent Application 99480057.1, July 1999

S. Loureiro, R. Molva
Function Hiding based on Error Correcting Codes
Manuel Blum and C.H. Lee, editors, Proceedings of Cryptec'99 International Workshop on Cryptographic Techniques and Electronic Commerce, pages 92-98, Hong-Kong, July 1999