SECURE DELEGATION OF COMPUTING AND DATA IN UNSAFE DISTRIBUTED SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER ARCHITECTURE
AND THE COMMITTEE ON GRADUATE STUDIES
OF UNIVERSITAT POLITÈCNICA DE CATALUNYA
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR

By
Ricardo Xavier Sánchez del Castillo
December 1998
I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor.

Josep Domingo-Ferrer  
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor.

Manuel Medina Llinàs  
(Tutor)

Approved by the University Committee on Graduate Studies:
Preface

This thesis is about secure computation and all related subjects that can help to understand and define the problem of computing in an insecure environment. In the last ten years’ literature a lot of different scenarios can be found where for each scenario a specific construct is proposed. A lot of work has been done trying to characterize the different proposals. To summarize, it can be said that all the different scenarios are equivalent to the problem of shared computing, or at least to some variation of this problem. This work focuses in the extreme case when one party wants to maintain control on the disclosure of his data, but at the same time wants another (untrusted) party to be able to perform some computation on those data.
Acknowledgments

Many thanks go to all the people and organizations who supported and encouraged me when working on this thesis. My sincerest gratitude goes to:

Josep Domingo-Ferrer for having inspired and advised this thesis and for his continuous encouragement, essential to the completion of this work; Manuel Medina Llinàs for his support and friendliness; Javier Castála Díaz for his help in building and documenting Dikê; Anna Escudero i Quesada for her assistance in the final stage of this work; Jordi Herrera i Joancomartí for his availability; Josep Casco Robledo, for his clever insight and practical advice.

Last but not least, I must thank my parents Isabel and Joaquim, and my sisters Montserrat, Desirée and Yolanda, for their unconditional support. Also Piedad and Enric for being just friends. Most of all, I would like to thank my wife Luciana for more things than I can express.

This work was partly funded by the Institut d’Estadística de Catalunya through two research contracts and by the Comisión Interministerial de Ciencia y Tecnología (CICYT) through grant no. TIC95-0903-C02-02.
Contents

Preface 5

Acknowledgments 7

1 Introduction 17

2 Delegation in Distributed Systems 19
   2.1 Distributed systems security ......................... 20
   2.2 Delegation concepts .................................. 22
   2.3 Requirements for secure delegation ................... 23
   2.4 Available security functionalities ................... 24

3 Cryptographic Background 27
   3.1 Secure multi-party computation ...................... 27
       3.1.1 Definitions .................................. 29
       3.1.2 A historic overview ........................... 31
       3.1.3 Two-party computation: an example protocol .... 32
   3.2 Computing with encrypted data ......................... 34
       3.2.1 Definitions .................................. 34
       3.2.2 Examples of encryptable functions .............. 35
   3.3 Delegation by fragmentation, dissemination and replication .. 38
   3.4 Ad-hoc procedures for adding encrypted data ........... 39
   3.5 Privacy homomorphisms ................................ 40
   3.6 Comparison and interim discussion ................... 42


### Table of Contents

#### 4 A Secure Solution for Delegation
- 4.1 Need for secure delegation .............................................. 45
- 4.2 Requirements and protocols for delegation .......................... 46
  - 4.2.1 Data confidentiality .................................................. 48
  - 4.2.2 Computation verifiability .......................................... 49
- 4.3 Verification of computation through parity checking ............... 50
- 4.4 A new privacy homomorphism .......................... 53
  - 4.4.1 Specification of the new PH ........................................ 53
  - 4.4.2 Numerical example .................................................. 55
  - 4.4.3 Security of the new privacy homomorphism ..................... 57
  - 4.4.4 Suitability of the new PH for secure delegation ............... 64
  - 4.4.5 Summary .............................................................. 65

#### 5 Implementation Issues for Secure Delegation
- 5.1 Toward distributed computing .............................. 68
- 5.2 The client-server paradigm ............................................. 68
  - 5.2.1 Sockets ............................................................. 69
  - 5.2.2 Remote Procedure Calls .......................................... 70
  - 5.2.3 Distributed objects ............................................... 71
- 5.3 Distributed object technologies ..................................... 73
  - 5.3.1 The DCOM architecture ............................................ 74
  - 5.3.2 The CORBA architecture ......................................... 75
- 5.4 Comparison and interim discussion .................................. 76
- 5.5 Security in CORBA ....................................................... 78
  - 5.5.1 Secure socket layer ............................................... 79
- 5.6 Auxiliary implementation choices .................................... 80

#### 6 A Prototype for Secure Delegation
- 6.1 Requirements ............................................................. 84
  - 6.1.1 Non functional requirements ...................................... 84
  - 6.1.2 Types of users ...................................................... 85
  - 6.1.3 Statistical data ..................................................... 85
6.1.4 Privacy homomorphisms .......................... 86
6.1.5 Encryption of tables ............................. 86
6.1.6 Generic concerns ................................. 87
6.2 System architecture ................................. 88
6.2.1 Use cases view .................................. 89
6.2.2 Design view ..................................... 96
6.3 Implementation notes ............................... 103

7 Conclusions and Future Work ............................ 105
7.1 Main Results ........................................ 105
7.2 Future Research ..................................... 106

A The Unified Modeling Language (UML) .................. 109
A.1 Modeling using UML ................................. 109
A.2 The UML notation .................................. 110
A.3 Use case diagrams .................................. 111
A.4 Structure diagrams .................................. 111
A.4.1 Class diagrams ................................... 111
A.4.2 Object diagrams .................................. 112
A.5 Behavior diagrams ................................... 112
A.5.1 Statechart diagrams ............................... 112
A.5.2 Activity diagrams ................................. 112
A.5.3 Interaction diagrams ............................... 112
A.6 Implementation diagrams ............................. 113
A.6.1 Component diagrams .............................. 113
A.6.2 Deployment diagrams .............................. 114

B The CORBA Architecture ............................... 115
B.1 Architecture ........................................ 115
B.2 The IDL specification language ..................... 116
  B.2.1 Object description .............................. 117
  B.2.2 Signatures ...................................... 117
B.2.3 Types ........................................... 118
B.3 Using CORBA .................................... 118

C Basic Cryptologic Definitions 121
C.1 Cryptosystems .................................... 121
   C.1.1 Shared-key encryption .................... 121
   C.1.2 Public-key encryption .................... 122
C.2 Complexity ...................................... 122
   C.2.1 Complexity classes and standard definitions ....... 123
C.3 Cryptography definitions ..................... 125
   C.3.1 One-way functions .......................... 125
   C.3.2 Trapdoor one-way functions ............... 125
   C.3.3 Examples of one-way functions .............. 126
C.4 Cryptanalysis .................................. 128
   C.4.1 Attacks ..................................... 128
   C.4.2 Cryptanalyst behavior .................... 129
C.5 Number theory .................................. 129

Bibliography 131
# List of Tables

4.1 Example parameter choices for the proposed privacy homomorphism. .................................................. 64

6.1 Actors and use cases of the system .................................................. 90

6.2 Scenarios associated with the collaboration of the Manage Data use case .................................................. 93

6.3 Scenarios associated with the collaboration of Use Data use case .................................................. 94

6.4 Scenarios associated with the collaboration of Manage System use case .................................................. 99
List of Figures

2.1 Computing delegation scheme .................. 23
2.2 Data delegation scheme ........................ 23
3.1 Multi-party computation scheme ............... 28
3.2 Computing with encrypted data ................ 35
4.1 Secure computing delegation scheme .......... 47
4.2 Secure data delegation scheme ............... 48
5.1 Typical sequence diagram for the socket approach .......... 70
5.2 Typical sequence diagram for the RPC approach ........ 71
5.3 Typical sequence diagram for the DO approach ......... 74
6.1 Graphical representation of the different actors .......... 90
6.2 Use case diagram of the Dikê system ............. 92
6.3 Interaction diagram for the Login scenario of the Use Data use case .......... 95
6.4 Interaction diagram for the Compute Local scenario of the Use Data use case ........ 96
6.5 Interaction diagram for the Compute Remote scenario of the Use Data use case ....... 97
6.6 Activity diagram for the Add User scenario of the Manage System use case .......... 98
6.7 StatService class diagram ........................ 99
6.8 Interaction diagram for the client-server-administrator scenario . 100
6.9 Client class diagram .............................. 102
B.1 CORBA reference scheme .......................... 117
B.2 IDL use scheme ................................. 118
Chapter 1

Introduction

The need for delegating information arises when the data owner wants or have to have her data handled by an external party. If the external party is untrusted and data are confidential, delegation should be performed in a way that preserves security. Uses of delegation range from public administration to smart cards [Dom97]. In this work, correctness and security requirements as well as protocols are specified for delegation of computing and data. A cryptographic solution to the secure delegation problem is described which provides data confidentiality and computation verifiability. Finally, an implementation allowing secure delegation of information over the Internet is briefly discussed. The CORBA model of object-oriented distributed computing is used as the framework for the implementation of the prototype.

Chapter 2 analyzes security in distributed systems, motivates the delegation problem and presents delegation scenarios and their security requirements. Chapter 3 contains some background on secure multi-party computation, privacy homomorphisms and other approaches to secure computation in unsafe environments; it can be skipped by the reader familiar with these topics. Chapter 4 resumes the delegation problem with the security functionalities and requirements identified in Chapter 2 and then proceeds to sketch a practical cryptographic solution based on privacy homomorphisms and parity verification. Chapter 5 discusses how to apply the solution described in
Chapter 4 to the delegation scenarios identified in Chapter 2; this chapter also contains an introduction to the CORBA model, which will be used to implement the solution and is further described in Appendix B. Chapter 6 describes Dikë, a prototype for secure delegation. Chapter 7 contains some conclusions, summarizes the contributions of this thesis and lists some topics for future research. Appendix A contains the basic concepts of UML, the specification language used for Dikë. Appendix B deals with the CORBA essentials. Finally, Appendix C explains some of the basic cryptographic concepts and tools referenced in this thesis.
Chapter 2

Delegation in Distributed Systems

In many scenarios, data cannot be processed where they originate or belong to. In such cases, the data owner must transfer data to a remote environment (the handler) for processing. When these scenarios are associated with computers, the natural approach is to consider a mapping of these scenarios to computers connected by a network. In computer terms, this means that data are processed in a distributed system.

If data are confidential, a security problem arises. A legal solution to this problem is to require that the handler sign a non-disclosure agreement. For example, this is the procedure followed when some government agencies release data to universities for research purposes.

The above legal solution has a serious drawback: the data owner must believe that the handler is fair. This implies that the handler must be highly trusted, since there is no technical means to prevent data misuse. A better alternative would be a secure delegation mechanism, i.e. an automated secure mechanism allowing the data owner to control and limit the kind of operations performed by the handler, and the data to be unclassified for the handler’s use.
Section 2.1 summarizes the state of the art of security in distributed systems. Section 2.2 explicitly defines what delegation is and what types of delegation can be distinguished. Section 2.3 identifies the security requirements of delegation, which arise if the rights of the data owner are to be protected. Section 2.4 characterizes the functionality necessary for the fulfillment of the security requirements.

2.1 Distributed systems security

"No viable security system design can be based on the principles of policy, integrity and secrecy, because in the modern world integrity and secrecy are not achievable and policy is not manageable"

The above manifesto is the central proposition of [Bla96] and, in our opinion, adequately depicts the situation of the current security models for distributed systems. These models are based on the following principles:

1. Policy is enforced by the system and pretends to protect resources from unauthorized manipulation.

2. Integrity of the physical system and its code guarantees that policy is enforced.

3. Secrecy of cryptographic keys and sensitive data underlies policy enforcement mechanisms.

In distributed systems, policy has two related fundamental problems: complexity and scale. In each of these directions policy scales poorly. The complexity of policy which must be stated in order to manage a system securely increases if any of the following increases:

- number of subjects

- number of job functions objects
2.1 Distributed systems security

- number of operations

- number of semantic classes of data (sensitivity labels, categories, etc.)

The last two of these are the worst. The reason is that, due to the traditional view of policy as a centralized process, the policy maker has to know the implications and the structure of the access control operations. This means that the administrator has to be omnipresent in all the domain under her control. In a distributed system, this is impossible because detailed knowledge of the structure of the system is not possible as, under normal use, an application typically goes out of the domain under control of the administrator.

System integrity assures that the security policy of a system cannot be bypassed. In order to get a system with excellent system integrity, it should be extremely well designed and built. An illustration of the challenge integrity poses to the average working programmer appeared recently in [Fin96]. Presumably a prerequisite to demonstrating that a system always does what it is supposed to do, is specifying what it is supposed to do. In [Fin96] it is reported that less than one third of a sample of computer science students were able to understand a short specification in the Z formal language. Even if one succeeds in writing a correct specification, there is still a long way to go to reach system integrity for the final system. Furthermore, integrity is a quality that should be maintained during all the system life cycle, and this is still more difficult to achieve.

System secrecy is traditionally associated to the “fortress model”. Kerchoff’s principle states that, if the cryptographic keys remain confidential, the system is secure. The simple problem with secrets is that people are not good at keeping them: many attacks on “secure” systems succeed simply because they are performed by insiders who misuse their legitimate access to cryptographic keys or other confidential information. Outsiders also penetrate “secure” systems by exploiting secrecy failures tied to human factors. Another troublesome issue related to secrecy is who decides what is secret and how is that decision made. Cryptography provides some solutions to distribute the
control on secrets (secret sharing schemes, *e.g.* [Sae98]); however, the overhead and complexity of systematically implementing such schemes in practice is prohibitive.

### 2.2 Delegation concepts

The following classification is from the point of view of the use of classified information. Depending on who is interested in the results of the data processing, two types of delegation scenarios can be distinguished:

**Computing delegation** A data owner sends to a handler a data set, some basic operations and an expression to be evaluated on the data. The data owner is the party interested in the computation.

**Data delegation** A data owner sends a data set to a handler, who thereafter performs some computation with those data. The handler is the party interested in the computation.

Figure 2.1 contains a graphical representation of the computing delegation problem, where the two parties are considered as two independent computing environments; one (the owner) has confidential data and the other (the handler) has computational power. In an ideal model the computational protocol consists of two messages:

- One message from the owner to the handler with the data and the computation that the owner wants to be done by the handler.
- Another message from the handler to the owner with the result of the computation.

Figure 2.2 is a graphical representation of the data delegation problem. As in the computing delegation problem, the two parties are considered as two independent computing environments, one (the owner) having confidential data and the other (the handler) having computational power. In this case, the
Figure 2.1: Computing delegation scheme

A computational protocol ideally consists of only one message. The message is sent by the owner to the handler, and contains the data of the owner. From this point on, the handler can manipulate the data to obtain the desired results. In Figure 2.2, two more messages have been depicted from the handler to himself; they are there to highlight that subsequent use of the information is performed internally by the handler.

Figure 2.2: Data delegation scheme

2.3 Requirements for secure delegation

From the models presented above, data transfers are performed between the two computing environments, the handler and the owner. It is assumed that each computing environment is in a different security domain. Thus, control on the access and use of the data has to be considered if delegation is to be secure.

Security in the context of data delegation essentially means two things:

1. Protecting the confidentiality of information from unauthorized attempts to access it or interfere with its use.

2. Ensuring the integrity of computations in environments which are deemed unsafe by the data owner.
Therefore, the two basic security requirements to be fulfilled by the proposed solution are:

Confidentiality In computing delegation, no classified information should be disclosed. In data delegation, information can be partially disclosed, but in a controlled manner and only to authorized users.

Integrity In computing delegation, it is necessary to verify that the result is what was asked for. In data delegation, it is necessary to verify that the result corresponds to the claimed computation on the classified data, in order to control what information can be disclosed.

2.4 Available security functionalities

Security models usually consider a standard set of security functionalities. Such functionalities are the building blocks that can be used to partly satisfy the requirements set out in Section 2.3. The main standard security functionalities are:

Identification and authentication Human users and objects which need to operate under their own rights (principals) must be identified and authenticated to verify that they are who they claim to be.

Authorization and access control This means deciding whether a principal can access an object, using the identity and/or privilege attributes associated to the principal (capabilities) or to the object (access control lists).

Security auditing Users must be made accountable for their security-related actions. Logs should be maintained to record which actions the user has performed in the past.

Security of communication Communication channels are usually unsafe and subject to wire-tapping and intrusion. However, secure delegation cannot be achieved unless objects are able to communicate with each other
in a secure way. Communications security splits into:

- Authentication of clients to targets
- Authentication of targets to clients
- Integrity protection
- Confidentiality protection, normally through the use of encryption

**Non repudiation** This functionality provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny having sent or received some data.

**Administration** Security information (*e.g.* security policy) must be administered.

It must be noted the above functionalities do not suffice to completely satisfy the requirements of secure delegation. For example, the confidentiality requirement should be satisfied in a way that is compatible with the delegation of confidential data to untrusted handlers. Also, integrity is related to handler computation but is to be checked by the data owner. A complete solution for secure delegation is specified in Chapter 4.
Chapter 3

Cryptographic Background

In this chapter, previous research relevant to the delegation problem is reviewed. Section 3.1 contains an overview on secure multi-party computation and analyzes its applicability to delegation. Section 3.2 surveys Feigenbaum’s approach to computing on encrypted data. Section 3.3 surveys a solution for delegation of matrix computations based on the principles of fragmentation, dissemination and replication, following the ideas of fault-tolerant systems. Section 3.4 summarizes some ad-hoc solutions for adding encrypted data. Section 3.5 defines privacy homomorphisms and discusses previous proposals for privacy homomorphisms and their properties. Section 3.6 compares the usefulness of the above approaches for delegation.

3.1 Secure multi-party computation

Roughly speaking, secure multi-party computation [GMW87] is a concept by which a set of $n$ parties cooperate to calculate a function $f : I_1 \times \cdots \times I_n \mapsto O_1 \times \cdots \times O_n$. The inputs of $f$ are vectors $\langle i_1, \cdots, i_n \rangle$ such that $i_i$ is provided by the $i$-th party. The outputs of $f$ are vectors $\langle o_1, \cdots, o_n \rangle$ such that the output $o_i$ is intended for the $i$-th party. One of the main security requirements is that each party’s input and output should remain unknown to the rest of parties (except for the information that can be derived from the knowledge of
the party’s legitimate output); however, all parties should be convinced that $f$ has been correctly computed. See Figure 3.1 for a graphical representation of the model.

In Figure 3.1 each circle represents one of the parties, lines represent messages, and the arrow on a line represents the direction of the message. The cloud represents an unspecified construct to compute the function. For each party only two messages are shown: one message represents the input $i_n$ to the function $f$, and the other message represents the output $o_n$ to the party. In almost all the proposed models, a multi-party protocol typically requires several messages to/from each party. For the sake of simplicity, the unspecified construct can be idealized as behaving as a trusted third party that receives the inputs from the parties and sends them the outputs. However, the key feature of secure multi-party computation is to perform the same computation without a trusted third party.

Secure multi-party computation is a very general idea that conceptually encompasses the delegation problem. However, a closer look at it is necessary to realize that the protocols being proposed actually require all parties to actively take part in computations. This is clearly in conflict with the idea
of delegation, which is essentially passive on the owner’s side: the delegating party (owner) wants most of work done by the delegated party (handler).

3.1.1 Definitions

In this subsection we state some definitions used to describe and classify the different models of secure multi-party computation which are adapted from [Can95]. The parties’ distrust in each other and in the network is usually modelled via an adversary that has control over some of the parties, and perhaps also over the communication media, or channels. (We call parties controlled by the adversary corrupted.) Many different adversary types (or adversary models) may be considered, each modelling different problems, or addressing a different setting. The requirements for solutions of secure computation problems, as well as the techniques used, differ considerably with the adversary models. In order to be able to present the work done in this field, we briefly sketch some prominent parameters defining adversary models.

Computational power: Adversaries can be either computationally unbounded or restricted to probabilistic polynomial time (PPT, see Appendix C). We remark that throughout this work we assume that the uncorrupted parties are restricted to PPT.

Control over the communication: Three levels of abstraction can be distinguished. In the most abstract level, the adversary has no access to the channels; that is, two uncorrupted parties communicate securely without the adversary hearing or affecting the communication (secure channels assumption). In the second level, it is assumed that the adversary can hear all the communication among parties, but cannot alter that communication (insecure or authenticated channels assumption). The third level assumes unauthenticated channels, so that the adversary has full control over the communication; in addition to hearing the communication, the adversary can delete, generate and modify messages at will.
Synchrony: In a synchronous network all parties have a common, global clock. All messages are sent on a clock 'tick', and are received at the next 'tick'. In an asynchronous network no global clock exists. Furthermore, arbitrary (however finite) time may lapse between the sending and receipt of a message (in particular messages may be received in an order different than the order of sending). It must be remarked that, although often taken as a parameter of the network, synchrony may be considered as a parameter of the adversary. In particular, setting the actual delays of the messages may be naturally considered as an additional power given to the adversary.

Number of corrupted parties: The number of corrupted parties at any given time is normally considered to be limited. An adversary is \( t \)-limited if at any given time at most \( t \) parties are corrupted. A protocol is \( t \)-resilient if it meets its specification in the presence of \( t \)-limited adversaries. \( t \)-resilient protocols for secure computation are also called \( t \)-secure.

Control over corrupted parties: A distinction is made between eavesdropping adversaries that only gather information and do not alter the behavior of corrupted parties, and Byzantine adversaries that may alter the behavior of the corrupted parties in an arbitrary and coordinated way. In asynchronous networks it makes sense to consider also fail-stop adversaries. Here the only diversion from the protocol allowed to the corrupted parties is to "crash", that is to stop sending messages at some time during the computation. (A crashed party may not resume sending messages.) For security considerations, we assume that faulty parties continue receiving messages and have an output.

Adaptivity: The way in which the corrupted parties are chosen is called adaptivity. It is simplest to assume that the set of corrupted parties is arbitrary but fixed (of course, the uncorrupted parties do not know the identity of the corrupted parties). We call such adversaries non-adaptive.
Alternatively, we may let the adversary choose which parties to corrupt as the computation proceeds, based on the information gathered so far. Once a party is corrupted it remains so for the rest of the computation. We call such adversary *adaptive*.

Lastly, the adversary may be assumed to corrupt, in an adaptive way, a different set of parties at different times during the computations (Here, parties that were once corrupted may become uncorrupted again, and there is no limit on the total number of parties that were corrupted at some time or another during computation.) Such adversaries are called *mobile*.

### 3.1.2 A historic overview

The problem of secure computation was first formulated by Yao for the two-party case in 1982 [Yao82]. Five years later, Goldreich, Micali and Wigderson showed how to securely compute any function whose inputs are divided among the parties, in a computational setting [GMW87]. That is, in [GMW87] a synchronous network of \(n\) parties is considered, where the communications channels are insecure, and the parties, as well as the adversary, are restricted to probabilistic polynomial time (PPT). In this model they showed, under the assumption that one-way functions with trapdoor exist (see Appendix C), how to construct \(n\)-secure protocols for computing *any* function, in the presence of eavesdropping adversaries. In the case of Byzantine adversaries they presented \((\lfloor n/2 \rfloor - 1)\)-secure protocols for computing any function. Their protocols can be shown secure in the presence of non-adaptive adversaries. In [CDv88], a similar scenario is considered and a protocol is presented which is claimed to be conceptually simpler and more efficient for functions involving small Boolean circuits; moreover, parties are assumed to have unlimited computing power.

Ben-Or, Goldwasser and Wigderson [BGW88] (and, independently, Chaum, Crépeau and Damgaard [CCD88]) studied secure multi-party computation in the secure channels setting. They showed that:
• If the adversary is an eavesdropper then there exist \(\lceil n/2 \rceil - 1\)-secure protocols for computing any function.

• If the adversary is Byzantine, then any function can be \(\lceil n/3 \rceil - 1\)-securely computed. Furthermore, they show that these bounds on the number of corruptions are tight. These protocols can be shown secure in the presence of non-adaptive adversaries. Adaptive security (i.e., security in the presence of adaptive adversaries) is provable in certain variants of this setting.

Goldwasser and Levin built on a long sequence of works studying the case of Byzantine adversaries limited to PPT, where a majority of parties may be corrupted [GL91]. Chor and Kushilevitz studied secure multi-party computation with corrupted majority of the parties in the secure channels settings [CK91]. Goldreich, Goldwasser and Linial studied secure multi-party computation in the presence of insecure channels and computationally unlimited adversaries [GGL91]. Ostrovsky and Yung studied secure multi-party computation in the presence of secure channels and mobile adversaries [OY91]. Micali and Rogaway [MR92], and also Beaver [Bea92], proposed definitions for secure multi-party computation in the secure channels setting, in the presence of adaptive adversaries.

Almost all constructions proposed for secure computation follow the share, compute and reconstruct steps originally proposed by Goldwaser, Micali and Wigderson in [GMW87].

### 3.1.3 Two-party computation: an example protocol

The first protocol proposed for two party secure computation ([Yao82]) was the so-called millionaire problem. In this problem, two millionaires want to find out who is richer without revealing the amount of money they possess to each other.

We will next illustrate this problem in a somewhat simpler situation in which two persons, Alice and Bob, want to find out who is older without
3.1 Secure multi-party computation

revealing their age to each other (dealing with ages is simpler than dealing with money, because the range is more restricted). Assume that Alice and Bob are honest in that they use their true ages during the protocol. Otherwise, they would only know that the age of one of them is greater or less than the age claimed by the other, but the problem would not be exactly solved. Assume that Alice and Bob are \( i \) and \( j \) years old, respectively, where \( 1 \leq i, j \leq 100 \). The protocol specified below relies on a public-key cryptosystem (see Appendix C), such that both parties know each other’s public key (\( E_A \), resp. \( E_B \)), but not each other’s private key (\( D_A \), resp. \( D_B \)).

**Protocol 1 (Millionaire problem)**

1. Bob picks a large random number \( x \) and privately computes \( E_A(x) = k \).

2. Bob tells Alice the value \( k - j \).

3. Alice privately computes the values \( y_u = D_A(k - j + u) \), \( 1 \leq u \leq 100 \). Alice then chooses a large prime \( p \) (the size of \( x \) and \( p \) has been agreed by both parties in advance, so that the size of \( p \) is less than the size of \( x \)) and privately computes \( z_u \) such that \( y_u \equiv z_u \pmod{p} \), for \( 1 \leq u \leq 100 \). For all \( 1 \leq u, v \leq 100 \) with \( u \neq v \), Alice checks that

\[
|z_u - z_v| \geq 2 \quad \text{and} \quad 0 < z_u < p - 1
\]  

If the above check fails, Alice chooses another prime number.

4. Alice gives to Bob the following ordered number sequence

\[
z_1, \ldots, z_i, z_{i+1} + 1, z_{i+2} + 1, \ldots, z_{100} + 1, p
\]  

5. Bob checks whether the \( j \)-th number in the sequence is congruent with \( x \mod p \). If yes, Bob concludes that \( i \geq j \); otherwise, Bob infers that \( i < j \).
6. Bob tells his conclusion to Alice. Alternatively, Alice and Bob can exchange their roles and repeat the protocol from Step 1.

The conclusion of Step 5 is correct, because the \( j \)-th number \( z'_j \) in the sequence (3.2) verifies

\[
i \geq j \rightarrow z'_j = z_j \equiv y_j = x \pmod{p}
\]

\[
i < j \rightarrow z'_j = z_j + 1 \not\equiv z_j \equiv y_j = x \pmod{p}
\]

Condition (3.1) ensures that no number appears twice in the sequence (3.2). If a replicate of the \( l \)-th number appeared before the \( l \)-th place in the sequence (3.2), then \( B \) would know that \( i < l \). Finally, encryption of the values \( y_u \) \( \pmod{p} \) is necessary, since if Alice sent to Bob the sequence \( y_u \), then Bob could find \( i \) by just applying Alice’s public key \( E_A \) to that sequence, since Bob knows the values \( k - j + u \), for \( 1 \leq u \leq 100 \).

### 3.2 Computing with encrypted data

The problem of computing a given function on encrypted data was dealt with in [Fei86] and [AFK87]. The problem addressed can be described as follows. Given a function \( f : I \rightarrow O \) Alice wants to find out the value of \( f \) for a given value \( x \), but she either does not wish or does not have enough computational power to evaluate the function \( f \). On the other hand, Alice knows that Bob has enough computational power to evaluate \( f \); in other words, Bob can send to Alice the value \( f(x) \) for any \( x \). If Alice wants to use Bob, the security problem arises when Alice also wishes to keep secret the value \( x \).

#### 3.2.1 Definitions

Informally, an encryption scheme for the problem \( f \) is a method whereby Alice \((A)\), using her limited computational power, can transform the cleartext instance \( x \) into an encrypted instance \( y \), obtain \( f(y) \) from Bob \((B)\) and infer
$f(x)$ from $f(y)$ in such a way that Bob cannot infer $x$ from $y$. When such an encryption scheme exists, $f$ is said to be an *encryptable function*. In the sequel three encryption schemes are shown which are based in number-theoretic problems [AFK87]: the discrete logarithm function, the quadratic residuosity function and the primitive root function. See Appendix C for details on the complexity of those problems.

The above approach is conceptually depicted in Figure 3.2 and is only usable for functions $f$ for which an encryption scheme is known. However, there is no systematic method to derive an encryption scheme for an arbitrary function $f$. Therefore, the approach described in this section cannot be used in general for delegation of computing and data.

### 3.2.2 Examples of encryptable functions

**The Discrete Logarithm Problem (DLP)**

The DLP function $f$ takes as arguments a prime $p$, a generator $g$ for $\mathbb{Z}_p^*$, and an integer $u$ such that $(u, p) = 1$. The value $f(u, g, p)$ is the unique exponent $e \in [1, p - 1]$ for which $g^e \equiv u \mod p$. The key space $K$ used in the encryption scheme is $\mathbb{Z}$. The answer to the encrypted instance is denoted by $e'$.

```plaintext
k(⟨u, g, p⟩)
{
    Return(An element of [1, p − 1] chosen uniformly at random);
```
\}
E(\langle u, g, p \rangle, k)
\{
\textbf{Return}(\langle ug^k \mod p, g, p \rangle);
\}
D(\langle u, g, p \rangle, k, e')
\{
\textbf{Return} (e' - k \mod p - 1);
\}

\textbf{The Quadratic Residuosity Problem (QRP)}

The QRP function $f$ takes two arguments, an integer $n$ that is the product of two distinct primes $p$ and $q$, with $p \equiv q \equiv 3 \mod 4$, and an integer $u$ such that $(u, n)$ is 1 and the Jacobi Symbol ($\frac{u}{n}$) is $+1$. (We could use a wider class of $n'$ – for example, the so-called Blum integers – but the increased generality is not needed to illustrate encryptability.) The value $f(u, n)$ is 1 if there is an integer $a$ such that $a^2 \equiv u \mod n$, and $f(u, n)$ is 0 otherwise. Note that the Jacobi symbol ($\frac{u}{n}$) can be computed in polynomial time using the Reciprocity Law and that, if $n$ has the specified form, then $f(-1, n)$ is 0. We use $\mathbb{Z}^*_n[+1]$ to denote the integers in $[1, n - 1]$ that are relatively prime to $n$ and have Jacobi symbol $+1$. The key space $K$ is $\mathbb{Z} \times \{0, 1\}$, and the answer to the encrypted instance is denoted by $e'$.

\begin{verbatim}
k(\langle u, n \rangle)
\{
\textbf{Do}
Choose $z$ uniformly at random from $[1, n - 1]$
\textbf{While} $(z, n) > 1$ or ($\frac{z}{n}$) = $-1$;
Choose $\epsilon$ uniformly at random from $\{0, 1\}$;
\textbf{Return}(\langle z, \epsilon \rangle);
\}
E(\langle u, n \rangle, \langle z, \epsilon \rangle)
\end{verbatim}
\[
\{ \\
\text{If } \epsilon = 0 \\
\text{Then Return } (\langle uz^2 \mod n, n \rangle); \\
\text{Else Return } (\langle -uz^2 \mod n, n \rangle); \\
\} \\
D(\langle u, n \rangle, \langle z, \epsilon \rangle, \epsilon') \\
\{ \\
\text{If } \epsilon = 0 \\
\text{Then Return } (\epsilon'); \\
\text{Else Return } (1 - \epsilon'); \\
\}
\]

The Primitive Root Problem (PRP)

The PRP function \( f \) takes as arguments a prime \( p \) and a integer \( u \) that is relatively prime to \( p \). The value \( f(u, p) \) is 1 if \( u \) generates the cyclic group \( \mathbb{Z}_p^* \) (i.e., if \( u \) is a primitive root mod \( p \)), and 0 otherwise. The key space \( K \) is \( \mathbb{Z} \), and once again the answer to the encrypted instance is denoted by \( \epsilon' \).

\[
k(\langle u, p \rangle) \\
\{ \\
\text{Do} \\
\text{Choose an odd } k \text{ uniformly at random from } [1, p - 1] \\
\text{While } (k, p - 1) > 1; \\
\text{Return } (k); \\
\} \\
E(\langle u, p \rangle, k) \\
\{ \\
\text{Return } (\langle u^k \mod p, p \rangle); \\
\} \\
D(\langle u, p \rangle, k, \epsilon') \\
\{ \\
\text{Return } (\epsilon'); \\
\}
\]

\(^*/ f(u^k \mod p, p) = f(u, p) \text{ if } (k, p - 1) = 1^*/ \]
In all of these encryption schemes, the functions $E, D$ and $k(\cdot)$ have the required properties, i.e. $E$ and $D$ terminate in polynomial time (see Appendix C), $k(\cdot)$ terminates in expected polynomial time and produces keys of length polynomial in $|x|$, and the fact that a key $k$ is valid for a cleartext instance $x$ can be checked in polynomial time. To see that the function $k(\cdot)$ in the PRP scheme runs in expected polynomial time, note that $p - 1$ has fewer than $\log p$ distinct prime factors, whereas the total number of distinct primes less than $p$ is proportional to $p/\log p$; hence the number of valid keys for each pair $\langle u, p - 1 \rangle$ can be lower-bounded trivially by $\Omega(p/\log p)$, and we can expect to find one by trial and error in $O(\log p)$ trials.

The following result is proven in [AFK87].

**Theorem 1** The encryption scheme given for the DLP leaks at most $g$ and $p$. The scheme for the QRP leaks at most $n$. The scheme for the PRP leaks at most $p$ and the order of $u \mod p$.

### 3.3 Delegation by fragmentation, dissemination and replication

In [Tro91], several approaches for processing sensitive data in unsafe environments are analyzed and compared. The environments considered have a safe zone and an unsafe zone. The approaches can be listed as:

1. Protection and replication. Operations on sensitive data are performed in the safe zone. Besides, operations are replicated over several processors for reliability reasons.

2. Encryption and replication. Operations on sensitive data can be performed in the unsafe zone provided that data have been previously encrypted in the safe zone. Due to the lack of sufficiently flexible homomorphic encryption transformations at the time of writing [Tro91], this approach is not developed by the authors.
3. Fragmentation, redundancy and dissemination. Data and operations are split in the safe zone into several data subsets and tasks. The unsafe zone consists of several processors, each of which is responsible for performing a task on a data subset. The partial results thus obtained are then merged in the safe zone to get the overall result. In addition, tasks and partial data sets can be replicated for reliability reasons. Of course, this approach mainly applies to processing matricial data, because in general splitting the initial data set and the processing may not be obvious.

3.4 Ad-hoc procedures for adding encrypted data

In [ALN87], solutions for updating encrypted balances are discussed. Specifically, the following situations are considered:

1. Add an encrypted data element $C_1$ (last balance) to a cleartext $P_2$ (the updating transaction) without having to decipher the encrypted balance. The result of decrypting $C_1 + P_2$ should yield $P_1 + P_2$, where $P_1$ is the cleartext corresponding to $C_1$. The proposed solution consists of adding a key $X$ to the initial balance; to decipher the current balance, at any stage, it suffices to subtract $X$.

2. Add encrypted data $C_1$ to other encrypted data $C_2$ without having to decipher either data before the addition. The result of decrypting $C_1 + C_2$ should yield $P_1 + P_2$, where $P_i$ is the cleartext corresponding to $C_i$. Several solutions are examined:

- To encrypt $P_i$, compute $C_i = P_i + X$, where $X$ is a key and the addition is modular. The decrypt the $n$-th balance, $n$ must be recorded because $nX$ must be subtracted to the current encrypted balance.
• Use an additive privacy homomorphism (see Section 3.5) to encrypt $P_i$. This solution allows the ciphertexts $C_i$ to be added.

• Consider several keys $X_1, \ldots, X_m$, which are used in turn to encrypt $P_1, \ldots, P_m, \ldots$. Encryption consists of modulo adding the cleartext and the key.

The proposed solutions are restricted to performing additions on encrypted data; complete arithmetic is not considered. In the first two solutions, knowledge of a single cleartext-ciphertext pair allows to determine the key $X$. The authors of Ahituv, Lapid and Neumann (1987) discard the third solution with the argument that additive privacy homomorphisms can be broken using a chosen-ciphertext attack (yet this attack assumes that the cleartexts for a chosen set of ciphertexts can be determined!). The fourth solution requires a long random key if knowledge of several cleartext-ciphertext pairs is to be tolerated; besides, it becomes complex to keep track of which keys should be subtracted to decrypt an encrypted balance.

### 3.5 Privacy homomorphisms

The first general approach to encrypted data processing is due to the authors of [RAD78], when they introduced the notion of privacy homomorphism (PH from now on). Basically, such homomorphisms are encryption functions $E_k : T' \rightarrow T$ allowing a set $F$ of operations on encrypted data without knowledge of the decryption function $D_k$. Knowledge of $D_k$ allows the result of the corresponding set $F'$ of cleartext operations to be retrieved. The availability of secure PHs is central to the delegation of computation: the idea is to encrypt data at a classified level, to process them at unclassified computer facilities and to decrypt the result at a classified level. By way of illustration, consider the following example of PH, given in [RAD78]

**Example 1** Let $p$ and $q$ be two large and secret primes (100 decimal digits each). Let $m = pq$ be public. Define the cleartext set as $T' = \mathbb{Z}_m$ and
the set of cleartext operations as $F' = \{+_m, -_m, \times_m\}$ consisting, respectively, of addition, subtraction and multiplication modulo $m$. Let the ciphertext set be $T = \mathbb{Z}_p \times \mathbb{Z}_q$. Operations in the set $F$ of ciphertext operations are the componentwise version of those in $F'$. Define the encryption key as $k = (p, q)$ and the encryption transformation as $E_k(a) = [a \mod p, a \mod q]$. Given $k = (p, q)$, the Chinese remainder theorem is used to compute $D_k([b, c])$. A technical detail is that when the unclassified level computes on encrypted data, it cannot reduce partial results to the secret modulo $p$ and $q$; only reduction to the public modulus $m$ is possible, so that in fact the unclassified level operates over $\mathbb{Z}_m \times \mathbb{Z}_m$; however, at decryption time, knowledge of the key allows the classified level to map encrypted results from $\mathbb{Z}_m \times \mathbb{Z}_m$ back to $\mathbb{Z}_p \times \mathbb{Z}_q$ prior to using the Chinese remainder theorem.

Unfortunately, it is shown in [BY88] that this PH can be broken — i.e., $p$ and $q$ can be found — by a known-cleartext attack. ■

Next follows a summary of the state of the art on PHs. If a PH preserves order, then it is insecure against a ciphertext-only attack. If addition is one of the ciphertext operations of a PH, then such a PH is insecure against a chosen-cleartext attack ([ALN87]). With the exception of the RSA algorithm — which preserves only multiplication, see Appendix C —, all examples given in [RAD78] were broken by ciphertext-only attacks or, at most, known-cleartext attacks (see [BY88]); the authors of [BY88] invented $R$-additive PHs, which securely allow ciphertext addition at the cost of restricting the number of ciphertexts that can be added together. In [BM85], a partially homomorphic scheme for statistical computation on encrypted data was proposed that consists of a two-layer encryption: data records are first encrypted as sparse polynomials, and these are then encrypted as regular polynomials; while the first layer is homomorphic, the second is not (yet the second layer is needed because the PH in the first layer is insecure); therefore, encrypted data processing is not feasible without a trusted device able to decrypt the second layer. Lacking secure PHs that preserve more than one operation, subsequent attempts at encrypted data processing have relied on ad-hoc solutions ([ALN87], [Tro91]).
In [Dom96], a PH preserving addition and multiplication was presented which was conjectured to be computationally resistant against known-cleartext attacks. In Chapter 4 of this thesis, we propose the first PH preserving both operations that can be proven unconditionally secure against known-cleartext attacks, as long as the ciphertext space is much larger than the cleartext space.

3.6 Comparison and interim discussion

When comparing the various approaches presented in this chapter, the following remarks are in order.

Multi-party computation (Section 3.1) is a very general approach and in principle can simulate all other approaches. However:

- Multi-party computation normally assumes that all parties participate in the computation at the same level. Thus, there is no master-slave hierarchy, as is the case in the delegation.

- As a consequence of the above, multi-party protocols do not isolate parts of the computation to be performed exclusively by each party. Instead “everything is done in common”.

- In general multi-party protocols, functions are specified as Boolean circuits. Basic operations are logical gates, and are thus extremely low-level.

Computing on encrypted data (Section 3.2) already incorporates the notion of master-slave hierarchy. However, known protocols for computing on encrypted data are oriented to particular functions (in fact they are \textit{ad-hoc} protocols). An arbitrary function must be encryptable in order for this approach to be used; even if the function is encryptable, a specific algorithm has to be found for computing it on encrypted data.

The fragmentation-redundancy-dissemination approach (Section 3.3) is only applicable to a restricted class of functions and data. More specifically, the function and the data must be splittable into several tasks to be performed
on several data subsets. This approach is well-suited for some matrix computations, but it is not usable on arbitrary data and for arbitrary functions. Moreover, if all unsafe processors collude and pool their data subsets and partial results, the initial data set and the overall result may be compromised.

The \textit{ad-hoc} solutions of Section 3.4 are restricted to addition of encrypted data. Multiplications and comparisons for equality are not possible. Besides, some of the proposed solutions cannot withstand known-plaintext attacks.

Privacy homomorphisms (Section 3.5) allow a very flexible delegation of computation. The party encrypting the data (data owner) is the master and the party doing the computations on encrypted data is the slave (data handler). The basic operations allowed by privacy homomorphisms on encrypted data are typically of arithmetical nature, so they are high-level operations as compared to the Boolean operations provided by multi-party protocols. Furthermore, any function that can be expressed in terms of those basic operations can be constructively evaluated by the data handler.

Thus, privacy homomorphisms appear as the best option when trying to devise an implementable solution to the delegation problem. The main problem is to find privacy homomorphisms that:

1. Allow a sufficient set of operations to be performed on encrypted data.

2. Offer security against known-plaintext attacks. This is a requirement in the case of delegation, since the data handler may have access to some encrypted values and their corresponding decrypted values.

In Chapter 4, a new privacy homomorphism is presented that allows full arithmetic on encrypted data while remaining unconditionally secure against known-plaintext attacks.
Cryptographic Background
Chapter 4

A Secure Solution for Delegation

In Chapter 3 privacy homomorphisms were argued to be the most practical and flexible tool for secure delegation. This chapter describes a cryptographic solution which relies on probabilistic verification of computation and privacy homomorphisms. Section 4.1 illustrates some practical applications where delegation is not viable unless it is secure. Section 4.2 specifies the security requirements of delegation and proposes protocols for computing and data delegation. Section 4.3 describes the parity-checking verification mechanism used in the protocols of Section 4.2. Section 4.4 describes a new privacy homomorphism which is unconditionally secure against known-plaintext attacks and allows full arithmetic on encrypted data.

4.1 Need for secure delegation

In most cases, delegation of computing and/or data takes place in a hostile or unsafe environment. Thus, delegation must be protected either by legal measures or by technical measures. We illustrate next a few practical applications where delegation has to be secure:

- A computing delegation problem happens whenever a (small) company
A Secure Solution for Delegation

wants to use external computing facilities to do some calculations on corporate confidential data. Think of a medical research team using a (insecure) university mainframe for processing confidential healthcare records. The reason for using external facilities may be the complexity of the calculations but also the huge volume of the data set.

- Data delegation problems appear when several lower-level organizations (municipalities, member states, etc.) cooperate with a higher-level organization (national agency, European Union, etc.) in data collection (census data, etc.). In return, the former organizations would like to analyze the whole collected data set, but only the latter organization is legally entitled to do so. Without secure data delegation, the higher-level organization will have to spend resources in (uninteresting) analyses requested by the cooperating organizations.

- In [Dom97], availability of secure data delegation is relied on for increasing the multi-application capacity of smart cards. The basic idea is that, if a very resource-demanding application is to be run on card-stored data, the card exports these data in encrypted form and the application is run on an external computing server.

4.2 Requirements and protocols for delegation

The basic requirements of both computing and data delegation are security and correctness. In both types of delegation, security essentially translates to preservation of data confidentiality. This means that the handler should not know the computation input data.

In delegation of computing, correctness essentially translates to verifiability of the handler computation by the data owner. The handler may deviate from the claimed computation accidentally (overflow) or intentionally. In any verification procedure, there is a tradeoff between the confidence attained and the resources spent. If total confidence is desired, the owner must entirely
repeat all computation (which makes delegation of computing useless); on the other hand, if the owner does not worry about the correctness of the computation, no check is needed. Intermediate solutions will be proposed here which provide a reasonable confidence at a reasonable cost.

In data delegation, there is no correctness requirement from the data owner’s viewpoint, since computation is done by the handler for his own use. From the handler’s viewpoint, overflow detection is a serious problem when computing on encrypted data. The owner’s verification can help as follows: if the owner detects fraud, she tells the handler. If the handler committed no fraud, he knows that overflow has occurred.

To meet the above requirements, the following protocols are proposed which rely on homomorphic encryption.

Protocol 2 (Computing delegation)

1. Prior to sending data to the handler, the owner encrypts them using an encryption transformation which allows some operations to be performed directly by the handler on encrypted data (homomorphic encryption).

2. The handler returns a result which is still encrypted and must be decrypted by the data owner to recover the clear result.

A graphical representation of the above protocol is given in Figure 4.1.

![Figure 4.1: Secure computing delegation scheme](image)

Protocol 3 (Data delegation)

1. Data sent to the handler are encrypted by the owner using a homomorphic encryption transformation. The owner also supplies the handler with the operations on encrypted data supported by the encryption transformation used.
2. Upon completing his computation, the handler sends the (encrypted) result to the owner for decryption. The handler also provides the expression used to obtain such a result, in order to allow some verification.

3. The owner verifies the handler’s computation. If no fraud can be detected and the claimed expression does not lead to evident disclosure of input data (inference controls should be applied here, but this is beyond the scope of this thesis, see [Den82],[Mat98]), then the owner returns the decrypted result to the handler. Fraud occurs when the result does not correspond to the claimed expression.

A graphical representation of the above protocol is given in Figure 4.2.

![Figure 4.2: Secure data delegation scheme](image)

**Note 1** With the protocols proposed above, the owner does not need to be very active. In other words, the work performed by the handler depends on the nature of the processing to be done, whereas the work done by the data owner is fairly independent of the processing (in fact, the data owner functionality could be implemented in hardware or firmware).

In summary, to solve both the secure computing and data delegation problems, we need to solve two more basic problems: data confidentiality and computation verifiability. We next propose solutions to such problems.

### 4.2.1 Data confidentiality

As mentioned above, in both delegation problems the handler must be able to compute without being revealed the input data. This means that computation
is to be carried out on encrypted data. Encryption transformations allowing some operations to be carried out directly on encrypted data are known as privacy homomorphisms (PHs for short, see [RAD78]).

Basically, such homomorphisms are encryption functions \( E_k : T \rightarrow T' \) allowing a set \( F' \) of operations on encrypted data without knowledge of the decryption function \( D_k \). Knowledge of \( D_k \) allows the result of the corresponding set \( F \) of cleartext operations to be retrieved. As it has been shown above, the availability of secure PHs is central to the delegation of computation. RSA [RSA78] is a well-known example of PH which allows multiplication and test for equality to be carried out on encrypted data (see Appendix C). A summary of the state of the art on PHs can be found in Section 3.5.

The choice of a particular PH depends on the type of the data to be dealt with:

- For qualitative (non-numerical) data, RSA resists chosen-plaintext attacks and provides a way for the handler to perform checks for equality on encrypted data.

- For numerical data, a PH is specified in Section 4.4 which is unconditionally secure against known-plaintext attacks and allows full arithmetic to be carried out by the handler on encrypted data. In [Dom96], a computationally secure PH offering similar features was presented.

### 4.2.2 Computation verifiability

As noted above, absolute confidence can only be attained by the owner if she repeats the whole handler’s claimed computation. But this makes delegation useless. Repetition of the claimed computation by the owner with low probability \( q \), say 10%, only detects fraud or overflow with probability \( q \). Parity checking is a more interesting alternative explained in Section 4.3, which can yield fraud or overflow detection probabilities as high as 50% (or more if intermediate results are verified).
4.3 Verification of computation through parity checking

Let the parity of an integer \( x \) be \( Z(x) = x \mod 2 \) The idea behind verifying handler computations through parity checking is that arithmetical operations can be mapped to Boolean operations on data parities; the parities of \( a + b \) and \( ab \) can be computed as

\[
Z(a + b) = Z(a) \oplus Z(b); \quad Z(ab) = Z(a) \cdot Z(b)
\]  

(4.1)

where \( \oplus \) denotes exclusive OR and \( \cdot \) denotes logical AND. Now, let the \textit{claimed expression} the expression that the handler claims to have computed on the data; let the \textit{computed expression} be the expression actually computed by the handler. Upon completing a computation on encrypted data, the handler returns the (encrypted) result together with a claimed expression to the data owner. The data owner can decrypt the result and obtain its parity; on the other hand, the data owner can also quickly compute the parity resulting from feeding the input data to the claimed expression (only quick Boolean operations are needed). If both parities differ, then handler fraud has been detected.

In [CDS98], we proved the following properties of the parity checking mechanism:

**Lemma 1 (Fraud detection probability)** If the claimed and the computed expressions differ, the probability of fraud detection by the owner is

\[
Pr(\text{detect}) = p(1 - p') + p'(1 - p)
\]  

(4.2)

where \( p \) and \( p' \) are, respectively, the probabilities of the claimed and computed expressions being odd.

**Proof:** For both the claimed expression and the computed expression do:

1. Consider the parity formula of the expression, which yields the parity of the result by using XOR and AND gates on the parities of input data.
4.3 Verification of computation through parity checking

Perform Boolean reduction on the formula so that it can be represented as a tree where the root is the parity of the result, leaves are parities of input data, nodes are two-input logical gates and for each node, both inputs are independent from each other. Let the estimated parity distribution for an input datum $d$ be $(1 - p_d, p_d)$ where $p_d$ is the probability of $d$ being odd.

2. Starting from the leaves, compute the parity distribution of every node output $o$ (intermediate result) in the tree from the distributions of node inputs $a$ and $b$, using the following rules derived from equations (4.1)

**XOR nodes:** $p_o = p_a (1 - p_b) + (1 - p_a)p_b$

**AND nodes:** $p_o = p_a p_b$

**OR nodes:** $p_o = p_a + p_b - p_a p_b$

(Note that the OR nodes may appear in the tree as a result of Boolean reduction)

If parity distributions of the results of the claimed and computed expressions are $(1 - p, p)$ and $(1 - p', p')$, the probability of the handler being caught is

$$Pr(\text{detect}) = Pr(\text{odd claimed}) Pr(\text{even computed})$$

$$+ Pr(\text{even claimed}) Pr(\text{odd computed}) = p(1 - p') + p'(1 - p)$$

**Theorem 2** For random input data, the probability of fraud detection by parity checking of the final result is tightly upper-bounded by $1/2$.

**Proof:** If input data are random, they have parity distribution $(1/2, 1/2)$. Additions introduce no parity bias but results of multiplications tend to be even (only odd times odd yields odd). This means $p \leq 1/2$ and $p' \leq 1/2$ in equation (4.2). Thus $Pr(\text{detect}) \leq 1/2$. If the claimed and the computed expressions contain only additions, then $p = p' = 1/2$ and $Pr(\text{detect}) = 1/2$. So the bound is tight. ■
**Theorem 3** For random input data, the probability of fraud detection by parity checking of the final result is tightly lower-bounded by \( \max(p, p') \), where \( p \) and \( p' \) are, respectively, the probabilities of the claimed and computed expressions being odd. If always-even claimed expressions are forbidden by the data owner, nonzero detection probability is guaranteed.

**Proof**: The probability of detection is given by equation (4.2). We first show that it is greater or equal than \( p' \). We have

\[
Pr(\text{detect}) = p(1 - p') + p'(1 - p) = p' + p(1 - 2p')
\]

So, for fixed \( p' \) the probability of detection can be regarded as a straight line which is a function of \( p \). If input data are random, from the proof of Theorem 2 one has \( p' \leq 1/2 \), so that the slope of the above straight line is \( 1 - 2p' \geq 0 \). Since \( p \geq 0 \), it holds that \( Pr(\text{detect}) \geq p' \). Now exchanging \( p \) and \( p' \) in the above argument, we obtain \( Pr(\text{detect}) \geq p \), which proves the lower bound. Since the bounds on \( p \) and \( p' \) given in the proof of Theorem 2 are tight, so is the bound on \( Pr(\text{detect}) \). Finally, if always-even claimed expressions are forbidden, then \( p > 0 \) and therefore \( Pr(\text{detect}) > 0 \).}

**Note 2** Even if always-even claimed expressions are forbidden, in data delegation the handler can fabricate for every \( \epsilon > 0 \) a claimed expression such that \( 0 < p < \epsilon \). But, for fraud to make sense, the computed expression is expected to be chosen on the basis of its usefulness to the handler, which means that \( p' \) takes a fixed value. Therefore, the lower bound \( \max(p, p') \)—and consequently the detection probability—cannot be made arbitrarily small by the handler if the computed expression is to remain useful.

To recognize a (forbidden) always-even expression, the data owner must check whether the associated Boolean parity formula is equivalent to 0. This check is very easy if the claimed expression is required to be in sum-of-products (SOP) form, where it amounts to checking that each AND term contains some input variable \( d \) and its complemented \( \bar{d} \). Note that it is trivial for the owner to make sure that the Boolean formula provided by the handler is in SOP form.
4.4 A new privacy homomorphism

We conclude that the data owner can be assured of a nonzero detection probability but is unable to compute \( Pr(\text{detect}) \) nor the lower bound of Theorem 3 because she does not know \( p' \). On the other hand, Theorem 2 gives an upper bound on the confidence attainable if verification of the final result passes. A way for the owner to increase the probability of detection (maybe above 1/2) is to verify several independent intermediate results. For example, if the claimed expression is regarded as a binary tree having input data as leaves and the result as root, then the owner may request from the handler the two encrypted intermediate results preceding the final result in the tree. If detection probabilities for such results are \( p_l \) and \( p_r \), then the probability of fraud detection is \( p_l + p_r - p_l p_r \). The procedure can be iterated by backtracking up the tree: if the owner requests also the four encrypted intermediate results preceding the previous two ones, then the probability of detection increases further. A trade-off is obvious: increasing the detection probability entails more verification work for the data owner.

4.4 A new privacy homomorphism

In this section, we propose the first PH preserving addition and multiplication that can be proven unconditionally secure against known-cleartext attacks, as long as the ciphertext space is much larger than the cleartext space.

In Subsection 4.4.1 the homomorphism is specified. In Subsection 4.4.2 a numerical example is given. Security is proven in Subsection 4.4.3. Subsection 4.4.4 illustrates the suitability of the new PH for secure delegation. Subsection 4.4.5 summarizes the discussion on the new PH.

4.4.1 Specification of the new PH

The PH proposed in this section can be described as follows. The public parameters are a positive integer \( d \) and a randomly chosen large integer \( m \) \((\approx 10^{200} \) or maybe larger\). The secret parameters are \( r \in \mathbb{Z}_m \) such that \( r^{-1} \mod m \) exists and a small divisor \( m' > 1 \) of \( m \) such that \( s := \log_{m'} m \) is a
(secret) security parameter. In this case the set of cleartext is \( T' = \mathbb{Z}_{m'} \). The set of ciphertext is \( T = (\mathbb{Z}_m)^d \). The set \( F' \) of cleartext operations is formed by addition, subtraction and multiplication in \( T' \). The set \( F \) of ciphertext operations contains the corresponding componentwise operations in \( T \). The PH transformations can be described as

**Encryption** Randomly split \( a \in \mathbb{Z}_{m'} \) into secret \( a_1, \ldots, a_d \) such that \( a = \sum_{j=1}^{d} a_j \mod m' \) and \( a_j \in \mathbb{Z}_m \). Compute

\[
E_k(a) = (a_1r \mod m, a_2r^2 \mod m, \ldots, a_d r^d \mod m)
\]

(4.3)

**Decryption** Compute the scalar product of the \( j \)-th coordinate by \( r^{-j} \mod m \) to retrieve \( a_j \mod m \). Compute \( \sum_{j=1}^{d} a_j \mod m' \) to get \( a \).

As encrypted values are computed over \( (\mathbb{Z}_m)^d \) at an unclassified level, the use of \( r \) requires that the terms of the encrypted value having different \( r \)-degree be handled separately — the \( r \)-degree of a term is the exponent of the power of \( r \) contained in the term. This is necessary for the classified level to be able to multiply each term by \( r^{-1} \) the right number of times, before adding all terms up over \( \mathbb{Z}_{m'} \).

The set \( F' \) of ciphertext operations consists of

**Addition and subtraction** They are done componentwise, i.e. between terms with the same degree.

**Multiplication** It works like in the case of polynomials: all terms are cross-multiplied in \( \mathbb{Z}_m \), with a \( d_1 \)-th degree term by a \( d_2 \)-th degree term yielding a \( d_1 + d_2 \)-th degree term; finally, terms having the same degree are added up.

**Division** Cannot be carried out in general because the polynomials are a ring, but not a field. A good solution is to leave and handle divisions in rational format by considering the field of rational functions: the encrypted version of \( a/b \) is \( E_k(a)/E_k(b) \).
Note 3 Unlike for the PH in Example 1, in our PH the cleartext space is unknown to the unclassified level, because the parameter $m'$ is secret. This will be useful to prove the security of our proposal. Notice that if the unclassified level is told which is the ciphertext space, then it needs no knowledge on the cleartext space to do encrypted computations. However, one of the troubles with Example 1 is that the unclassified level cannot be revealed which is the ciphertext space (giving away the ciphertext space $\mathbb{Z}_p \times \mathbb{Z}_q$ is equivalent to revealing the secret key $(p, q)$); therefore, knowledge of the size $m = pq$ of the cleartext space (or another common multiple of $p$ and $q$) is needed to reduce partial encrypted results.

4.4.2 Numerical example

This example is unrealistically small but it illustrates the computation of a formula including two additions and one multiplication, namely $(x_1 + x_2 + x_3)x_4$. We will take $d = 2$, that is, cleartexts will be split into two parts during encryption. The public modulus is chosen to be $m = 28$.

Classified level

Let $r = 3$ and $m' = 7$ be the secret key. Let

$$(x_1, x_2, x_3, x_4) = (-0.1, 0.3, 0.1, 2)$$

In order to suppress decimal positions, initial data are multiplied by 10, which yields the fractions $x_1 = \tilde{x}_1/10 = -1/10$, $x_2 = \tilde{x}_2/10 = 3/10$, $x_3 = \tilde{x}_3/10 = 1/10$ and $x_4 = \tilde{x}_4/1 = 2/1$. Numerators are randomly and secretly split mod 7 and are transformed according to the proposed PH. In this way, first and second $r$-degree terms are obtained

$$E_k(\tilde{x}_1) = E_k(-1) = E_k(2, 4) = (6, 8)$$

$$E_k(\tilde{x}_2) = E_k(3) = E_k(2, 1) = (6, 9)$$

$$E_k(\tilde{x}_3) = E_k(1) = E_k(4, 4) = (12, 8)$$
\[ E_k(\tilde{x}_4) = E_k(2) = E_k(3, 6) = (9, 26) \]

Encrypted data are forwarded to the unclassified level, along with their denominators: \((1,1)\) for \(E_k(\tilde{x}_4)\) and \((10,10)\) for the rest of data.

**Unclassified level**

First, do the additions by directly adding the numerators in the fractions, since the denominator is 10 for all data

\[
\sum_{i=1}^{3} E_k(\tilde{x}_i) = (6 + 6 + 12 \mod 28, 8 + 9 + 8 \mod 28) = (24, 25)
\]

The denominator of the sum is obviously \((10,10)\). Then, multiply by \(E_k(\tilde{x}_4)\)

\[
(E_k(\tilde{x}_1) + E_k(\tilde{x}_2) + E_k(\tilde{x}_3))E_k(\tilde{x}_4) = (24, 25) \times (9, 26)
\]

\[ = (0, 24 \times 9 \mod 28, 24 \times 26 + 25 \times 9 \mod 28, 25 \times 26 \mod 28) = (0, 20, 9, 6) \]

In this way, the numerator of the result has terms up to the fourth \(r\)-degree. The denominator of the product is \(10 \times 1 = 10\) for all terms. Return both numerator and denominator to the classified level.

**Classified level**

Compute

\[
(0 \times r^{-1} \mod m, 20 \times r^{-2} \mod m, 9 \times r^{-3} \mod m, 6 \times r^{-4} \mod m)
\]

\[ = (0 \times 19 \mod 28, 20 \times 19^2 \mod 28, 9 \times 19^3 \mod 28, 6 \times 19^4 \mod 28)
\]

\[ = (0, 24, 19, 26) \]

Now add all terms in the last step over \(\mathbb{Z}_{m'}\) to obtain \(6 \mod 7 = 6\). Thus, we have \((\tilde{x}_1 + \tilde{x}_2 + \tilde{x}_3)\tilde{x}_4 = 6 \mod m' = 6\). Finally, divide 6 by the denominator 10 returned by the unclassified level, so that the final result is \((x_1 + x_2 + x_3)x_4 = 0.6\).
4.4.3 Security of the new privacy homomorphism

**Definition 1** A privacy homomorphism is said to be unconditionally secure against a known-plaintext attack if, for any fixed number \( n \) of known cleartext-ciphertext pairs, the probability of successful decryption of a ciphertext for which the cleartext is unknown can be made arbitrarily small by properly choosing the security parameters of the homomorphism.

We will show in this section that the PH whose encryption function is given by expression (4.3) is unconditionally secure. First, it will be shown that, for a fixed number \( n \) of known cleartext-ciphertext pairs, the probability of randomly guessing the right key can be made arbitrarily small. Second, it will be shown that there is only a small probability that a ciphertext decrypts to the same cleartext using two different keys. Combining both results, unconditional security will follow. We next recall three known preliminary results:

**Lemma 2** Assume that divisibility of an integer by different primes is independent and that divisibility of randomly chosen integers by the same prime is independent. Let \( B \) be a positive integer. If positive integers \( c_1, \cdots, c_n \) are randomly drawn from the interval \((0, B)\), then

\[
\lim_{B \to \infty} \Pr\{\gcd(c_1, c_2, \cdots, c_n) = 1\} \approx \frac{1}{\zeta(n)}
\]

(4.4)

where \( \zeta(n) = \sum_{i=1}^{\infty} t^{-n} \) is Riemann's zeta function.

**Proof** : Let us first consider two unbounded positive integers \( c_1 \) and \( c_2 \). The probability that one of them is divisible by the prime \( p_i \) is \( 1/p_i \), and the probability that both of them are divisible by the same prime, assuming independence, is \( 1/p_i^2 \). Thus, the probability that they are *not* both divisible by \( p_i \) equals \( 1 - 1/p_i^2 \). If we assume divisibility by different primes to be independent, then the probability of coprimality becomes

\[
W_2 \approx \prod_{p_i} (1 - 1/p_i^2)
\]
Now consider $n$ unbounded positive integers $c_1, c_2, \cdots, c_n$. By a similar argument, the probability that they are coprime (i.e. that their gcd is 1) can be approximated as

$$W_n \approx \prod_{p_i} (1 - p_i^{-n})$$

Since $p_i \geq 2$ for any $i$,

$$\frac{1}{1 - p_i^{-n}} = 1 + p_i^{-n} + p_i^{-2n} + \cdots$$

If we take $p_1 = 2, 3, \cdots, P$, and multiply the series together, the general term resulting is of the type

$$2^{-a_2} 3^{-a_3} \cdots P^{-a_P} = t^n$$

where

$$t = 2^{-a_2} 3^{-a_3} \cdots P^{-a_P}$$

for $a_2 \geq 0, a_3 \geq 0, \cdots, a_P \geq 0$. A number $t$ will occur if and only if it has no prime factors greater than $P$, and then only once (the factorization of integers is unique). Hence,

$$\prod_{p_i \leq P} \frac{1}{1 - p_i^{-n}} = \sum_{(P)} t^{-n}$$

where the summation on the right-hand side extends over numbers formed from the primes up to $P$. These numbers include all numbers up to $P$, so that

$$0 < \sum_{t=1}^{\infty} t^{-n} - \sum_{(P)} t^{-n} < \sum_{P+1}^{\infty} t^{-n}$$

and the last sum tends to 0 when $P \to \infty$. Hence,

$$1/W_n \approx \lim_{P \to \infty} \prod_{p_i \leq P} \frac{1}{1 - p_i^{-n}} = \lim_{P \to \infty} \sum_{(P)} t^{-n}$$
\[ = \sum_{t=1}^{\infty} t^{-n} = \zeta(n) \]

Finally, if the integers \( c_1, \cdots, c_n \) are drawn from the interval \((0, B)\) and \( B \to \infty\), the last equality can be used and the result follows. \( \blacksquare \)

**Lemma 3** If \( \phi(m) \) is Euler’s totient function counting the number of integers less than \( m \) that are coprime with \( m \), then

\[ \phi(1) + \cdots + \phi(m) = \frac{3m^2}{\pi^2} + O(m \log m) \quad (4.5) \]

*In particular, the average order of \( \phi(m) \) is \( 6m/\pi^2 \approx 0.608 \).*

**Proof**: See [HW93], section 18.5. \( \blacksquare \)

**Lemma 4** Let \( d(n) \) be the number of divisors of a positive integer \( n \), counting 1 and \( n \). The average order of \( d(n) \) is \( \log n \).

**Proof**: See [HW93], section 18.2 \( \blacksquare \)

The first result concerning the security of the new privacy homomorphism against known-plaintext attacks regards the subset of keys consistent with the known cleartext-ciphertext pairs:

**Theorem 4** Consider a PH whose encryption function is given by expression (4.3). Let \( n \) be the number of random cleartext-ciphertext pairs known by the cryptanalyst. If the \( r \)-degree of all ciphertexts is greater than 1, then the size of the subset of keys consistent with the known pairs grows exponentially with \( s - n \) and has an expected value of at least \( \max(6(m')^{s-n}/\pi^2, 1) \), where \( s = \log m, m \). Cleartext-ciphertext pairs derived from the \( n \) known pairs using the homomorphic properties do not compromise the security of the PH.

**Proof**: Denote by \( d \) the maximal \( r \)-degree of ciphertexts in known message pairs. Let the \( n \) known random message pairs consist of cleartexts \( a_i \) and ciphertexts \( (b_{i1}, \cdots, b_{id}) \), for \( i = 1, \cdots, n \). The following construction shows that if \( n \) is not too large, then there exist several keys \( (\hat{r}, \hat{m'}) \) consistent with the \( n \) known pairs.
1. Randomly pick \( \hat{r} \) such that \( \hat{r}^{-1} \mod m \) exists. Clearly, all numbers co-prime to \( m \) are eligible; thus, there are \( \phi(m) \) candidates.

2. For \( i = 1, \cdots, n \) compute \( \hat{a}_{i1}, \cdots, \hat{a}_{id} \) such that \( \hat{a}_{ij} = b_{ij} \hat{r}^{-j} \mod m \).

3. Find \( \hat{m}' \) such that it divides \( m \) and verifies

\[
\hat{a}_{i1} + \cdots + \hat{a}_{id} = a_i \mod \hat{m}'
\]

for \( i = 1, \cdots, n \). A possibility (perhaps not unique) is to take

\[
\hat{m}' = \gcd \left( \sum_{j=1}^{d} \hat{a}_{ij} - a_i, m \right)
\]

If \( \hat{m}' \leq \max_{1 \leq i \leq n} a_i \) is obtained, then go to Step 1. Otherwise a key \((\hat{r}, \hat{m}')\) consistent with the known pairs has been obtained and the procedure is finished.

The probability of coming up with a good \( \hat{m}' \) at Step 3 of the above construction can be lower-bounded as

\[
Pr(\gcd \left( \sum_{j=1}^{d} \hat{a}_{ij} - a_i, m \right) > \max a_i) \geq Pr(\gcd \left( \sum_{j=1}^{d} \hat{a}_{ij} - a_i, m \right) \geq m')
\]

\[
\geq Pr(\gcd \left( \sum_{j=1}^{d} \hat{a}_{ij} - a_i, m \right) = m') = Pr(A)Pr[\gcd \left( \frac{\sum_{j=1}^{d} \hat{a}_{ij} - a_i}{m'}, m/m' \right) = 1 | A]
\]

\[
\approx \left( \frac{1}{m'} \right)^n \frac{1}{\zeta(n+1)} \approx \frac{1}{(m')^n}
\]  \hspace{1cm} (4.6)

where the last approximation is valid if \( n \) is not too small (say \( \geq 10 \)) and the \( \zeta \) approximation is obtained from Lemma 2. \( A \) is the event \( "m' \) divides \( m \) and all \( \sum_{j=1}^{d} \hat{a}_{ij} - a_i, \ i = 1, \cdots, n" \); clearly, by assumption \( m' \) divides \( m \), and the probability that \( m' \) divides a random integer (such as \( \sum_{j=1}^{d} \hat{a}_{ij} - a_i \)) is \( 1/m' \); thus, \( Pr(A) = (1/m')^n \). Let us check that Lemma 2 can be used:

1. In the last \( \gcd \) computation, \( m/m' \) is random (because \( m \) is) and the rest
of numbers can be viewed as being randomly drawn from the interval $(0, dm/m')$, since they depend on a random number $\hat{r}$.

2. $dm/m'$ is large since $m'$ is a small divisor of $m$.

Now the above construction can be run for $\phi(m)$ different values of $\hat{r}$. This means that the expected number of keys $(\hat{r}, m')$ consistent with the known pairs is at least

$$\max\left(\frac{\phi(m)}{(m')^n}, 1\right) \approx \max\left(\frac{6}{\pi^2} \frac{m}{(m')^n}, 1\right) = \max\left(\frac{6}{\pi^2} (m')^{n-1}, 1\right)$$

where the approximation is obtained from Lemma 3. Finally, to prove the last assertion of the theorem, imagine that two known pairs are added, subtracted or multiplied by the cryptanalyst to generate a new cleartext-ciphertext pair. If this new pair is input to the gcd computation at Step 3, it is easy to see that the gcd value remains unchanged. Thus only genuine randomly split cleartext-ciphertext known pairs are to be taken into account. ■

**Note 4** Notice that random cleartext splitting is central to the proof of Theorem 4. Imagine that no splitting is done (i.e., $d = 1$) and that some pairs $(a_i, b_i)$ are known. Then $b_i = a_i r \mod m$, for all $i$, so $r$ cannot be chosen at random.

**Note 5** $m'$ must be considered as a secret parameter for the above proof of Theorem 4 to be correct. Otherwise, consistent keys would be only those having the form $(\hat{r}, m')$ and their number would be much smaller. Since unconditional security is claimed, it is assumed that the enemy cryptanalyst can compute all divisors $\hat{m}'$ of $m$, but cannot decide which divisor is actually being used; the only clue is that $\hat{m}' > \max a_i$, as reflected in derivation (4.6).

**Note 6** Instead of using the average approximation from Lemma 3, one might think of using the Rosser-Schoenfeld lower bound on $\phi(m)/m$ ([RS62], p. 72) in the proof of Theorem 4. However, being based on a smooth function of $m$, this bound is often too conservative and can lead to a serious underestimation
of the size of the subset of consistent keys. Due to this drawback and the probabilistic nature of the proof, it seems more realistic and natural to use the expected value of \( \phi(m) \).

**Corollary 1** If leakage of \( n \) cleartext-ciphertext pairs is to be tolerated, then the probability of success for a known-cleartext attack with unlimited computing power can be made arbitrarily small by a proper choice of the security parameter \( s = \log_{m'} m \).

**Proof:** Assume that the cryptanalyst has enough computing power to enumerate the subset of keys consistent with the known pairs. From Theorem 4 it is clear that the best attacking strategy is randomly guessing the key, which has a probability of success at most equal to

\[
\begin{cases}
\pi^2(m')^{n-s}/6 & \text{if } s > n \\
1 & \text{if } s \leq n
\end{cases}
\]

For any \( \epsilon > 0 \), there exists a value of \( s \) that makes this probability smaller than \( \epsilon \). ■

The second result concerning the security of the proposed PH regards the behavior of keys:

**Theorem 5** The expected probability that any two keys \((r_1, m'_1)\) and \((r_2, m'_2)\) decipher a random ciphertext to the same cleartext is \( O((\log m)/m) \). Therefore, this probability can be made arbitrarily small by increasing \( m \).

**Proof:** Let the two keys be \((r_1, m'_1)\) and \((r_2, m'_2)\). Let \((b_1, b_2, \ldots, b_d)\) be a randomly chosen ciphertext. Assume that both keys decrypt to the same cleartext \( a \). Then from the specification of the PH (Subsection 4.4.1):

\[
a = b(r_1^{-1}) \mod m'_1 = b_1 r_1^{-1} + b_2 r_1^{-2} + \cdots + b_d r_1^{-d} \mod m'_1
\]

\[
= b(r_2^{-1}) \mod m'_2 = b_1 r_2^{-1} + b_2 r_2^{-2} + \cdots + b_d r_2^{-d} \mod m'_2
\]

where the inverses are modulo \( m \). Three cases must be considered:
4.4 A new privacy homomorphism

- If \( r_1 = r_2 = r \) then \( m'_1 \neq m'_2 \) since both keys are assumed to be different. In that case, equation (4.7) holds only if both \( m'_1 \) and \( m'_2 \) divide \( a - b(r^{-1}) \). Assuming that \( a \) is such that \( m'_1 \) is a divisor, from Lemma 4 the expected probability that \( m'_2 \) is also a divisor is \((\log m)/m\).

- If \( m'_1 = m'_2 = m \) then \( r_1 \neq r_2 \) since both keys are assumed different. In that case, equation (4.7) holds only if \( m \) divides \( b(r_1^{-1}) - b(r_2^{-1}) \). From Lemma 4, this happens with an expected probability

\[
\frac{\log |b(r_1^{-1}) - b(r_2^{-1})|}{|b(r_1^{-1}) - b(r_2^{-1})|} = O((\log m)/m) \tag{4.8}
\]

- If \( m'_1 \neq m'_2 \) and \( r_1 \neq r_2 \), equation (4.7) implies that there exists an integer \( a \) such that \( m'_1 \) divides \( b(r_1^{-1}) - a \) and \( m'_2 \) divides \( b(r_2^{-1}) - a \). Using Lemma 4, the probability of such an event can be upper bounded by

\[
\max_{0 \leq a \leq \min(m'_1,m'_2)-1} \left( \frac{\log |b(r_1^{-1}) - a|}{|b(r_1^{-1}) - a|}, \frac{\log |b(r_2^{-1}) - a|}{|b(r_2^{-1}) - a|} \right) = O((\log m)/m) \tag{4.9}
\]

Thus in all cases the expected probability of obtaining the same cleartext from decryption of the same ciphertext using two different keys is \( O((\log m)/m) \). ■

**Note 7** It should be noted that the security provided by the proposed scheme is *unconditional* because the subset of keys consistent with the known pairs is kept large and any two different keys yield different cleartexts from the same ciphertext with a high probability. Further, in the proof of Corollary 1 it is assumed that an infinitely powerful cryptanalyst can enumerate the subset of consistent keys, but no easy way to do this is obvious. In computational terms, even if there exists only one consistent key (probability of random key guessing equal to 1 for infinite computing power), this does not mean that such a key is easy to find.

Table 4.1 illustrates the dependency between parameters with several example choices. There are at least two scenarios for parameter design:
<table>
<thead>
<tr>
<th>$n$</th>
<th>$s$</th>
<th>$l(m')$</th>
<th>$l(m)$</th>
<th>Prob. rand. key guessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>20</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>20</td>
<td>120</td>
<td>$\approx 1.64 \times 10^{-20}$</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>20</td>
<td>220</td>
<td>$\approx 1.64 \times 10^{-21}$</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>5</td>
<td>250</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>5</td>
<td>255</td>
<td>$\approx 1.64 \times 10^{-9}$</td>
</tr>
<tr>
<td>50</td>
<td>53</td>
<td>5</td>
<td>265</td>
<td>$\approx 1.64 \times 10^{-10}$</td>
</tr>
</tbody>
</table>

Table 4.1: Example parameter choices for the proposed privacy homomorphism.

- If $n$, $m'$ and the probability of random key guessing are specified as requirements, then suitable values for $s$ and $m$ must be determined.

- If $m'$ and $m$ are fixed in advance (i.e. the PH is fixed), then the probability of random key guessing can be computed for each number $n$ of known cleartext-ciphertext pairs. If more and more random pairs are leaked over time, then a proactive key renewal scheme should be enforced by the classified level in order to keep the probability of random key guessing smaller than an alarm threshold set beforehand.

### 4.4.4 Suitability of the new PH for secure delegation

The new PH is specifically designed for computing or data delegation on numerical data, since full arithmetic can be performed on encrypted data.

Note that data delegation has stronger security requirements than computing delegation. In computing delegation the data handler only sees ciphertext. However, in data delegation the data owner decrypts the final result of computations and returns it as cleartext to the data handler. This means that in data delegation each time the data owner returns one decrypted result to the data handler, the data handler learns one new cleartext-ciphertext pair (or two
if what is returned is an unreduced fraction). Thus, it is essential that PHs used for data delegation be secure against known-cleartext attacks. The new PH fulfills this requirements. However, the data owner should take care that the number $n$ of returned decrypted results is not too large to become unsafe (given the security parameter $s$ of the homomorphism, see Corollary 1). Once the safety limit has been reached, no more decrypted results should be returned by the data owner, unless she decides to change the key of the homomorphism and reencrypt all delegated data under the new key.

### 4.4.5 Summary

The features of the proposed homomorphism can be summarized as follows

- Addition, subtraction, multiplication and division can be carried out on encrypted data at an unclassified level.

- The proposed homomorphism is the first one to allow full arithmetic while being unconditionally secure against known-cleartext attacks. Unconditional security against known-cleartext attacks can be proven provided that cleartext splitting is always used when encrypting (i.e. $d > 1$) and the ciphertext space is much larger than the cleartext space. Pairs that are derived from random pairs using the homomorphic properties do not compromise the security of the PH.

- Encryption and decryption transformations can be implemented efficiently, because they only require modular multiplications. Note that no exponentiation is needed, because the powers of $r$ can be precomputed. Unlike exponential ciphers, the proposed PH can be fast even if the ciphertext space is very large.

- A ciphertext with $r$-degree $d$ is about

$$d \frac{\log m}{\log m'} = d \log m, m = ds$$
times longer than the corresponding cleartext. Although this is a storage penalty, a choice of \( d = 2 \) at the time of encryption should be affordable while remaining secure. For a given \( r \)-degree, the ciphertext expansion grows linearly with the security parameter \( s \).

- The equality predicate is not preserved, and thus comparisons for equality cannot be done at an unclassified level based on encrypted data. A given cleartext can have many ciphertext versions for two reasons: A) random splitting during encryption; B) the unclassified level computes over \((\mathbb{Z}_m)^d\) and only the classified level can perform a reduction to \(\mathbb{Z}_{m'}\) during decryption.
Chapter 5

Implementation Issues for Secure Delegation

Secure delegation can be viewed as a distributed computing problem and can be implemented using a variety of computer technologies. In this chapter, we analyze technologies that can be relevant and highlight their strong and weak points in connection to secure delegation. This chapter is the implementation-level parallel of Chapter 3, which analyzed cryptographic techniques for solving secure delegation at the theoretical level. The implementation-level analysis requires giving some background on distributed computing, client server computing, objects and object oriented systems. Section 5.1 sketches the evolution toward distributed computing. Section 5.2 exposes some background on the predominant distributed computing paradigm, namely the client-server model. Section 5.3 deals with distributed objects and reviews technologies for managing them. Section 5.4 justifies the choice of CORBA as our development architecture. Section 5.5 discusses the CORBA security features. Section 5.6 deals with auxiliary implementation choices.
5.1 Toward distributed computing

In the last decades, information technology has changed dramatically. We passed from manual processing to batch processing, on-line processing, networked processing, client-server processing, and finally distributed object processing. This evolution has been possible as a consequence of the improvements in the computing systems and the appearance of computer networks. Also, the increasing complexity of computing systems has stimulated the search for new, more manageable paradigms.

5.2 The client-server paradigm

In a classical situation when two systems have to communicate, and one of them wants the other to perform some computation, a communication channel has to exist. The first system is called client and the second is called server. This is the most popular model for describing distributed computing. The communication channel can be assimilated to a network [Rag93] [Ste90].

Three different approaches can be basically considered for building distributed systems:

**Socket:** The socket approach is the less abstract, and is basically an interface to deal with network protocols. The basic element of abstraction are communication channels.

**RPC:** The Remote Procedure Call (RPC) approach is a method to make remote function calls. In RPC, the basic elements of abstraction are functions (intermediate level).

**DO:** Distributed Objects (DO) is the object oriented approach to implement distributed systems, the basic elements of abstraction are remote objects (higher level).

In the next three subsections we review several techniques for developing
client/server applications in order to illustrate key aspects of distributed programming.

5.2.1 Sockets

Distributed applications have traditionally been developed using network programming interfaces such as sockets [Ste90]. Sockets were originally developed for the BSD Unix system but are widely available on many platforms. A socket is considered an abstraction used as a communication endpoint that is bound to an address on a communication domain. A communication domain is a logical grouping of protocols based on similar communication and naming conventions. The TCP/IP protocols are considered the most important communication domain used by the socket approach.

Without loss of generality for our discussion we consider sockets to be primarily regarded as a programming interface to the TCP/IP communication domain. From the application point of view, a socket is used to access a service. The service can be provided by the local system or by another system in the network. The socket abstraction only covers the transport of information between two points and the interpretation of this information has to be done explicitly by the application. The socket interface is considered a procedural mechanism and is defined by a set of C functions as the programming interface. The functions basically establish or terminate connections, and send or receive data. Sockets are a relative low-level interface. A typical client-server communication is depicted in Figure 5.1:

1. The server creates a socket, binds an address to it and waits for requests

2. The client creates a socket and connects to the server address

3. The client sends a message to the server

4. The server sends a message to the client
5.2.2 Remote Procedure Calls

A second approach to construct distributed applications is represented by the RPC mechanism. The semantics of RPC is considered equivalent to a function call mechanism where the functions are out of the context of execution of the application. The main benefit of this approach is that applications can be developed without having to deal with details of connection establishment, message transmission and reception, and differences between data representation on different computer architectures.

The RPC implementation proposed by Sun Microsystems [Rag93] can be considered representative of this paradigm, and is used in what follows. The tree main components of the Sun Microsystems approach to implement RPC are:

- External data representation (XDR). XDR is used to represent data structures in a machine-independent form.

- Portmap routines and service. These are used to register the functions that a server implements and to locate servers for the functions called by the client.

- Core RPC routines. They are used to construct calls and to serve results to/from remote functions.
The programming interfaces are C functions, and the RPC is considered as a procedural paradigm with a procedural programming interface. The typical connection sequence of an RPC mechanism is depicted in Figure 5.2:

1. The server of a function registers it to the portmapper
2. The server wait for calls on its port
3. The client asks for a server port
4. The client receives the server port
5. The client makes a request on the port
6. The client receives the result from the server or an exception condition

A major contribution of this approach is that the programmer does not need to worry about marshalling and unmarshalling messages. This task is done by the system. A second advantage is that the process of locating the function is partially done by the system.

5.2.3 Distributed objects

Distributed object (DO) technologies try to incorporate the main benefits of the object oriented paradigm to the construction of distributed systems. Some
of the benefits of the use of Object Oriented (OO) technologies are: encapsulation, abstraction, and polymorphism. These characteristics are useful to manage the increasing complexity of systems by reducing the complexity of software design and allowing software reuse. Each approach supports these principles in different degrees.

In a distributed object scheme we have three fundamental and basic operational steps that are covered in distinct forms by different propositions. The three steps are object location, object activation and method call.

The distributed object paradigms can be characterized through different groupings and location of functionalities and by differences in the temporal sequence of operations to be performed.

In a DO approach, a broker acts as a message passing medium and is responsible for the transportation of calls from the client application to the target object, as well as the transportation of results and any exception condition back to the client application. In order to avoid the client having to know the location of a server object, the broker implements a location/registering mechanism. For our purpose and for simplicity we consider that when the broker receives a call for an object, the object is activated if it was not active and the call is passed to it. The broker knows about all available servers and the objects they implement because the objects register themselves to the broker.

The client stub is located in the client space, which implements the same interface as the remote server object. Instead of using the server object, the client calls the corresponding method of the client stub. On behalf of the client, the stub object transparently converts the call parameters to a platform-independent format. It then builds a request, sends it to the broker, and awaits the result. The client stub then converts the result to the local environment format, thus ending the call.

The server skeleton is located in the server space and has a complementary function relative to the client stub and broker. The server skeleton acts as a client to the object implementation being called; it translates the call
parameters from the platform-independent format to the server object plat-
form format and the result from the server object platform to the platform-
independent format. The object adapter (OA) is responsible of the activation
and management of the object implementation. The object implementation
receives the call and then returns the response to the broker to deliver it to
the client.

The sequence of steps for a method call in a simplified DO system is de-
picted in Figure 5.3:

1. The client makes a method call through an object stub, the call goes to
   the Broker. The Broker hands the call to an OA.
2. The OA activates the implementation. The implementation registers
   itself.
3. The object implementation is ready to receive requests.
4. The OA activates the server stub (skeleton) for the object implementa-
   tion
5. The OA passes the method invocation to the object implementation
   through the object skeleton.
6. The object implementation returns a result or an exception to the client.

5.3 Distributed object technologies

To support distributed technologies, several architectures have been proposed.
In the field of distributed object technologies, the two more influential pro-
posals are DCOM (Distributed Component Object Model, [BK98]) from Mi-
crosoft and Common Object Request Broker Architecture (CORBA, [OMG96])
from OMG. The first because of the large installed base of systems, and the
second because of the large number of independent supporters.
5.3.1 The DCOM architecture

DCOM is a distributed incarnation of the Component Object Model (COM) that is built on top of the DCE RPC mechanism with some extensions to support remote object calls. A DCOM server can incarnate object instances of multiple object classes. A DCOM object instance can be accessed by different interfaces, each one representing a different view or behavior of the object. It is considered that an interface represents a set of functionally related methods. A client accesses a DCOM object instance by acquiring a reference to one of the interfaces of the object.

The client reference to an interface is considered as a pointer to the address space of the client. The memory layout for the address space associated with the interface is standard and equivalent to the C++ virtual function table. A method of an object is accessed through the pointer of an interface. DCOM uses the client-server approach for communication – as stated before the DCE RPC mechanism. The use of an RPC causes the DCOM to inherit the advantages and disadvantages of RPC.

The functionality provided by a server is encapsulated in objects and accessed through their interfaces. The interfaces of a DCOM object are described in an interface definition language (IDL). In summary, the client interacts with a server by invoking methods described in IDL and implemented in the server.

Figure 5.3: Typical sequence diagram for the DO approach
IDL can represent some object oriented features: data encapsulation, polymorphism, and single inheritance.

To invoke a remote method in a DCOM object instance of a server, the client makes a call through the client stub. The stub marshalls the parameter in network data representation (NDR) format, constructs a request message, and ships the message to the server. The server receives a message, unpacks the message by demarshalling the parameters and determining the method, interface and object, and finally calling the method in the interface of the object instance provided by the DCOM server. Each object class or interface of an object class have a global unique identifier (UUID). For the interface the UUID is called interface identifier (IID), and for the object class the UUID is called class identifier (CLSID).

### 5.3.2 The CORBA architecture

The Common Object Request Broker Architecture (CORBA) is the implementation of the architecture defined in the Object Management Architecture (OMA) proposed by the object management group (OMG) consortium. The core of the CORBA is Object Request Broker (ORB) that acts as an object bus over which objects transparently interact with other objects that could be located locally or remotely. The view that the outside world has about an object is represented by an interface composed of some methods. A particular instance of an object is identified by an object reference.

The client of a CORBA object instance acquires the object reference to the object, and uses this reference as a handler to perform method calls—as if the object was located in the same address space of the client. The ORB is responsible for all the mechanisms required to find an object implementation, prepare it to receive a request (activate the object and send the request to the object instance) and carry the reply back to the client. The object instance interacts with the ORB through either an Object Adapter (OA) or through the ORB.

The interfaces of the CORBA objects are defined in the OMG Interface
Definition Language (IDL) and clients interact with a server by calling methods described in IDL. The IDL has some object oriented features such as: encapsulation, polymorphism and multiple inheritance. Also IDL can specify exceptions. CORBA has no restriction about how the ORB transports the method calls; in fact there are two approaches in the specification: one using DCE RPC and the other using the Internet Inter ORB Protocol IIOP that is mandatory and is used for the purpose of interoperability between different implementations of ORBs.

To invoke a method in a remote object, the client makes the call through the object stub, the object stub packs the call parameters into a request message and invokes a wire protocol to ship the message to the server. At the server side, the wire protocol delivers the message to the server skeleton (i.e. server stub), which unpacks the request message and calls the actual function on the object.

When an instance of an object in activated, it is registered in an implementation repository. More precisely, the association between an interface and the implementation of an object with this interface is registered.

5.4 Comparison and interim discussion

The main advantage of using DCOM is that it is perfectly integrated in the Windows platform and thus enjoys a widespread dissemination. In addition, DCOM offers pass by value and pass by reference semantics for the parameters of method calls. Since DCOM can be viewed as an extension of COM, it benefits from the expertise with COM. Also, this architecture allows to dynamically inject code to the client thanks to the predominance of the X86 hardware platform.

The main shortcomings are:

- DCOM does not have real interface inheritance, but actually uses aggregation and containment to simulate it. Thus programming is difficult and error-prone.
• Object instances are stateless and the references to objects are valid only within the current connection.

• DCOM is now in its first generation, so it has not yet reached a sufficient maturity level.

• The IDL is complex and too much procedural.

• The use of code injection is practical as long as it is limited to a single hardware platform (X86). Providing code injection for multiple binary formats is difficult and complex.

The main advantage of the CORBA architecture is the possibility of using it in highly heterogeneous environments (multiple hardware platforms, operating systems, and programming languages). Also, CORBA has attained a high level of maturity—the third generation is near to completion. Another strong point is that CORBA is very similar to an open standard: being controlled by a committee (OMG), it is in principle less influenced by marketing or commercial forces. Finally, the CORBA specification is clear and modular and offers a large number of services; several interoperative implementations exist, some of which are free.

The main disadvantages of CORBA are:

• The IDL language adds an extra level of complexity (this is a problem shared with DCOM).

• The approval of new improvements to the current specification is likely to be slow, as it is dependent on a committee.

• There is no pass by value semantic.

We have selected the CORBA architecture to implement the prototype described in Chapter 6. The above arguments have been taken into account to make the decision, and especially the fact that there are public implementations that allow free experimentation. CORBA is a really object oriented
approach for the development of distributed object systems. Its model is clear and robust and permits adding new functionalities and doing partial modifications. The increasing number of developers and independent manufacturers who support this technology is also a non-technical factor that should not be overlooked.

5.5 Security in CORBA

Having selected the CORBA approach as our main implementation environment, it is necessary to consider how CORBA can incorporate security. The CORBA ORB by itself only offers a minimal security functionality. The CORBA services specification [OMG98] has proposed a generic framework for security and proposes a security reference model. The security reference model describes how and where a secure system enforces security policies; in other words, it is considered a meta-model of security in which different instances of security policies can be characterized. A security policy defines:

- The conditions in which active entities (client objects) access objects.
- The kind of authentication for users or entities acting on their behalf (what they can do and whether they can delegate their rights).
- The security of communication between objects, the trust between them, and the quality of protection for data flowing between them.
- The accountability of the activities relevant to the security enforcement.

The scenario where security is considered is the following. On behalf of a user or active entity, a client communicates with a target object using the ORB. We say that a client object and a target object are in a security association, and there are four places for making security decisions: at the client, at the client invocation/reply, at the target invocation/reply, and at the target object. A combination of decisions at each of the aforementioned places configures a security policy. At each place, decisions are made based on the context and
on the value of attributes. An active entity must establish its rights to access objects in the system. It must either be a principal, or a client acting on behalf of a principal. A principal is a human user or system entity that has an identity that is registered to the system and can be authenticated. A principal can have more than one identity.

Another aspect of security policy is the delegation of privileges, which occurs when the immediate target object has to make requests to other objects. The set of objects to which a policy applies are called policy domain.

The CORBA security model reflects the current state of distributed systems security, and has therefore the problems pointed out in Section 2.1. Our approach is to use the minimal functionality that is incorporated on the ORB; most aspects of security are explicitly controlled in the application. For our purposes we need:

1. Secure invocation between client and target objects.
3. Access control.
4. Unrestricted delegation.

These requirements can be covered by using the SSL approach.

5.5.1 Secure socket layer

The Secure Socket Layer (SSL) [FKK96] protocol, originally proposed by Netscape, and its successor, the Transport Layer Security (TLS) [DA98] protocol, offer communications privacy over the Internet. The aforementioned protocols allow client/server applications to communicate in a way that is secure against eavesdropping, tampering, or message forgery. Both protocols are considered to be in the upper sublayer of the transport layer (sitting on top of the socket level interface). The basic functions they provide are:

- Verification of the client identity
• Verification of the server identity

• Encryption of the data exchange between client and server

5.6 Auxiliary implementation choices

In this section, several auxiliary implementation choices for the prototype described in Chapter 6 are discussed and justified.

Implementing the privacy homomorphisms mentioned in Chapter 4 requires handling large integers (more than 100 decimal digits long).

Our first idea was to use a library combining basic mathematical and cryptographic primitives. Some versions of CryptoLib [Lac] were tried which failed to meet our interoperability and speed requirements. Another reason for discarding CryptoLib is that this library is subject to U.S. export restrictions, so it cannot be used outside the U.S.

The second choice was the LiDIA[Gro98] package, which allows to use large integers even if it does not contain specific cryptographic primitives. The main advantages of LiDIA are:

1. Fast prime number computation.

2. Good implementation of large integer arithmetic. It performs better than CryptoLib.

3. C++ implementation. The fact that it is implemented in an object oriented language makes integration in our prototype fairly easy.

4. Ability to handle several mathematical objects (finite fields, elliptic curves, etc.) which may turn out to be useful in the future.

LiDIA was actually used in the first version of our prototype. However, the current version does not use it any longer. The final choice is the GNU Multiprecision Package (GMP, [Gra96]). GMP is a portable library written in C for arbitrary-precision arithmetic on integers, rational numbers, and floating-point
numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types. GMP is actually a basic component of LiDIA, so that, for our purpose, it is simpler and shares most of the above advantages.

Finally, C++ was chosen as the implementation language, mainly for interoperability reasons. UNIX was selected as the environment for development and initial deployment.
Chapter 6

A Prototype for Secure Delegation

A prototype christened Dikē (blind Greek goddess of justice and social order, and also Delegation of Information without Knowledge Exposure) has been patented which allows secure delegation (see [CDS96], [CDS97a], [CDS97b], [CDS97c], [DS97] and [DFD98]). Our intention in the construction of a prototype was twofold:

1. To evaluate programming paradigms and tools to construct secure distributed computing systems.

2. To test the protocols and algorithms resulting from our work on privacy homomorphisms and computation verification applied to the secure delegation problem.

To materialize the prototype we have selected the manipulation of confidential statistical data as the specific application field of our research on secure delegation. However, other application fields are conceivable and should be considered in future research; for instance, secure delegation in the context of electronic voting schemes [Bor96]. For an introduction to the UML notation used in this Chapter see Appendix A; for more details see [BRJ98] or [Obj97].
Section 6.1 describes the implementation requirements that have been considered as a starting point for the prototype. Section 6.2 outlines the architecture of the system from different points of view. Section 6.3 goes into implementation issues.

6.1 Requirements

This section exposes the requirements that the system must satisfy. They are obtained from the target problem (management of confidential statistical data) and from our proposed solution (secure delegation of computing and data).

6.1.1 Non functional requirements

In Chapter 4, the requirements and the solution for secure delegation have been dealt with at a conceptual level. In order to produce a prototype, some additional design considerations are in order:

- The application has to be used in a distributed computing environment, so it should be possible to distribute the overall functionality in different computer settings.

- The data, user information, usage logs, etc. should be held in a permanent repository.

- The use of the system should be easy to understand. In fact, it should be as self-explanatory as possible.

- The execution environment consists of Unix machines. However, Windows compatibility should be considered at the design stage.

- There must be a user interface that can manipulate encrypted data.

- The communication between the different components has to be performed in a secure setting.
6.1.2 Types of users

From the two models of delegation that have been identified, it is possible to derive the corresponding types of users of the system (i.e. computing user and data user). Analogously, the need for system management originates another type of user: the system manager. Finally, data management is a function to be performed by the data owner; this yields a fourth type of user: the data manager. Thus, we have the following user types:

**Computing user** This type of user is a client of the system and has to be capable of sending encrypted statistical data to the system, asking the system to perform computation on the data set, requesting the encrypted results, and decrypting results locally.

**Data user** This type of user is a client of the system and has to be capable of requesting encrypted statistical data, making computations on them, and requesting decryption of results of the locally performed encrypted computations.

**Data manager** It is responsible for managing the association between users, data, logs, and security parameters with the statistical data.

**System manager** It is responsible for creating/deleting users of the system and managing the log files generated by the use of the system. It has no responsibility on who accesses what data or on the statistical data stored on the server.

6.1.3 Statistical data

The statistical data that can be manipulated by the system are of two kinds, and are dealt with as tables:

**Microdata** A microdata set can be viewed as a table where rows correspond to individuals and columns correspond to variables. Each variable has an associated label that explains its meaning. There are one or several key
variables (direct or indirect identifiers of the individual) and one or more sensitive variables (variables that contain information on the individual). Variables can be qualitative or quantitative; in general, key variables are qualitative.

**Macrodata** Also known as contingency tables. A tabulation of variables $A$ and $B$, where $A$ takes values $a_1, \ldots, a_m$ and $B$ takes values $b_1, \ldots, b_n$, can be viewed as a set of rows $(a_i, b_j, f_{ij})$, where $f_{ij}$ is the joint frequency of values $a_i$ and $b_j$. This set of rows is actually a table with $m \times n$ rows and three columns. The third column of the table can be viewed as a quantitative variable (joint frequency), and the first and second columns of the table correspond to variables $A$ and $B$, which may be quantitative or qualitative. The first and second columns of the table have explanatory labels associated with them.

### 6.1.4 Privacy homomorphisms

The homomorphisms used to encrypt the tables used by the system are two:

**RSA:** Data encrypted with the RSA homomorphism (see Appendix C) allow two basic operations. One is comparison for equality between two encrypted values, and the other is multiplication of two encrypted values.

**JR:** It is the homomorphism proposed in Section 4.4, which will be denoted by JR in what follows. With the JR homomorphism, addition, subtraction multiplication and fraction division can be performed on two encrypted values.

### 6.1.5 Encryption of tables

There are three options related to encryption of columns and labels in a table:

- No encryption at all.
• Encryption under the RSA homomorphism. RSA allows comparison for equality and multiplication to be carried out on encrypted data. Therefore, RSA is a suitable homomorphism for encrypting qualitative variables or labels, which are mainly used for locating records.

• Encryption under the JR homomorphism. This homomorphism provides complete arithmetic on encrypted data but does not allow comparison for equality (two identical cleartexts can result in two different ciphertexts). Therefore, the JR homomorphism is useful mainly for quantitative variables.

6.1.6 Generic concerns

Data users, computing users and data managers can be located in computing environments other the one containing the data that (data and computing) users manipulate. The communication between the components that are located in different computing environments takes place in encrypted form. Encapsulation of functionalities in a distributed computing environment has to be performed in accordance with the different user types of the system.

The characterization of the identity of a user of the system is of critical importance, as most security functionalities depend on it. The identity of a user is determined by a certificate. The certificate can be issued by the system or by some certification authority recognized by the system (see [RCMG97] for a discussion on certification authorities).

The system associates each user with several security sets. A security set is linked to a table (statistical data): in fact, the security set consists of a key for each privacy homomorphism (RSA, JR), a reference to an encrypted version of the table (some columns may be encrypted under RSA and some under JR) and a reference to the original table.

In order to be able to carry out a controlled release of confidential data, the system must register all operations performed by users on the data contained in the system. Those logs are the starting point to implement a policy for
disclosure and verification of results.

6.2 System architecture

The architecture of the system is characterized by the following views of the system: the use case view, the design view, the process view, the implementation view, and the deployment view.

The use case view of the system encompasses the use cases that describe the behavior of the system as seen by each of its users. In this view we use use case diagrams to model the static aspects of the system and interaction diagrams, statechart diagrams, and activity diagrams to model the dynamic aspects.

The design view of the system encompasses the characterization of the classes, interfaces and collaborations that determine the context of the problem and the solution. The class diagrams and object diagrams model the static aspects of the view, and interaction diagrams, statechart diagrams and activity diagrams model the dynamic aspects of this view.

The process view of the system has to do with the processes and threads that determine the synchronization and the concurrencies of the system. The same diagrams are used as for the design view, but the active classes and objects that represent processes and threads of execution are highlighted.

The implementation view of the system deals with the components that are used to assemble and release the physical system. Component diagrams are applied to model static aspects, and interaction diagrams, statechart diagrams and activity diagrams to model the dynamic aspects.

The deployment view encompasses nodes that form the system’s hardware topology on which the system executes. Deployment diagrams are used to model static aspects, and interaction diagrams, activity diagrams, and statechart diagrams to model the dynamic aspects.

For the sake of clarity and brevity we will limit ourselves to specifying the two highest level views for Dikè, namely the use cases and the design views.
6.2.1 Use cases view

To construct the use case view of the system, it is necessary to determine the different types of users (actors) of the system and also the use case associated to each one. We also devise use cases that represent commonalities between use cases associated with the actors.

Actors

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. The actors represent categories of users that interact with use cases.

The analysis of the requirements in the previous section shows that there are four actors that can interact with the system:

- The data manager is the owner of some data. It can put new data in the system, determine who can access the data, and how the data can be used.

- The data user is the one that can use the data that owners grant access to. The data user can retrieve data and security parameters used to permit manipulation of the data.

- The system administrator is the third actor and is responsible for managing the potential users of the data sets, as well as the logs that record use of the data by the users.

- The computing user is the owner of some data and needs some computing resources to manipulate the data. It sends to the system encrypted data and asks for some computation to be performed by the system on the data.

With the requirements and the previous description of the kinds of user that the system has to interact with, and of what the users expect from the system, we define the following actors: data manager, data user, computing
user and system manager. Figure 6.1 shows a graphical representation with a label for each of the actors of the system.

![Graphical Representation of Actors](image)

Figure 6.1: Graphical representation of the different actors

**Use cases**

We next specify from the system’s viewpoint how to group the functionalities that the actors expect from the system. In this way, the main use cases of the system are obtained. Table 6.1 shows the main use cases associated with the corresponding actor.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Manager</td>
<td>Manage data</td>
</tr>
<tr>
<td>Data User</td>
<td>Use data (locally)</td>
</tr>
<tr>
<td>Computing User</td>
<td>Use data (remote)</td>
</tr>
<tr>
<td>System Manager</td>
<td>Manage system</td>
</tr>
</tbody>
</table>

Table 6.1: Actors and use cases of the system

In summary, the data manager wants the system to store tables (statistical data), encrypt tables, and associate some users with tables. The data user wants to retrieve tables (encrypted statistical data) make computations, and request the system for decryption of results. The computing user wants to send tables (encrypted statistical data) to the system, ask the system to perform computations on the tables and get the encrypted results. The system manager is in charge of the management of the users of the system, and of the logs of the operations performed in the system by the different users.
6.2 System architecture

The list of main use cases of Table 6.1 must be completed with some additional use cases. The complete list is as follows:

- **Manage Data.** This use case is responsible for the incorporation of new data sets in the system. It is also responsible for associating users to data sets, encrypting the data sets and creating the security parameters for data sets. Security parameters determine the extent of access permitted to each user on some data.

- **Use Data.** It is responsible for all the steps necessary to obtain some data, perform some calculation and retrieve the result.

- **Manage System.** It deals with users and with usage logs.

- **Manage Privacy Homomorphisms.** This use case is responsible for managing the privacy homomorphisms that can be used by the data owners to encrypt data. There must be mechanisms to incorporate, update, remove, and use privacy homomorphisms.

- **Manage Users.** This use case is responsible for the creation of identities for users and for introducing the requirements that users must satisfy to be eligible for accessing data sets. It is also responsible for verifying whether a user exists in the system and whether a user can access some specific data.

- **Manage Statistics.** This use case is responsible for maintaining the statistics of the use of the data sets by the users. It is used to keep track of and verify the calculations that some user performed on a data set.

- **Manage Access.** It is responsible for associating users with data sets and for determining whether the encrypted results of calculations can be decrypted based on the usage statistics of the data set by the user.

- **Perform calculi.** It is responsible for receiving an expression, invoking the corresponding privacy homomorphism and performing the computations specified by the expression.
Figure 6.2 shows the use case diagram of the system as characterized by the actors and the use cases. This is the main use case diagram of the system, and shows the users and the uses of the whole system. Starting from the use cases we have depicted the main functionalities from the point of view of the users of the system.

![Use case diagram of the Dikē system](image)

**Figure 6.2: Use case diagram of the Dikē system**

**Collaborations**

The use case describes a functionality and to describe how the functionality is implemented we use *collaborations*. Collaborations model the set of elements (with static and dynamic structure) that implement a use case. A collaboration is a collection of classes, interfaces, and other elements that work together to provide some cooperative behavior.

This subsection details the different collaborations that take place between the different objects to perform the functionalities determined by the use cases.

The collaboration associated with Manage Data implements the following scenarios:

- **Login.** The data manager actor sends a login message with a certificate as a parameter that identifies it to the system. The system receives the certificate and does the following:
1. Verify whether the certificate authority is validated by the system.
2. Verify whether the certificate is valid.
3. Verify that the certificate is not a revoked one.
4. Return a response to the actor.

- *List Tables, Put Table, Get Table, Delete Table, Encrypt Table.* These are scenarios for the maintenance of tables owned by an actor. In particular Encrypt Table create the keys and the security set associated to the table.

- *Grant Access, List Granted, Revoked Grant, List CA, List Users.* These scenarios are responsible to manage the association of users with tables.

Table 6.2 summarizes the different scenarios of the collaboration associated to the Manage Data use case. The collaboration associated with the Use Data

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage Data</td>
<td>Login</td>
</tr>
<tr>
<td>Manage Data</td>
<td>List Tables</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Put Table</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Get Table</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Delete Table</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Encrypt Table</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Grant Access</td>
</tr>
<tr>
<td>Manage Data</td>
<td>List Granted</td>
</tr>
<tr>
<td>Manage Data</td>
<td>Revoked Grant</td>
</tr>
<tr>
<td>Manage Data</td>
<td>List CA</td>
</tr>
<tr>
<td>Manage Data</td>
<td>List Users</td>
</tr>
</tbody>
</table>

Table 6.2: Scenarios associated with the collaboration of the Manage Data use case

use case implements the following scenarios:

- *Login.* The computing user or the data user actor connects to the system and sends his certificate to the system. This scenario performs the same operations as in the Manage Data use case collaboration.
A Prototype for Secure Delegation

- **List Tables, Put Table, Get Table, Delete Table.** These are scenarios for the management of tables that the actor wants the system to compute with. The system only has encrypted tables and no security parameters associated with them. So it cannot decrypt results.

- **Compute Remote.** It is performed remotely and passes as a parameter the expression that the computing user actor wants the server to compute on the encrypted table located remotely.

- **Compute Local.** It is performed locally and passes as a parameter the expression that the data user actor wants to compute on the encrypted table located locally.

- **Decrypt Remote.** The data user actor requests the server to decrypt the result.

- **Decrypt Local.** The computing user actor decrypts the result locally (it has the keys to decrypt the result).

Table 6.3 summarizes the different scenarios of the collaboration associated with the Use Data use case.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Data</td>
<td>Login</td>
</tr>
<tr>
<td>Use Data</td>
<td>List Tables</td>
</tr>
<tr>
<td>Use Data</td>
<td>Put Table</td>
</tr>
<tr>
<td>Use Data</td>
<td>Delete Table</td>
</tr>
<tr>
<td>Use Data</td>
<td>Get Table</td>
</tr>
<tr>
<td>Use Data</td>
<td>Compute Remote</td>
</tr>
<tr>
<td>Use Data</td>
<td>Compute Local</td>
</tr>
<tr>
<td>Use Data</td>
<td>Decrypt Remote</td>
</tr>
<tr>
<td>Use Data</td>
<td>Decrypt Local</td>
</tr>
</tbody>
</table>

Table 6.3: Scenarios associated with the collaboration of Use Data use case

Figure 6.3 shows the interaction diagram for the Login scenario of the Use
Data collaboration (in fact this scenario with the corresponding alterations can be used to represent the same scenario in all the collaborations.

![Interaction diagram for the Login scenario of the Use Data use case](image)

Figure 6.3: Interaction diagram for the Login scenario of the Use Data use case

Figure 6.4 shows the interaction diagram for the Compute Local scenario of the Use Data collaboration.

Figure 6.5 shows the interaction diagram for the Compute Remote scenario of the Use Data collaboration.

The collaboration associated with the Manage System use case implements the following scenarios:

- **Login.** The Manage System actor sends a Login message with a certificate as a parameter. The collaboration associated to the Manage System use case performs the same operations as in the Manage Data use case collaboration.

- **List Users, Add User, Delete User, Modify User, Validate User.** These are scenarios for the management of users.

- **Get Log, Put Log.** These are scenarios for the management of logs.

Table 6.4 summarizes the different scenarios of the collaboration associated with the Manage System use case.
Figure 6.6 is an activity diagram that describes the work-flow activity associated with the addition of a new user to the system. The user is characterized by a certification authority certificate and by a user certificate from the certification authority. The Manage System use case manages a list of acceptable certification authorities. In the event that the user does not have a certificate from an acceptable certification authority, Manage System may act as a certification authority and create a certificate for that user based on the information filled in a form and introduced to the system. The information and the requirements needed for the creation of a user certificate are determined by the policy of the system and the data owner.

### 6.2.2 Design view

From the use cases we derive the basic classes that are part of the different collaboration scenarios:

- A table is the basic class of manipulation for the system. Each table corresponds to one of the two types of statistical data previously stated in the requirements.
Figure 6.5: Interaction diagram for the Compute Remote scenario of the Use Data use case

- Users are another basic class. A user object is represented by certificates and by some characteristics that determine what it can do and what it can access.

- LogEntry is another class that corresponds to some accountable action performed by/on the system.

- Security Association is a class whose objects maintain the security parameters associated with a table.

- Privacy Homomorphism is the class that determines how to encrypt, decrypt, and compute with tables.

- Computations are expressions that can be evaluated on a table.

Classes

The main techniques used to describe system functionalities are use case diagrams and class diagrams. In the first step toward the construction of an analysis model, and based on the informal description of the system, we use two kinds of high level descriptions: the first, a use case diagram of the system, and the second the class diagrams that are identified from the use case
Figure 6.6: Activity diagram for the Add User scenario of the Manage System use case
<table>
<thead>
<tr>
<th>Use Case</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage System</td>
<td>Login</td>
</tr>
<tr>
<td>Manage System</td>
<td>List Users</td>
</tr>
<tr>
<td>Manage System</td>
<td>Add User</td>
</tr>
<tr>
<td>Manage System</td>
<td>Delete User</td>
</tr>
<tr>
<td>Manage System</td>
<td>Modify User</td>
</tr>
<tr>
<td>Manage System</td>
<td>Validate User</td>
</tr>
<tr>
<td>Manage System</td>
<td>Get Log</td>
</tr>
<tr>
<td>Manage System</td>
<td>Put Log</td>
</tr>
</tbody>
</table>

Table 6.4: Scenarios associated with the collaboration of Manage System use case

Figure 6.7: StatService class diagram

diagram and from the preliminary description and requirements of the system. In subsequent steps, the previously mentioned two types of diagrams are enriched with further details to obtain other diagrams that completely describe the system.

Figure 6.2 is the principal use case diagram and represents the interaction that three different kinds of agents can perform with the system. The data user is—as the name implicitly suggests— the agent that manipulates data. The data manager is the agent responsible for putting data in the system and determining who can access the data. As a special case, the data manager can
Figure 6.8: Interaction diagram for the client-server-administrator scenario

perform encrypted computation on the server. The system manager agent is responsible for the users and for the logs that show how and by whom the data is used.

Figure 6.8 represents the scenario of interaction between the three top-level components of Đikē. Initially, a client, the administrator and the server contain, respectively, an object of class StatClient, an object of class StatAdmin and an object of class StatService.

StatService class

The structure of StatService is depicted in Figure 6.7 using the notation described in [CD94]. Boxes represent object types. A line between two boxes represents a relationship between two object types. A line ending with a ◦ means that the object type at the beginning of the line is part of the object type at the end of the line. When a line represents a one-to-many relationship, a • indicates the “many” end of the line.

Initially, StatService consists of a UserService, a SecurityService, a TableService and a LogService. Each time a Client logs into StatService, an instance of StatServiceAccess is created. When the Administrator logs into StatService, an instance of StatServiceAdmin is created; note that there may be only one instance of StatServiceAdmin at the same time, i.e. only one Administrator may be manipulating the system.

UserService is responsible for the management of users of Dikē. Its main functions are:

- Control that only one client per user identity can access the StatService at the same time.

- Implement the user identification and table access authorization by maintaining an access matrix that specifies which data tables can be accessed by each user.

SecurityService manages the security sets, i.e. the associations between users and tables. Such management involves creating and deleting security sets as well as extracting a key or another parameter from a security set.

TableService is responsible for the management of tables in the server. It contains tables and its interface consists of operations to add and remove tables, add and remove columns from a table, retrieve a table.

LogService logs all requests to decrypt encrypted results obtained from encrypted data. Each log entry contains a reference to the user requesting the decryption, the encrypted result, a reference to the encrypted table used as
input data to the user computation, and the expression describing the computation performed by the user on the encrypted table. To implement computation verifiability, a function exists which returns all log entries corresponding to a given user-table pair; parity checking is used on each log entry to detect potential user fraud (see section 4). The parity checking function can be invoked each time the user asks for decryption of a result or it can be invoked after a certain number of decryptions have been logged.

**StatClient class**

The structure of StatClient is depicted in Figure 6.9. StatClient consists of a CalcService, a TableService and a reference to a StatServiceAccess.

CalcService is used to perform operations on an encrypted table. It maintains a stack on which operations are performed. Operands are pushed onto the stack; arithmetical and logical operations (add, multiply, compare) take as operands the top two elements popped from the stack; the result is pushed onto the stack. Note that operations are carried out on encrypted data; thanks to polymorphism, an operation such as multiply is performed in a way which depends on the PH used for encryption.

TableService is analogous to TableService in StatService, but it only contains tables associated to the current user.
StatAdmin class

StatAdmin is in charge of the administration of the users, the tables and the security sets. When StatAdmin invokes the Admin method in StatService, then StatService creates an instance of StatServiceAdmin. It is through the interface of StatServiceAdmin that StatAdmin does its job.

6.3 Implementation notes

Dike uses the CORBA [OMG96] [Sie96] distributed computing standard to describe and implement its main components. All the main components of the system are described using the OMG IDL. The implementation consists of two levels:

- The first level is the library for handling encrypted data, i.e. the library implementing the privacy homomorphisms RSA and JR. This library is written in C++ and is currently based on the arithmetical library from GNU MP [Gra96]. The first version was based on the LiDIA [Gro98] arithmetical library. Each PH is implemented as a method for encryption, a method for decryption and a method for each operation that can be performed on encrypted data using this PH.

- The second level is the visible part of CORBA, i.e. the interface of components. CORBA IDL (Interface Definition Language) is used to describe the interfaces of the objects that compose the different parts of the system.

When operating on encrypted data, the way a given operation is implemented depends on the particular privacy homomorphism used for encryption. Polymorphism provided by object-oriented technology is very useful here, because it allows coding handler applications that are independent of the privacy homomorphisms used by the data owner. The code for an operation may depend on the data type in a transparent way. For instance, “multiply \( E_k(a) \) and \( E_k(b) \)” will call a different code if \( E_k() \) is RSA encryption or encryption...
using the JR homomorphism. (the code to be used is provided by the data owner, who chooses either privacy homomorphism).
Chapter 7

Conclusions and Future Work

Section 7.1 summarizes the main results obtained. Section 7.2 explicitly exposes future lines of research, computing delegation.

7.1 Main Results

Due to its very nature, the solution to the delegation problem must be a distributed one. We have proposed cryptographic techniques to provide secure delegation, i.e. to enable an untrusted handler to deal with sensitive data without compromising statistical confidentiality. Also, the structure of a prototype based on the CORBA technology has been described. More specifically:

- We have designed and implemented a new privacy homomorphism, which is the first one to allow complete arithmetic on encrypted data while being unconditionally secure against known-cleartext attacks.

- We have developed a method for quick verification of computations based on parity checking.

- We have used the privacy homomorphism and the parity checking method as building blocks of a system for secure delegation. In this system, a data owner encrypts confidential statistical data under the privacy homomorphism and sends them to an untrusted data handler, who does
some computations on the encrypted data and returns the encrypted result to the data owner; using the parity checking mechanism, the data owner can quickly verify the correctness of handler computation.

- In the process of implementing a prototype for the delegation system, we have experimented with the CORBA approach to implement secure distributed applications. Also, UML notation has been used to describe the modeling and implementation stages of the prototype.

7.2 Future Research

In the practical domain, we are interested in equipping our delegation system with security mechanisms that reduce the complexity of managing security. Such mechanisms should maintain the security of data in a decentralized manner.

In the theoretical domain, an interesting topic is to work out a precise set of rules to help the data owner to decide when decryption of a result should be denied to the data handler. Such a set of rules should depend on:

- The security limitations inherent to the privacy homomorphisms used. For example, it may be dangerous to release too many ciphertext-cleartext pairs.

- The lowest probability of handler fraud detection that can be tolerated by the owner. Some expressions claimed by the handler may be nearly always even (see section 4) and should probably be rejected.

- For data delegation, inspection of logs (see below).

Whereas ensuring secure computation as all that is required by computing delegation, in data delegation the data handler could pool the decrypted results of several “innocent” expressions to infer one or more sensitive input data. This is known as the statistical database problem, which was shown to be very hard long ago ([Sch81]) and is neither solved by our approach nor completely
solved by any known approach. So another line of future research can focus on mitigating the statistical database problem. Two possible complementary ideas are:

- Creating a set of tools to analyze the logs of the operations on encrypted data. This analysis would take place off-line and could help detect “unfaithful” handlers, who could be subsequently denied authorization to access sensitive data.

- Using an output-perturbation method on decrypted results. Before returning a decrypted result to the data handler, the data owner perturbs it randomly (and slightly). In this way, inferences by the data handler from decrypted results will not be exact. Note that output-perturbed data delegation still provides higher quality results to the data handler than he could obtain if he were supplied clear perturbed input data (in data delegation, the handler can compute on exact, though encrypted, data).
Appendix A

The Unified Modeling Language (UML)

The UML specifies a modeling language that incorporates the object oriented consensus on core modeling concepts from the modeling languages found in the Booch[Boo93], OOSE/Jacobson[Jac94], OMT[RPB+91] and other methods. It allows deviations to be expressed in terms of its extension mechanisms. It provides an expressive visual modeling language to develop and exchange models. It is supported by a formal foundation. It is designed to be capable of addressing high level development concepts.

Section A.1 how to model using UML. Section A.2 to Section A.6 goes into a short description of the different UML diagrams.

A.1 Modeling using UML

UML does not represent a method of software development. Instead, it is used as an object oriented modeling language. Any method can use it to model and document the analysis and design processes of a system. Having captured the requirements, it is possible to begin modeling with UML as follows:

1. Starting with the requirements, model the different scenarios in the system or business with Use Case diagrams.

109
2. From the Use Cases, migrate to Sequence and Collaboration diagrams.

3. From the Sequence and Collaboration diagrams, derive the Class diagrams.

4. Having the Class diagrams, model the class behavior with State diagrams.

5. To model the structure of the software, use the Component diagram.

6. Finally, create a Deployment diagram to model the configuration of the system in a computational setting.

A.2 The UML notation

The notation is designed to represent different aspects of a model in terms of graphical views of the model. UML defines the following graphical diagrams:

- Use case diagram

- Structure diagrams
  - Class diagram
  - Object diagram

- Behavior diagrams
  - Statechart diagram
  - Activity diagram
  - Interaction diagrams

- Implementation diagrams
  - Component diagram
  - Deployment diagram

The following sections shortly explain the various diagrams offered by UML.
A.3 Use case diagrams

Use case diagrams show elements from the use case model and represent the functionality of a system or class as manifested to external interactors with the system. The use case is a unit of functionality provided by a system or class as manifested by sequences of messages exchanged among the system and one or more outside interactors (actors) together with actions performed by the system. The diagram consists of ellipses, each of which contains the name of a use case.

A.4 Structure diagrams

They describe the situation in the world, by typifying objects, their composition, relations, associations and roles. Structure diagrams can also describe object properties. They do not show temporal relations. There are two types of structure diagrams:

1. Class diagrams that show relations between classes of objects

2. Object diagrams that show relations between instances of different objects

A.4.1 Class diagrams

The class diagram shows the static structure of the model. In particular, it reflects aspects such as classes and types, their internal structure and their relationship to other aspects. Class diagrams do not show temporal information, although they may contain reified occurrences of things that have or things that describe temporal behavior. A class diagram is a graph of classifier elements connected by their various static relationships.
A.4.2 Object diagrams

Object diagrams represent graphs of object instances that may appear during runtime in a system. They are used in early stages of development and are modified by refinement steps and augmented with operations and attributes. Eventually, object diagrams can be translated into class skeletons.

A.5 Behavior diagrams

This kind of diagram provides a means to describe dynamic behavior. The main idea is to describe situations that change as a consequence of events. For example, we can consider a single object and its changes as a consequence of the events; the changes that an operation causes in a situation; or the changes that some object causes in each object that participates in the situation.

A.5.1 Statechart diagrams

The statechart diagram shows the sequences of states that an object or an interaction goes through during its lifetime in response to received stimuli, together with its responses and actions.

A.5.2 Activity diagrams

An activity diagram is a variation of a state machine, in which states represent the realization of the operations and transitions are triggered by the completion of the operations. An activity diagram represents a procedure, which is the implementation of an operation on the owning class.

A.5.3 Interaction diagrams

They represent a pattern of interaction among objects. Interaction diagrams can take two forms:

1. Sequence diagrams that emphasize the sequence of messages
2. Collaboration diagrams that emphasize the relationships among the objects

**Sequence diagrams**

A sequence diagram represents a sequence of messages exchanged among objects to perform some operation or obtain some result. This kind of representation has two dimensions, one for objects and the other for time. The time dimension permits to specify message precedence.

**Collaboration diagrams**

A collaboration diagram represents a collaboration among a set of objects in a determined context to perform some operation or obtain some result. A collaboration diagram shows the relationships among the object roles and the sequence of messages. Concurrency has to be expressed using sequence numbers.

**A.6 Implementation diagrams**

Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms: component diagrams that show the structure of the code itself and deployment diagrams that show the structure of the run-time system.

**A.6.1 Component diagrams**

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. A software module may be represented as a component type. Some components exist at compile time, some exists at link time, and some exist at run time; some exist more than one time. A compile-only component
is one that is only meaningful at compile time; the run-time component is this case would be an executable program.

A.6.2 Deployment diagrams

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams; they should be shown on component diagrams.
Appendix B

The CORBA Architecture

CORBA (Common Object Request Broker Architecture) is the standard for a distributed object architecture developed by the Object Management Group (OMG) consortium and specified in [OMG96]. The main objective of this consortium is to create an architecture for an open software bus, or Object Request Broker (ORB), on which object components written by different vendors can inter-operate across networks and operating systems. The CORBA objects can invoke other objects or be invoked by them, without knowing where the objects they access reside on or in what language the requested object is implemented or in what execution environment the called object is executing. The Interface Definition Language (IDL) is used to define the interfaces of CORBA Objects. Section B.1 summarizes the main characteristics of the CORBA model. Section B.2 explicitly details the characteristics of the IDL language and its use in describing objects. Section B.3 describes the main steps needed to produce software using CORBA.

B.1 Architecture

The concepts introduced in this section are more fully defined in [OMG96]. The client for a CORBA object has an object reference for the object and the client uses this object reference to issue method requests. If the server
object is remote, the reference points to a stub function, which uses the ORB functionality to identify the machine that runs the server object and asks that machine’s ORB for a connection to the object’s server. When the stub code has the connection, it sends the object reference and parameters to the skeleton code linked to the destination object’s implementation. The skeleton code transforms the call and parameters into the required implementation-specific format and calls the method in the object. Any exceptions or results are returned through the same path.

Figure B.1 contains a scheme for the main components of the CORBA architecture and the most important relations between these components. The communication between two different ORBs are performed via the OMG Internet Inter ORB Protocol (IIOP). The client has no knowledge of the CORBA object location, implementation detail, or what ORB it uses to access the server object.

A CORBA object exists only if there is an interface defined in IDL for this object. The stubs and skeleton code can be automatically generated by a compiler from the IDL interface definition. The stub code makes it possible to the client programming language to access CORBA objects. The server skeleton code is used by the ORB to dispatch methods to the object implementation instance (servant).

B.2 The IDL specification language

The IDL is the language used to describe the functionality that client objects can call and that object implementations provide. The IDL language is not a programming language and its only purpose is to define complex data types that in this context are called interfaces. The IDL supports multiple inheritance, exceptions and the declaration of contexts for the interfaces. The IDL interface of a CORBA object completely characterizes the external view of an object. An IDL compiler transparently transforms OMG IDL definitions into an application programming language like C++ or Java. The use of
IDL compilers eliminates a source of errors and allows the implementation of automated optimizations by the compiler. The compiler implements the language mappings defined by the CORBA specification for each language that it supports.

### B.2.1 Object description

An object is described by an interface, so an object identity is determined by the corresponding interface. An interface is composed of signatures of operations.

### B.2.2 Signatures

A signature characterizes the operations that can be performed by the object, the type of parameters accept by operations, the responses returned and the exceptional circumstances that may occur. The responses, return values and parameters can be of a basic type or another object interface. Each parameter of an operation can be declared as: input-only, output-only or input/output.
B.2.3 Types

The basic types are the construction blocks for the more complex types and are mapped almost directly to the target language. These types, together with some rules of composition and interfaces, are the objects that can be used as parameters or responses to operation requests.

B.3 Using CORBA

For simplicity, developing under CORBA can be thought of as two separate activities: implementing the CORBA object and implementing the application that uses the implementation of CORBA objects. In this way, the tools, the language implementation, the location and the ORB implementation can be selected independently from one another.

The first step in each case is to define the interface of the objects that are necessary for the applications. In the case where the implementation of the objects initially exists, obtain the interface for that object — for example from an interface repository.
Normally, the interfaces are defined in OMG IDL (Interface Definition Language). Through a translator, the IDL definition is transformed to the language in which the object is going to be implemented, or in the case of the use of some CORBA object by the application, to the same implementation language of the application.
Appendix C

Basic Cryptologic Definitions

In this Appendix, we give some basic background on cryptology, based on [GB97] and [Gru97]. Cryptography is about everything that has to do with someone having to communicate securely in the presence of an adversary. Cryptanalysis is about everything that the adversary has to do to break such setting. The combination of these two conflicting objectives is the domain of cryptology. Here we are interested in exposing some of the building blocks for constructing cryptosystems and cryptographic protocols.

C.1 Cryptosystems

The basic scenario for cryptography is when a party \( \mathcal{A} \) wants to send to party \( \mathcal{B} \) a secret message \( m \) over a communication channel. The communication channel may be compromised because of it been tapped by an adversary of both parties. We can consider two solutions to these problem: shared-key encryption and public-key encryption.

C.1.1 Shared-key encryption

In this setting, the two parties meet before a private message transmission takes place and agree on two algorithms \( \mathcal{E} \) and \( \mathcal{D} \), and some piece of information \( S \) to be kept secret by both parties. The private message \( m \) is referred to as the
cleartext or cleartext. S is referenced as shared secret key. If A wants to send the message \( m \) to B, the first step is to encrypt \( m \) using the function \( \mathcal{E} \) and the secret key \( S \) to obtain a ciphertext \( c ; c = \mathcal{E}(S, m) \). As a second step, A sends to B the encrypted message \( c \). In the last step, B decrypts the message \( c \) using the function \( \mathcal{D} \) and the secret key \( S \) to recover the original value \( m \); \( m = \mathcal{D}(S, c) \).

### C.1.2 Public-key encryption

In this setting, the requirement that both parties share a secret key \( S \) is dropped; instead, a key pair is used. A public key \( P \) that is published in a way that it can be associated with the party B, and also a secret key \( S \) that only party B knows. The public key is used by anyone that wants to send a secret message to B (e.g. party A). Party A uses the public key \( P \) and the encryption algorithm \( \mathcal{E} \) to encrypt the message \( m \) and sends the resulting ciphertext \( c \) to the party B, i.e. \( c = \mathcal{E}(P, m) \). The party B has a private key \( S \) associated to the public key \( P \) and uses it with the algorithm \( \mathcal{D} \) to decrypt the ciphertext \( c \) sent by party A and recover the original message \( m \), i.e. \( m = \mathcal{D}(S, c) \).

### C.2 Complexity

The first theoretical approach to cryptography was based on information theory. The main assumption of the information-theoretic approach is the belief that the adversary (cryptanalyst) has not enough information to decrypt messages. The current approach to cryptography is based on complexity theory. The assumption of the complexity-theoretic approach is that the adversary has not enough time or space to decrypt messages.

In the previous definitions of cryptosystems, the computational power of the different parties and especially that of the adversary are of great importance to quantify the security of the system. For practical purposes, we can
C.2 Complexity

assume that the encryption, decryption, and adversary algorithms are probabilistic polynomial time algorithms (see Subsection C.2.1). The running time of the different algorithms are considered as a function of a security parameter \( k \). When we say that some algorithm runs in polynomial time, we mean that it is time-bounded by some polynomial function in \( k \).

Taking in consideration the computational complexity of the adversary we have to move from the information-theoretic concept of impossibility of breaking an encryption scheme (cryptosystem) by finding information about the exchanged messages to the complexity-theoretic concept of infeasibility of breaking it. So an encryption scheme is considered secure if it is infeasible to break the scheme by the a computationally bounded adversary.

C.2.1 Complexity classes and standard definitions

In the Turing Machine computing model, a problem can be solved by a machine \( M \) if and only if the input \( x \) of the problem belongs to the language \( L(M) \) accepted by the machine \( M \). Complexity classes are classes of problems or equivalently classes of languages. We next define a few of them which are relevant in cryptography.

**Definition 2 (Complexity class P)** A language \( L \) is in \( P \) if and only if there exists a Turing machine \( M(x) \) and a polynomial function \( Q(y) \) such that on input string \( x \)

1. \( x \in L \) if and only if \( M \) accepts \( x \) (denoted by \( M(x) \)).

2. \( M \) terminates after at most \( Q(|x|) \) steps.

The class of languages \( P \) is classically considered to be those languages which are “easily computable”. We use this term to refer to these languages and the term “efficient algorithm” to a polynomial-time Turing machine.

**Definition 3 (Complexity class NP)** A language \( L \) is in \( NP \) if and only if there exists a Turing machine \( M(x, y) \) and polynomials \( p \) and \( l \) such that on input string \( x \)
1. $x \in L \Rightarrow \exists y$ with $|y| \leq l(|x|)$ such that $M(x, y)$ accepts and $M$ terminates after at most $p(|x|)$ steps.

2. $x \notin L \Rightarrow \forall y$ with $|y| \leq l(|x|)$, $M(x, y)$ rejects.

An equivalent definition is that $L \in \text{NP}$ if there exists a non-deterministic polynomial time Turing machine $M$ which accepts $x$ if and only if $x \in L$; the string $y$ above corresponds to the guess of the non-deterministic Turing machine.

**Definition 4 (Complexity class BPP)** A language $L$ in $\text{BPP}$ (Bounded-error Probabilistic Polynomial time) if and only if there exists a Turing machine $M(x, y)$ and polynomials $p$ and $l$ such that on input string $x$:

1. $x \in L \Rightarrow Pr_{|y|<l(|x|)}[M(x, y) \text{ accepts}] \geq 1/2 + 1/p(|x|)$

2. $x \notin L \Rightarrow Pr_{|y|<l(|x|)}[M(x, y) \text{ accepts}] \leq 1/2 - 1/p(|x|)$

3. $M(x, y)$ always terminates after at most $p(|x|)$ steps.

Let a probabilistic polynomial time (PPT) Turing machine $M$ be a Turing machine which can flip coins as an additional primitive step, and on input string $x$ runs for at most a polynomial in $|x|$ steps. An equivalent definition of the complexity class $\text{BPP}$ is that $L \in \text{BPP}$ if there exists a probabilistic polynomial time Turing machine $M$ which accepts $x$ with probability greater than $1/2$ if and only if $x \in L$; the string $y$ corresponds to the sequence of coin flips made by the machine $M$ on input $x$.

The class $\text{BPP}$ will be considered as the class of “efficiently computable” languages or problems.

It is known that $\text{P} \subseteq \text{NP}$ and $\text{P} \subseteq \text{BPP}$. It is not known whether these containments are strict although it is often conjectured to be the case. It is not known whether $\text{BPP}$ is a subset of $\text{NP}$. 
C.3 Cryptography definitions

C.3.1 One-way functions

Informally, they are functions which are easy to compute but hard to invert. The term *easy computation* in this context can be defined as one which can be performed by a PPT algorithm (Turing machine). The term *hard to invert* can be defined more precisely as: any probabilistic polynomial time (PPT) algorithm attempting to invert a one-way function in its range, has a *negligible probability* of success. This probability is taken over the elements of the domain of the one-way function, and the coin tosses of the PPT algorithm which attempts to do the inversion. The term *negligible* is associated with a function that vanishes faster than the inverse of any polynomial.

**Definition 5** a function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is negligible if $\forall c : c > 0$ there $\exists k_c : k_c \in \mathbb{N}$ and $\nu(k) < k^{-c}$

**Definition 6** A function $f : \{0, 1\}^* \mapsto \{0, 1\}^*$ is one-way if:

1. There exists a PPT algorithm that on input $x$ outputs $f(x)$;

2. For every PPT algorithm $A$ and for a sufficient large $k$, there is a negligible function $\nu_A$ such that,

$$Pr[f(z) = y : x \xleftarrow{R} \{0, 1\}^k ; y \leftarrow f(x) ; z \leftarrow A(1^k, y)] \leq \nu_A(k)$$

C.3.2 Trapdoor one-way functions

A *trapdoor one-way function* is a one-way function with a secret inverse function (trapdoor) that inverts any value of the domain of the trapdoor function efficiently. The one-way function has to be easy to compute on any input, but infeasible to invert without the knowledge of the trapdoor function associated with it. The pair of functions (one-way and trapdoor) has to be easy to find. The disclosure of the one-way function should not reveal anything about how to compute the inverse on any value on the range.
Definition 7 A one-way function \( f : \{0,1\}^* \mapsto \{0,1\}^* \) is a trapdoor one-way function if there exists a polynomial \( p \) and a PPT algorithm \( I \) that \( \forall k \), there \( \exists t_k \in \{0,1\}^* \) such that \( |t_k| \leq p(k) \), and \( \forall x : x \in \{0,1\}^* \), \( I(f(x), t_k) = y \) such that \( f(y) = f(x) \).

A trapdoor one-way function is the primitive used to build a public-key cryptosystem. The one-way function (without the trapdoor) can be revealed as the public key; the secret inverse function (trapdoor) is the private key.

C.3.3 Examples of one-way functions

Definition 8 (RSA trapdoor function [RSA78]) Let \( N = pq \) be product of two distinct primes \( p \) and \( q \) of roughly equal length. Let \( k = |N| \); this is about 1024, since it is generally believed that numbers of this length, or greater, are hard to factor. \( \phi(N) = |\mathbb{Z}_N^*| \) is the Euler totient function and \( \phi(N) = (p-1)(q-1) \). Now let \( e \) be such that \( (e, \phi(N)) = 1 \), that is, \( e \in \mathbb{Z}_{\phi(N)}^* \). The RSA function is defined by

\[
f : \mathbb{Z}_N^* \to \mathbb{Z}_N^* \\
x \mapsto x^e \mod N
\]

As \( \mathbb{Z}_{\phi(N)}^* \) is a group, \( e \) has an inverse \( d \in \mathbb{Z}_{\phi(N)}^* \). So \( e \) and \( d \) satisfy \( ed \equiv 1 \mod \phi(N) \). To obtain \( x \) again:

\[
(x^e)^d \mod N = x^{ed \mod \phi(N)} \mod N = x^1 \mod N = x
\]

RSA is a privacy homomorphism. Let \( \ast \) denote the modular multiplication over \( \mathbb{Z}_N \), and define the predicate \( = \) as a test for equality. It holds that

\[
a' \ast b' = (a^e \mod m)(b^e \mod m) \mod m = (a \ast b)^e \mod m
\]

\[
a' = b' \text{ if and only if } a = b
\]
This homomorphism allows only one operation, but appears to be very secure. Finding \( d \) from \( e \) seems to be equivalent to factoring a large modulus \( N \)—no polynomial-time algorithm for factoring has been published up-to-date—.

**Definition 9 (Blum Integers).** A number \( N \) is called a Blum integer if it satisfies \( N = pq \) where \( p \equiv 3 \mod 4 \) and \( q \equiv 3 \mod 4 \). Usually, \( p \) and \( q \) are specified to have roughly the same size.

**Definition 10 (Discrete Logarithm Problem).** Finding a discrete logarithm consists of determining, given integers \( a, n, x \), an integer \( m \) such that \( a^m \mod n = x \), if such \( m \) exists. It may happen that there are two such \( m \), for example, \( m = 10 \) and \( m = 4 \) for the equation \( 5^m \mod 21 = 16 \), or none, for example, for the equation \( 5^m \mod 21 = 3 \). An important case is when \( g \) is a generator or a principal root of \( \mathbb{Z}_n^* \), that is if \( \mathbb{Z}_n^* = \{ g^i \mod n | 0 \leq i \leq \phi(n) \} \). In such case, for any \( x \in \mathbb{Z}_n^* \), there is a unique \( m < \phi(n) \) such that \( g^m \mod n = x \). Such \( m \) is called the discrete logarithm or index of \( x \) with respect to \( n \) and \( g \). No efficient deterministic algorithm is known that can compute, given \( a, n \) and \( x \), the discrete logarithm \( m \) such that \( a^m \mod n = x \). This fact plays an important role in cryptography and its applications, and also for (perfect) random number generators. An exception is the case in which \( p \) is a prime and factors of \( p - 1 \) are small, of the order \( O(\log(p)) \); in this case, computing discrete logarithms is feasible.

**Definition 11 (Quadratic Residuosity)** An element \( x \in \mathbb{Z}_N^* \) is a square, or quadratic residue, if it has a square root, this is, there is a \( y \in \mathbb{Z}_N^* \) such that \( y^2 \equiv x \mod N \). If not, it is a non-square or non-quadratic-residue. A number may have lots of square roots.

**Definition 12 (Primitive Root Function)** A primitive root of a number \( p \) is an integer \( g \) satisfying \( 1 \leq g \leq p - 1 \) such that the residue classes of \( g, g^2, g^3, \ldots, g^{p-1} = 1 \) are all distinct. If \( p \) is a prime number, then there are exactly \( \phi(p - 1) \) incongruent primitive roots of \( p \).
C.4 Cryptanalysis

Cryptanalysis is the science and study of methods for breaking ciphers. A cipher is breakable if it is possible to determine the key from the cleartext or the ciphertext, or to determine the cleartext from the ciphertext.

C.4.1 Attacks

There are five basic methods for a cryptanalyst to attack a cryptosystem:

- Ciphertext-only attack
- Known-cleartext attack
- Chosen-cleartext attack
- Chosen-ciphertext attack
- Known-encryption-algorithm attack

**Definition 13 (Ciphertext-only attack)** The cryptanalyst gets ciphertexts $c_1 = \mathcal{E}(S, m_1), ..., c_n = \mathcal{E}(S, m_n)$ and tries to infer the key $S$ or as many $w_1, ..., w_n$ as possible.

**Definition 14 (Known-cleartext attacks)** The cryptanalyst knows some pairs $(m_i, \mathcal{E}(S, m_i)), 1 \leq i \leq n$ and tries to infer $S$, or at least to determine $m_{n+1}$ for a new ciphertext $\mathcal{E}(S, m_{n+1})$.

**Definition 15 (Chosen-cleartext attacks)** The cryptanalyst chooses cleartexts $m_1, ..., m_n$, obtain ciphertexts $\mathcal{E}(S, m_1), ..., \mathcal{E}(S, m_n)$ and tries to infer $S$, or at least to determine $m_{n+1}$ for a new ciphertext $\mathcal{E}(S, m_{n+1})$.

**Definition 16 (Chosen-ciphertext attack)** The cryptanalyst knows some pairs $(c_i, \mathcal{D}(S_{c_i})), 1 \leq i \leq n$, where the ciphertexts $c_i$ have been chosen by the cryptanalyst. The goal is to determine the key $S$.

**Definition 17 (Known-encryption-algorithm attack)** The encryption algorithm $\mathcal{E}$ is given and the cryptanalyst tries to obtain the decryption algorithm $\mathcal{D}$ before actually receiving any samples of the ciphertext.
C.4.2 Cryptanalyst behavior

In order to collect data for mounting the attacks described in the previous subsection, the cryptanalyst must interact with the medium where the encrypted information is sent. Such interaction is called wiretapping and can be passive or active.

Definition 18 (Eavesdropping). Eavesdropping or passive wiretapping refers to the interception of messages usually without alteration of these.

Definition 19 (Tampering). Tampering or active wiretapping refers to the deliberate modifications made to the message stream. These can be for the purpose of making arbitrary changes to a message, to replace data in a message with replays of data from earlier messages or to insert a fake message (message forgery).

C.5 Number theory

Definition 20 (Legendre symbol) If \( p \) is a prime greater than 2 and \( 0 < a < p \), the Legendre symbol \( (a/p) \) is a characteristic function of the set of quadratic residues modulo \( p \):

1. \( (a/p) = 1 \) if \( a \) is a quadratic residue modulo \( p \).
2. \( (a/p) = -1 \) if \( a \) is a quadratic nonresidue modulo \( p \).

Definition 21 (Jacobi symbol) If \( k > 1 \) is odd and \( h \) is in \( \mathbb{Z}_k^* \), the Jacobi symbol \( (h/k) \) can be defined as follows:

1. The Jacobi symbol \( (h/p) \) coincides with the Legendre symbol if \( p \) is prime.
2. If \( k = \prod_{i=1}^{m} p_i \) with \( p_i \) primes, then \( (h/k) = \prod_{i=1}^{m} (h/p_i) \).

An efficient algorithm for computing the Jacobi symbol \( (a/n) \) in \( O(\log n) \) steps is the recursion:
1. 

\((1/k) = 1\)

2. 

\((a \cdot b/k) = (a/k) \cdot (b/k)\)

3. 

\((2/k) = 1\) if \((k^2 - 1)/8\) is even, \(-1\) otherwise.

4. 

\((b/a) = ((b \mod a)/a)\)

5. If \(\gcd(a, b) = 1\) then

   (a) \((a/b)(b/a) = 1\) if \((a - 1)(b - 1)/4\) is even

   (b) \((a/b)(b/a) = -1\) if \((a - 1)(b - 1)/4\) is odd
Bibliography


