UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S.I. INFORMÁTICOS

MASTER IN SOFTWARE AND SYSTEMS

MASTER THESIS

A FRAMEWORK FOR IMPLEMENTING OUTSOURCING SCHEMES

AUTHOR: ROSARIO SEBASTIANO RUSSO

SUPERVISOR: MANUEL CARRO LIÑARES
CO-SUPERVISOR: DARIO FIORE

MADRID, de 15 de Julio 2015

Resumen

En esta tesis se aborda el problema de la externalización segura de servicios de datos y computación. El escenario de interés es aquel en el que el usuario posee datos y quiere subcontratar un servidor en la nube ("Cloud"). Además, el usuario puede querer también delegar el cálculo de un subconjunto de sus datos al servidor. Se presentan dos aspectos de seguridad relacionados con este escenario, en concreto, la integridad y la privacidad y se analizan las posibles soluciones a dichas cuestiones, aprovechando herramientas criptográficas avanzadas, como el Autentificador de Mensajes Homomórfico ("Homomorphic Message Authenticators") y el Cifrado Totalmente Homomórfico ("Fully Homomorphic Encryption").

La contribución de este trabajo es tanto teórica como práctica. Desde el punto de vista de la contribución teórica, se define un nuevo esquema de externalización (en lo siguiente, denominado con su término inglés Outsourcing), usando como punto de partida los artículos de [3] y [12], con el objetivo de realizar un modelo muy genérico y flexible que podría emplearse para representar varios esquemas de "outsourcing" seguro. Dicho modelo puede utilizarse para representar esquemas de "outsourcing" seguro proporcionando únicamente integridad, únicamente privacidad o, curiosamente, integridad con privacidad. Utilizando este nuevo modelo también se redefine un esquema altamente eficiente, construido en [12] y que se ha denominado $\mathsf{Outsourcing}_{lin}$. Este esquema permite calcular polinomios multivariados de grado 1 sobre el anillo $\mathbb{Z}_{2^k}$. Desde el punto de vista de la contribución práctica, se ha construido una infraestructura marco ("Framework") para aplicar el esquema de "outsourcing". Seguidamente, se ha testado dicho "Framework" con varias implementaciones, en concreto la implementación del criptosistema Joye-Libert ([18]) y la implementaciún del esquema propio $\mathsf{Outsourcing}_{lin}$.

En el contexto de este trabajo práctico, la tesis también ha dado lugar a algunas contribuciones innovadoras:

- el diseño y la implementación de un nuevo algoritmo de descifrado para el esquema de cifrado Joye-Libert, en colaboraciún con Darío Fiore. Presenta un mejor comportamiento frente a los algoritmos propuestos por los autores de [18];

- la implementación de la función eficiente pseudo-aleatoria de forma amortizada cerrada ("amortized-closed-form efficient pseudorandom function") de [12]. Esta función no se había implementado con anterioridad y no supone un problema trivial, por lo que este trabajo puede llegar a ser útil en otros contextos.

Finalmente se han usado las implementaciones durante varias pruebas para medir tiempos de ejecución de los principales algoritmos.

Abstract

In this thesis we tackle the problem of *secure* outsourcing of data and computation. The scenario we are interested in is that in which a user owns some data and wants to "outsource" it to a Cloud server. Furthermore, the user may want also to delegate the computation over a subset of its data to the server. We present the security issues related to this scenario, namely *integrity* and *privacy* and we analyse some possible solutions to these two issues, exploiting advanced cryptographic tools, such as Homomorphic Message Authenticators and Fully Homomorphic Encryption.

Our contribution is both theoretical and practical. Considering our theoretical contribution, using as starting points the articles of [3] and [12], we introduce a new cryptographic primitive, called Outsourcing with the aim of realizing a very generic and flexible model that might be employed to represent several *secure* outsourcing schemes. Such model can be used to represent secure outsourcing schemes that provide only integrity, only privacy or, interestingly, integrity with privacy. Using our new model we also re-define an highly efficient scheme constructed in [12], that we called $\mathsf{Outsourcing}_{lin}$ and that is a scheme for computing multi-variate polynomials of degree 1 over the ring $\mathbb{Z}_{2^k}$. Considering our practical contribution, we build a Framework to implement the Outsourcing scheme. Then, we test such Framework to realize several implementations, specifically the implementation of the Joye-Libert cryptosystem ([18]) and the implementation of our $\mathsf{Outsourcing}_{lin}$ scheme.

In the context of this practical work, the thesis also led to some novel contributions:

- the design and the implementation, in collaboration with Dario Fiore, of a new decryption algorithm for the Joye-Libert encryption scheme, that performs better than the algorithms proposed by the authors in [18];
- the implementation of the amortized-closed-form efficient pseudorandom function of [12]. There was no prior implementation of this function and it represented a non trivial work, which can become useful in other contexts.

Finally we test the implementations to execute several experiments for measuring the timing performances of the main algorithms.

Contents

1 Introduction 1

2 Background 5
2.1 Preliminary definitions and notation 5
2.2 The problem of secure outsourcing of data and computation 5
2.3 An approach to solve *integrity* 7
2.3.1 Homomorphic message authenticators with efficient verification 7
2.4 An approach to solve *privacy* 16
2.4.1 Fully homomorphic encryption 16
2.5 An approach to solve integrity *and* privacy 18

3 A unifying model for secure outsourcing 20
3.1 The Outsourcing scheme 20
3.1.1 Outsourcing schemes with efficient decoding 24
3.2 A generic construction of the Outsourcing scheme 25
3.3 An efficient realization of the Outsourcing scheme: verifiable computation of linear functions on encrypted data 27
3.3.1 Some facts of number theory 27
3.3.2 The Joye-Libert cryptosystem 31
3.3.3 The $\mathsf{Outsourcing}_{lin}$ scheme for computing linear functions over the ring $\mathbb{Z}_{2^k}$ 39

4 A *Framework* for implementing Outsourcing schemes 45
4.1 Framework architecture design 45
4.1.1 Support tools 46
4.1.2 High level description 47

5 Implementations using our Framework 58
5.1 Implementation of the Joye-Libert homomorphic encryption scheme . 58
5.1.1 Key generation 59
5.1.2 Encoding 66
5.1.3 Computation 68
5.1.4 Decoding 69
5.2 Implementation of the $\mathsf{Outsourcing}_{lin}$ scheme 70
5.2.1 Implementation of the amortized closed-form efficient pseudo-random function 71
5.2.2 Implementation of the scheme's algorithms 91

6 Experiments 101
6.1 Experiments about the implementation of the Joye-Libert cryptosystem 102
6.1.1 Key generation 102
6.1.2 Encoding (encryption) 102
6.1.3 Computation 103
6.1.4 Decoding (decryption) 104
6.2 Experiments about the implementation of the $\mathsf{Outsourcing}_{lin}$ scheme . 105
6.2.1 Key generation 106
6.2.2 Encoding 107
6.2.3 Computation 107
6.2.4 Decoding 109

7 Conclusions 114

Bibliography 118

Chapter 1

Introduction

Cloud computing is a novel computing paradigm that allows users to outsource resources such as storage and computing power to external providers (so-called *cloud providers*). For example, a user that takes advantage of the cloud computing technology can store huge amount of data in on-line data storage or use remote software not directly installed in the own physical computer.

In the last years we are witnessing a continuous growth of the Cloud Computing paradigm. Some examples of applications whose success is based on such paradigm are *DropBox*, *Google Drive* and *Amazon Elastic Compute Cloud* (EC2).

The reasons of the success of the Cloud Computing paradigm are multiple. For example, users can substantially reduce the costs related to the maintenance of their IT infrastructures; moreover, thanks to the scalability offered by the Cloud Computing services, users can change their request of resources in the time, e.g. increasing or decreasing the required memory space or the CPU computation power. Surely, another strong point of the Cloud Computing is also represented by the fact that users can access to the external resources by using several devices (e.g. computers, smartphones, tablets, etc.) located in different places.

Even if it is a very powerful technology, the Cloud Computing is also the target of many criticisms related to the security risks to which the users are subjected.

The scenario we are interested in is that in which a user owns some data and wants to "outsource" it to a Cloud server (Fig. 1.1). Furthermore, the user may want also to delegate the computation over a subset of its data to the server.

Figure 1.1: Our scenario of interest.

In such scenario, if the server is *untrusted* (Fig. 1.2), how can the user verifies that the results provided by the server are *correct*? How can the user be sure that the server does not *steal* information from its data?

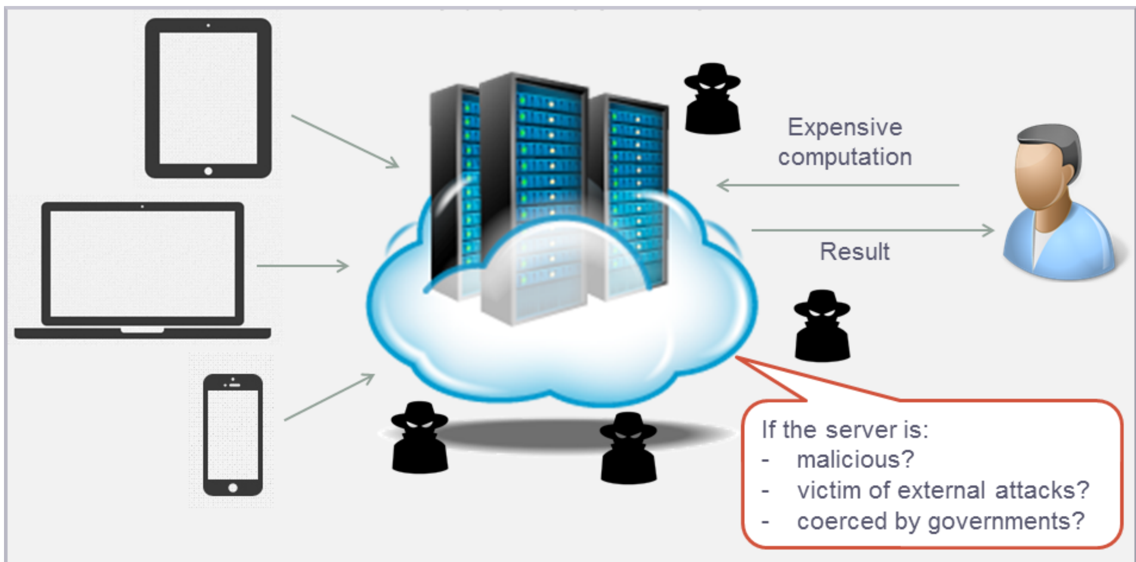

Figure 1.2: Our scenario of interest with an untrusted server.

The security issues related to the two above questions are known as *integrity* and *privacy*. By integrity we mean that the user wants to be sure, without having

to trust the server, that the computation made by the server is effectively the one requested; moreover the user wants to obtain the result in an efficient way, namely it should not have to re-perform the same work delegated to the server. By privacy we mean that the user wants to be sure that the server can not steal information from its data, as data might be sensitive (e.g. credit cards, medical records, etc.).

Of course, there could be applications in which the user needs only integrity but not privacy or the opposite, as well as applications where the user could need both properties.

To address the issues of privacy and integrity when outsourcing data and computation to untrusted entities, a very fervent research field in cryptography is trying to provide solutions through the development of new advanced cryptographic tools. This work of thesis follows recent results in this research area, and our main contributions are the following:

- *A unifying model for secure outsourcing.*

  There exists many works that provide protocols and schemes that aim to solve the integrity and privacy issues. For example, as we will see in detail in the following chapters, Homomorphic Message Authenticators (HMACs) ([3]) can provide secure outsourcing protocols for data and computation that solve the integrity problem; in the same way a Fully Homomorphic Encryption (FHE) scheme can provide a secure outsourcing protocol for data computation that solves the privacy problem.

  In this scenario, our contribution is to define a model that can unify and represent all such secure outsourcing protocols, namely protocols that guarantee integrity, privacy or integrity *and* privacy.

  Our main starting point was the recent work of D. Fiore, R. Gennaro, V. Pastro ([12]). In such article the authors propose a strong solution that allows to store large, privacy-sensitive, datasets on a server, and get statistics or distance measures (computed by the server) on them, with guarantees of *correctness* (i.e. integrity) and *privacy* of the data.

  In this thesis we re-define the model of [12] and we present a *generic* Outsourcing scheme; using such scheme it is possible to model several possible protocols for secure outsourcing that satisfy the two above properties of *integrity* and *privacy*. The Outsourcing scheme is as general as possible so as to capture all types of protocols, i.e. the protocols that guarantee only integrity, those that guarantee only privacy and those that guarantee both integrity and privacy.

- *Design of a framework for implementing protocols for secure outsourcing.*

  Starting from the theoretical model elaborated in the previous point, as a next we provide a framework able to implement all the protocols for secure

outsourcing. Such framework, written in C++ with the support of external libraries like `NTL` and `Crypto++`, has the goal of facilitating the implementation of new and existing protocols for secure outsourcing, providing common interfaces and tools. This also helps in making comparisons among different protocols.

- *Implementations of efficient schemes using the framework.*

  Finally, we use the novel framework developed in the previous point to realize implementations of recently proposed protocols, for which no implementation was known.

  In [12], the authors constructed highly efficient schemes for delegation of various classes of functions, such as linear combinations, high-degree univariate polynomials, and multivariate quadratic polynomials.

  In this work of thesis, we re-define one of these schemes, that we have called $\mathsf{Outsourcing}_{lin}$, using our new $\mathsf{Outsourcing}$ model. We also presente the Joye-Libert homomorphic encryption scheme that is an encryption scheme employed in the $\mathsf{Outsourcing}_{lin}$ scheme. Regarding the Joye-Libert encryption scheme, we want to highlight an interesting contribution of this thesis that consists of a *new decryption algorithm* that performs better than the Joye-Libert decryption algorithms.

  After that we propose the theoretical re-definition of the $\mathsf{Outsourcing}_{lin}$ scheme, we implement it using our new framework. As cross result, we also implement the Joye-Libert encryption scheme. Regarding such implementations, they have been realized also experiments to test the performances of the main algorithms.

The work of thesis is organized as follows. In the Chapter 2 we review the state of the art of some secure outsourcing protocols, like the Homomorphic Message Authenticators and the Fully Homomorphic Encryption already mentioned before. In the Chapter 3 we present our new $\mathsf{Outsourcing}$ unifying model for secure outsourcing, the Joye-Libert cryptosystem and the $\mathsf{Outsourcing}_{lin}$ scheme. Next, Chapter 4 contains the description of the framework. In the Chapter 5 we shows the implementations of the Joye-Libert cryptosystem and of the $\mathsf{Outsourcing}_{lin}$. Finally, the Chapter 6 contains the results of several experiments realized using the implementations described in the Chapter 5.

Chapter 2

Background

2.1 Preliminary definitions and notation

In this section, we review the notation and some basic definitions that we will use in our work. We will denote with $\lambda \in \mathbb{N}$ the security parameter, and by $\mathsf{poly}(\lambda)$ any function which is bounded by a polynomial in λ. We say that a function $\epsilon : \mathbb{N} \rightarrow \mathbb{R}^+$ is negligible if and only if for every positive polynomial $p(\lambda)$ there exists a $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0 : \epsilon(\lambda) < 1/p(\lambda)$, and we compactly denote it as $\epsilon(\lambda)$=$\mathsf{negl}(\lambda)$. If S is a set, $x \leftarrow_{\mathcal{R}} S$ denotes the process of selecting x uniformly at random in S. If $\mathcal{A}$ is a probabilistic algorithm $x \leftarrow_{\mathcal{R}} \mathcal{A}(\cdot)$ denotes the process of running $\mathcal{A}$ on some appropriate input and assigning its output to x.

2.2 The problem of secure outsourcing of data and computation

In the introduction, we have already mentioned the problem we are going to address and the related security issues. We have seen how the Cloud Computing is a very interesting and powerful paradigm; at the same time, the great rise in its employment is raising many security issues. Indeed, what is the guarantee that this *third-part entity* works correctly and honestly?

In this work we consider the following specific scenario of relevant interest in practice: there is a user A that owns some data (e.g. medical data collected by some sensors, such as blood pressure, blood oxygen level, etc.) and that wants to *continuously* "outsource" all such data to a Cloud server S. At certain point in time, A (or another user B) needs to make a computation over a subset of the data and decides to *delegate* this task to S.

It is interesting to highlight that this scenario perfectly fits to all those applications where one:

- wants to outsource data without fixing in advance the size of the data, namely it should always possible to add new data;
- does not know in advance which functions he will apply on the outsourced data, namely he may need many different computations (statistics).

Another important point to consider is that the user who can ask for the computation can also be an user different from the one who outsources the data.

How we have already said in the introduction, such scenario, in presence of an *untrusted* server leads to two important security problems:

- how can $A(B)$ verify that the results provided by the server are correct;
- how can A (B) be sure that the server does not violate its *privacy* accessing to its personal data.

The solutions that we will study in this work therefore aims to answer the above questions. More specifically, they aim to obtain the following two main properties:

1. ***privacy***, meaning that the server should *not learn information* about the user data as the data might be sensitive (e.g. credit cards, medical records, etc.) and A would like to keep them private;
2. ***integrity***, meaning that S must *not be able to send incorrect results* to A (B).

   Actually, the proposed solution will guarantee also other properties:

   (a) *security*, meaning that S should be able to "prove" the correctness of the delegated computation for some program $\mathcal{P}$;

   (b) *efficiency*, meaning that A (B) should be able to check the proof by requiring significantly fewer resources than those needed to compute $\mathcal{P}$;

   (c) *unbounded storage*, meaning that the size of the outsourced data should not be fixed a priori, i.e., A should be able to outsource any (possibly growing) amount of data;

   (d) *program-independence*, meaning that A should be able to outsource its data without having to know in advance the functions that it will delegate later.

In the next sections, we will examine some possible approaches to achieve these goals, showing, in order, how to obtain only integrity (see Sec. 2.3), only privacy (see Sec. 2.4) or integrity *and* privacy (Sec. 2.5).

2.3 An approach to solve *integrity*

Someone might be tempted to solve the *integrity* problem (2) by using standard cryptographic tool like *digital signatures*. Using digital signatures, the user A can outsource its data $m_1, ..., m_n$ to the cloud server S signing each of them, and later A can ask the server to run a program $\mathcal{P}$ over the outsourced data $(m_1, ..., m_n)$. The server computes $\mathcal{P}(m_1, ..., m_n)$ and sends the result m to A. The problem is that A wants to be sure that $m \stackrel{?}{=} \mathcal{P}(m_1, ..., m_n)$ and that the server applied $\mathcal{P}$ on *its* data. To this end, S could send back to A the original data with all the signatures giving to A the chance to verify the result by recomputing $\mathcal{P}(m_1, ..., m_n)$. Clearly, this is not an efficient solution because to obtain a simple result, the server has to send back an entire portion of the database, A should store it and redo the computation (whereas this is the reason for which it is paying the cloud provider). This simple example shows that a standard cryptographic tool cannot help A. In the next section we will present a more advanced cryptographic technique that can be employed to solve the integrity problem in this scenario.

2.3.1 Homomorphic message authenticators with efficient verification

There exists a huge and fervent research field, called *Verifiable computation* (VC) that deals with the problem of *securely* and *efficiently* outsourcing the computation of a function f to a remote server.

[1] Verifiable computing (or verified computation or verified computing) is enabling a computer to offload the computation of some function to other perhaps untrusted clients, while maintaining verifiable results. The other clients evaluate the function and return the result with a proof that the computation of the function was carried out correctly. The term "verifiable computing" was formalized by Rosario Gennaro, Craig Gentry, and Bryan Parno [14]. Verifiable computing is not only concerned with getting the result of the outsourced function on the client's input and the proof of its correctness, but also with the client being able to verify the proof with significantly less computational effort than computing the function from scratch.

Most of the work on VC is only concerned with verifying the correctness of function's outputs *while knowing the input*. The problem of VC when the input is stored at the server has been addressed by the recent work of M. Backes, D. Fiore and R. M. Reischuk [3]. In [3] they proposed the first *practical* protocol that achieves all the goals stated in Sec. 2.2 except for *privacy*. The protocol of [3] allows a client

[1] https://en.wikipedia.org/wiki/Verifiable_computing.

to (continuously) store a large amount of data $D = D_1, D_2, D_3...$ on the server, and then, at certain points in time, to request the computation of a function f on (a portion of) the outsourced data, e.g., $v = f(D_{i1}, ..., D_{in})$. Using their protocol, the server sends to the client a short piece of information vouching for the correctness of v. The client checks this proof to verify the result of the computation.

The protocol achieves an additional property, namely the *input-independent efficiency* property, meaning that verifying the correctness of a computation $f(D_1, D_2, D_3, ..., D_n)$ requires time *independent of* n. Such property is guaranteed in the *amortized model*: after a single precomputation with cost $|f|$, the client can verify *every* subsequent evaluation of f in *constant time*, i.e., regardless of the input size n on *his* data.

Their key technical contribution was the introduction of *homomorphic MACs with efficient verification*. This cryptographic primitive extends homomorphic message authenticators [15] by adding a crucial efficiency property for the verification algorithm. They proposed a first realization of homomorphic MACs with efficient verification, proving its security under the Decision Linear assumption [6].

The basic idea of homomorphic MACs is that a user can use a secret key to generate a set of tags $\sigma_1, ..., \sigma_n$ authenticating values $D_1, ..., D_n$ respectively. Then, anyone can homomorphically execute a function f over $(\sigma_1, ..., \sigma_n)$ to generate a short tag σ that authenticates D as the output of $f(D_1, ..., D_n)$.

At first glance, homomorphic MACs seem to perfectly fit the problem of verifiable computations on (growing) outsourced data. However, this primitive reveals that this idea lacks the very important property of *efficient verification*. The issue is that in all existing constructions the verification algorithm of homomorphic MACs runs in time proportional to the description of the function.

In [3] their contribution was therefore to solve this efficiency issue by proposing a definition and a first practical realization of homomorphic MACs with efficient verification. In the following section we present a formal model and definitions for this notion. Most of the definitions are taken from [3].

The model: multi-labeled programs

The basic idea behind the model is to introduce the notion of a *multi-label* L to *authenticate* a value m; the intention is to uniquely *remember* the outsourced data.

A multi-label L consists of two parts: a *data set identifier* Δ and an *input identifier* τ. Input identifiers, in isolation, are used to label the variable inputs of a function f, whereas the combination of both, i.e., the full multi-label $\mathsf{L} = (\Delta, \tau)$, is used to *uniquely* identify a specific data item. Precisely, binding a value m with multi-label (Δ, τ) means that m can be assigned to those input variables with input

identifier τ. Indeed, one cannot re-use a pair (Δ, τ) for authentication purposes but can re-use the input identifier τ. For the sake of illustration, consider the multi-labeled approach as a separation of data items into two independent dimensions. One might think of a database table (Fig. 2.1), e.g., storing blood oxygen levels, where some function $f : \mathcal{M}^n \to \mathcal{M}$ is evaluated over n rows (labeled $\tau_1, ..., \tau_n$). Each such row could represent a point in time, e.g., 7:05, 07:10, etc. These rows are grouped into many groups and each group could represent a different day, e.g., 2015/06/20, 2015/06/21, etc. (Fig. 2.2). The computation is performed for a group of rows (labeled Δ_i) of the table. We hence evaluate $f_{\Delta_i}(\tau_1, ..., \tau_n)$ for each group of rows i, hence for each day.

Blood O_2 level	Point in time	Day
96	7:05	2015/06/20
95	8:05	2015/06/20
98	9:05	2015/06/20
94	10:05	2015/06/20
95	11:05	2015/06/20
...	...	...
98	7:05	2015/06/21
99	8:05	2015/06/21
96	9:05	2015/06/21
97	10:05	2015/06/21
95	11:05	2015/06/21
...	...	...

Δ_1 (rows dated 2015/06/20)

Δ_i (rows dated 2015/06/21)

Figure 2.1: Example of a table structure in the multi-label model.

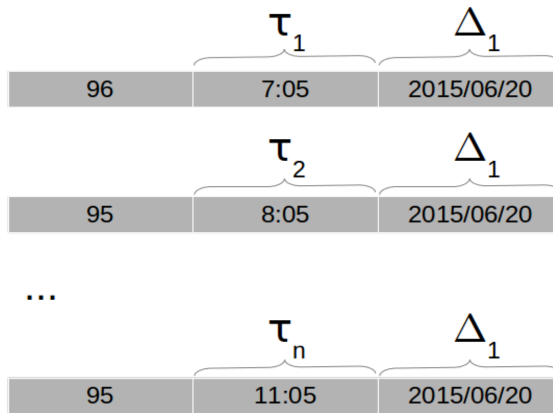

Figure 2.2: Multi-label model.

In order to define the notion of multi-labeled program, it is necessary to review the notion of labeled program introduced in [15] for the case of Boolean circuits $f : \{0,1\}^n \rightarrow \{0,1\}$, here generalizing it to the case of any function f defined over an appropriate set $\mathcal{M}$.

Labeled Programs. A *labeled program* $\mathcal{P}$ is defined by a tuple $(f, \tau_1, ..., \tau_n)$ where $f : \mathcal{M}^n \rightarrow \mathcal{M}$ is a function on n variables, and *each* $\tau_i \in \{0,1\}^*$ *is the label of the* i*-th variable input of* f. Labeled programs allow for composition as follows. Given labeled programs $\mathcal{P}_1, ..., \mathcal{P}_t$ and given a function $g : \mathcal{M}^t \rightarrow \mathcal{M}$, the *composed program* $\mathcal{P}^*$ corresponds to evaluating g on the outputs of $\mathcal{P}_1, ..., \mathcal{P}_t$. The composed program is compactly denoted as $\mathcal{P}^* = g(\mathcal{P}_1, ..., \mathcal{P}_t)$. The labeled inputs of $\mathcal{P}^*$ are all distinct labeled inputs of $\mathcal{P}_1, ..., \mathcal{P}_t$, i.e., all inputs with the same label are grouped together in a single input of the new program. If $f_{id} : \mathcal{M} \rightarrow \mathcal{M}$ is the canonical identity function[2] and $\tau \in \{0,1\}^*$ is a label, then $\mathcal{I}_\tau = (f_{id}, \tau)$ denotes the *identity program* for input label τ. Notice that any program $\mathcal{P} = (f, \tau_1, ..., \tau_n)$ can be expressed as the composition of n identity programs $\mathcal{P} = f(\mathcal{I}_{\tau_1}, ..., \mathcal{I}_{\tau_n})$.

Multi-labeled Programs. Formally, a *multi-labeled program* $\mathcal{P}_\Delta$ is defined as a pair $(\mathcal{P}, \Delta)$ where $\mathcal{P} = (f, \tau_1, ..., \tau_n)$ is a labeled program (as defined above) and $\Delta \in \{0,1\}^*$ is a binary string called the data set identifier. Multi-labeled programs allow for composition within the same data set in the most natural way, i.e., given multi-labeled programs $(\mathcal{P}_1, \Delta), ..., (\mathcal{P}_t, \Delta)$ having the same data set identifier Δ, and given a function $g : \mathcal{M}^t \rightarrow \mathcal{M}$, the *composed multi-labeled program* $\mathcal{P}^*_\Delta$ is the pair $(\mathcal{P}^*, \Delta)$ where $\mathcal{P}^*$ is the composed program $g(\mathcal{P}_1, ..., \mathcal{P}_t)$, and Δ is the data set identifier shared by all the $\mathcal{P}_i$. If $f_{id} : \mathcal{M} \rightarrow \mathcal{M}$ is the canonical identity function and $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^*)^2$ is a multi-label, then $\mathcal{I}_\mathsf{L} = (f_{id}, \mathsf{L})$ denotes the *identity multi-labeled program* for data set Δ and input label τ. As for labeled programs, any multi-labeled program $\mathcal{P}_\Delta = ((f, \tau_1, ..., \tau_n), \Delta)$ can also be expressed as the composition of n identity multi-labeled programs: $\mathcal{P}_\Delta = f(\mathcal{I}_{\mathsf{L}_1}, ..., \mathcal{I}_{\mathsf{L}_n})$ where $\mathsf{L}_i = (\Delta, \tau_i)$.

The main difference in a multi-labeled program is the (explicit) notion of labeled data sets that it is used in order to group together several inputs. This explicit splitting will turn out to be crucial in order to achieve the desired property of efficient verification.

[2]In mathematics, an identity function, also called an identity relation or identity map or identity transformation, is a function that always returns the same value that was used as its argument. In equations, the function is given by $f(x) = x$.

Homomorphic message authenticators for multi-labeled programs

Definition 2.1 A homomorphic message authenticator scheme $\mathsf{HomMAC-ML}$ for multi-label programs is a *tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Auth}, \mathsf{Ver}, \mathsf{Eval})$ satisfying four properties: *authentication correctness, evaluation correctness, succinctness, and security.* More precisely:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{sk})$**:** given the security parameter λ, the key generation algorithm outputs a *secret key* sk and a *public evaluation key* ek.

$\mathsf{Auth}(\mathsf{sk}, \mathsf{L}, m) \to \sigma$**:** given the secret key sk, a multi-label $\mathsf{L} = (\Delta, \tau)$ and a message $m \in \mathsf{M}$, it outputs a tag σ.

$\mathsf{Eval}(\mathsf{ek}, f, \boldsymbol{\sigma}) \to \sigma$**:** on input the evaluation key ek, a circuit $f : \mathcal{M}^n \to \mathcal{M}$ and a vector of tags $\boldsymbol{\sigma} = (\sigma_1,..., \sigma_n)$, the evaluation algorithm outputs a new tag σ.

$\mathsf{Ver}(\mathsf{sk}, \mathcal{P}_\Delta, m, \sigma) \to 0/1$**:** given the secret key sk, a multi-labeled program $\mathcal{P}_\Delta = ((f, \tau_1, ..., \tau_n), \Delta)$, a message $m \in \mathcal{M}$, and a tag σ, the verification algorithm outputs 0 (reject) or 1 (accept).

Authentication Correctness. Informally speaking, a homomorphic MAC has authentication correctness if any tag σ generated by the algorithm $\mathsf{Auth}(\mathsf{sk}, \mathsf{L}, m)$ authenticates m with respect to the identity program $\mathcal{I}_\mathsf{L}$. More formally, we say that a scheme $\mathsf{HomMAC-ML}$ satisfies authentication correctness if for any message $m \in \mathcal{M}$, all keys $(\mathsf{sk}, \mathsf{ek}) \leftarrow_\mathcal{R} \mathsf{KeyGen}(1^\lambda)$, any multi-label $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^*)^2$, and any tag $\sigma \leftarrow_\mathcal{R} \mathsf{Auth}(\mathsf{sk}, \mathsf{L}, m)$, we have that $\mathsf{Ver}(\mathsf{sk}, \mathcal{I}_\mathsf{L}, m, \sigma) = 1$ holds with probability 1.

Evaluation Correctness. This property aims at capturing that if the evaluation algorithm is run on a vector of tags $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$ such that each σ_i authenticates some message m_i as the output of a multi-labeled program $(\mathcal{P}_i, \Delta)$, then the tag σ produced by Eval must authenticate $f(m_1, ..., m_n)$ as the output of the composed program $(f(\mathcal{P}_1, ..., \mathcal{P}_n), \Delta)$. More formally, let us fix a pair of keys $(\mathsf{sk}, \mathsf{ek}) \leftarrow_\mathcal{R} \mathsf{KeyGen}(1^\lambda)$, a function $g : \mathcal{M}^t \to \mathcal{M}$ and any set of message/program/tag triples $\{(m_i, \mathcal{P}_{\Delta,i}, \sigma_i)\}_{i=1}^t$ such that all multi-labeled programs $\mathcal{P}_{\Delta,i} = (\mathcal{P}_i, \Delta)$ (i.e., sharing the same data set identifier Δ) and $\mathsf{Ver}(\mathsf{sk}, \mathcal{P}_{\Delta,i}, m_i, \sigma_i) = 1$. If $m^* = g(m_1, ..., m_t), \mathcal{P}^* = g(\mathcal{P}_1, ..., \mathcal{P}_t)$, and $\sigma^* = \mathsf{Eval}(\mathsf{ek}, g, (\sigma_1, ..., \sigma_t))$, then $\mathsf{Ver}(\mathsf{sk}, \mathcal{P}^*_\Delta, m^*, \sigma^*) = 1$ holds with probability 1.

Succinctness. The size of a tag is bounded by some fixed polynomial in the security parameter, which is independent of the number n of inputs taken by the evaluated circuit.

Security. A homomorphic MAC has to satisfy the following notion of unforgeability. Let $\mathsf{HomMAC-ML}$ be a homomorphic MAC scheme as defined above and

let $\mathcal{A}$ be an adversary. $\mathsf{HomMAC-ML}$ is said to be unforgeable if for every PPT adversary. $\mathcal{A}$, we have $\Pr[\mathsf{HomUF-CMA}_{\mathcal{A},\mathsf{HomMAC-ML}}(\lambda) = 1] \leq \epsilon(\lambda)$ where $\epsilon(\lambda)$ is a negligible function. The experiment $\mathsf{HomUF-CMA}_{\mathcal{A},\mathsf{HomMAC-ML}}(\lambda)$ is the one defined below.

SETUP The challenger generates $(\mathsf{sk}, \mathsf{ek}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$ and gives ek to $\mathcal{A}$.

AUTHENTICATION QUERIES The adversary can adaptively ask for tags on multi-labels and messages of its choice. Given a query (L, m) where $\mathsf{L} = (\Delta, \tau)$, the challenger proceeds as follows: If (L, m) is the first query with data set identifier Δ, then the challenger initializes an empty list $T_\Delta = \emptyset$ for data set identifier Δ. If T_Δ does not contain a tuple $(\tau, \cdot)$ (i.e., the multi-label (Δ, τ) was never queried), the challenger computes $\sigma \leftarrow_{\mathcal{R}} \mathsf{Auth}(\mathsf{sk}, \mathsf{L}, m)$, returns σ to $\mathcal{A}$ and updates the list $T_\Delta \leftarrow T_\Delta \cup (\tau, m)$. If $(\tau, m) \in T_\Delta$ (i.e., the query was previously made), then the challenger replies with the same tag generated before. If T_Δ contains a tuple (τ, m') for some message $m' \neq m$, then the challenger ignores the query.

VERIFICATION QUERIES The adversary has access to a verification oracle as follows: Given a query $(\mathcal{P}_\Delta, m, \sigma)$ from $\mathcal{A}$, the challenger replies with the output of $\mathsf{Ver}(\mathsf{sk}, \mathcal{P}_\Delta, m, \sigma)$.

FORGERY The adversary terminates the experiment by returning a forgery $(\mathcal{P}^*_{\Delta^*}$, m^*, $\sigma^*)$ for some $\mathcal{P}^*_{\Delta^*} = (\mathcal{P}^*, \Delta^*)$ and $\mathcal{P}^* = (f^*, \tau^*_1, ..., \tau^*_n)$. Notice that, equivalently, $\mathcal{A}$ can implicitly return such a tuple as a verification query $(\mathcal{P}^*_{\Delta^*}, m^*, \sigma^*)$ during the experiment.

Before describing the outcome of this experiment, it is necessary to review the notion of well-defined programs with respect to a list T_Δ [9]. A labeled program $\mathcal{P}^*$ $= (f^*, \tau^*_1, ..., \tau^*_n)$ is well-defined with respect to T_{Δ^*} if either one of the following two cases holds:

- there exist messages m_1, ..., m_n such that the list T_{Δ^*} contains all tuples $(\tau^*_1$, $m_1), ...$,$(\tau^*_n$, $m_n)$. Intuitively, this means that the entire input space of f for data set Δ^* has been authenticated;
- there exist indices $i \in \{1, ... , n\}$ such that $(\tau^*_i, \cdot) \notin T_{\Delta^*}$ (i.e., $\mathcal{A}$ never asked authentication queries with multi-label (Δ^*, τ^*_i)), and the function $f^*(\{m_j\}_{(\tau_j, m_j) \in T_{\Delta^*}}$ $\cup \{\tilde{m}_j\}_{(\tau_j, \cdot) \notin T_{\Delta^*}})$ outputs the same value for all possible choices of $\tilde{m}_j \in \mathcal{M}$. Intuitively, this case means that the unauthenticated inputs never contribute to the computation of f.

To define the output of the experiment $\mathsf{HomUF-CMA}$, we said it outputs 1 if and only if $\mathsf{Ver}(\mathsf{sk}, \mathcal{P}^*_{\Delta^*}, m^*, \sigma^*) = 1$ and one of the following conditions holds:

- *Type 1 Forgery:* no list T_{Δ^*} was created during the game, i.e., no message m has been authenticated with respect to a data set identifier Δ^* during the experiment;

- *Type 2 Forgery:* $\mathcal{P}^*$ is well-defined w.r.t. T_{Δ^*} and $m^* \neq f^*(\{m_j\}_{(\tau_j, m_j) \in T_{\Delta^*}})$, i.e., m^* is not the correct output of the labeled program $\mathcal{P}^*$ when executed on previously authenticated messages $(m_1,\ldots, m_n)$;

- *Type 3 Forgery:* $\mathcal{P}^*$ is not well-defined w.r.t. T_{Δ^*}.

This definition is obtained by extending the one in [9] to the model of multi-labeled programs. The resulting definition is very close to the one proposed by Freeman for homomorphic signatures [13], with the exception that they allow for arbitrary labels, and they do not impose any a-priori fixed bound on the number of elements in a data set. In the most general case where f can be any function, it might not be possible to efficiently (i.e., in polynomial time) check whether a program $\mathcal{P}$ is well-defined w.r.t. a list T. However, for more specific classes of computations, this is not an issue. In the following proposition it is showna similar result for the classes of computations considered in [3], i.e., arithmetic circuits defined over the finite field $\mathbb{Z}_p$ where p is a prime of roughly λ bits, and whose degree d is bounded by a polynomial. In particular, they showed that any adversary who wins by producing a *Type 3 forgery* can be converted into one who outputs a *Type 2 forgery.*

Just for completeness, we give the following proposition whose proof appears in [3].

Proposition 2.1 *Let $\lambda \in \mathbb{N}$ be the security parameter, let $p > 2^\lambda$ be a prime number, and let $\{f_\lambda\}$ be a family of arithmetic circuits over $\mathbb{Z}_p$ whose degree is bounded by some polynomial $d = poly(\lambda)$. If for any adversary $\mathcal{B}$ producing a Type 2 forgery we have that $\Pr[\mathsf{HomUF-CMA}_{\mathcal{B},\mathsf{HomMAC-ML}}(\lambda) = 1] \leq \epsilon$, then for any adversary $\mathcal{A}$ producing a Type 3 forgery it holds $\Pr[\mathsf{HomUF-CMA}_{\mathcal{A},\mathsf{HomMAC-ML}}(\lambda) = 1] \leq \epsilon + d/p$.*

Homomorphic message authenticators with efficient verification for multi-labeled programs

In this section it is introduced the new property of *efficient verification* for homomorphic MACs. Informally, a homomorphic MAC satisfies efficient verification if it is possible to verify a tag σ against a multi-labeled program $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ in *less time than that required to compute* $\mathcal{P}$. This efficiency property is defined in an *amortized sense*, so that the verification is more efficient when the same program $\mathcal{P}$ is executed on different data sets. The formal definition follows.

Definition 2.2 *Let $\mathsf{HomMAC-ML} = (\mathsf{KeyGen}, \mathsf{Auth}, \mathsf{Ver}, \mathsf{Eval})$ be a homomorphic MAC scheme for multi-labeled programs as defined in the previous section. $\mathsf{HomMAC-ML}$ satisfies efficient verification if there exist two additional algorithms $(\mathsf{VerPrep}, \mathsf{EffVer})$ as follows:*

$\mathsf{VerPrep}(\mathsf{sk}, \mathcal{P}) \rightarrow \mathsf{VK}_{\mathcal{P}}$**:** on input the secret key sk and a labeled program $\mathcal{P} = (f, \tau_1, ..., \tau_n)$, this algorithm generates a concise verification key $\mathsf{VK}_{\mathcal{P}}$. We stress that this verification key *does not depend on any data set identifier* Δ.

$\mathsf{EffVer}(\mathsf{sk}, \mathsf{VK}_{\mathcal{P}}, \Delta, m, \sigma) \rightarrow 0/1$**:** given the secret key sk, a verification key $\mathsf{VK}_{\mathcal{P}}$, a data set identifier Δ, a message $m \in \mathsf{M}$ and a tag σ, the efficient verification algorithm outputs 0 (reject) or 1 (accept).

The above algorithms are required to satisfy the following two properties:

Correctness. Let $(\mathsf{sk}, \mathsf{ek}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$ be honestly generated keys, and $(\mathcal{P}_\Delta, m, \sigma)$ be any program/message/tag tuple with $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ such that $\mathsf{Ver}(\mathsf{sk}, \mathcal{P}_\Delta, m, \sigma) = 1$. Then, for every $\mathsf{VK}_{\mathcal{P}} \leftarrow_{\mathcal{R}} \mathsf{VerPrep}(\mathsf{sk}, \mathcal{P})$, we have $\Pr[\mathsf{EffVer}(\mathsf{sk}, \mathsf{VK}_{\mathcal{P}}, \Delta, m, \sigma) = 1] = 1$.

Amortized efficiency. Let $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ be a program, let $(m_1, ..., m_n) \in \mathcal{M}^n$ be any vector of inputs, and let $t(n)$ be the time required to compute $\mathcal{P}(m_1, ..., m_n)$. If $\mathsf{VK}_{\mathcal{P}} \leftarrow_{\mathcal{R}} \mathsf{VerPrep}(\mathsf{sk}, \mathcal{P})$, then the time required for $\mathsf{EffVer}(\mathsf{sk}, \mathsf{VK}_{\mathcal{P}}, \Delta, m, \sigma)$ is $O(1)$, i.e., independent of n.

Notice that in this efficiency requirement, it is not included the time needed to compute $\mathsf{VK}_{\mathcal{P}}$. The reason is, since $\mathsf{VK}_{\mathcal{P}}$ is independent of Δ, the same $\mathsf{VK}_{\mathcal{P}}$ can be re-used in many verifications involving the same labeled program P but many different Δ. In this sense, the cost of computing $\mathsf{VK}_{\mathcal{P}}$ is *amortized* over many verifications of the same function on different data sets.

Application to Verifiable Computation on Outsourced Data.

A homomorphic MAC scheme with efficient verification can be easily used to obtain a protocol for verifiable delegation of computations on outsourced data, achieving goals (2.a - 2.d) mentioned in Sec. 2.2. A sketch of such a protocol between a client $\mathcal{C}$ and a server $\mathcal{S}$ follows:

Setup: $\mathcal{C}$ generates the keys $(\mathsf{sk}, \mathsf{ek}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$ for a homomorphic MAC, sends ek to $\mathcal{S}$ and stores sk.

Data Outsourcing: to outsource a value m, $\mathcal{C}$ first authenticates m wrt. some multi-label L, i.e., $\sigma \leftarrow_{\mathcal{R}} \mathsf{Auth}(\mathsf{sk}, \mathsf{L}, m)$, and then sends (m, L, σ) to the server. It is easy to see that this phase achieves the goal of *unbounded storage* (2.c) and *function independence* (2.d).

Client's Preparation: assume that $\mathcal{C}$ needs to evaluate a labeled program $\mathcal{P} = (f, \tau_1, ..., \tau_n)$ on some of its outsourced data sets. In this preparation phase (offline), the client computes and stores $\mathsf{VK}_{\mathcal{P}} \leftarrow_{\mathcal{R}} \mathsf{VerPrep}(\mathsf{sk}, \mathcal{P})$ (independently of any Δ).

Delegation: when the client wants to compute $\mathcal{P}$ on a data set Δ (online), it simply sends $(\mathcal{P},\ \Delta)$ to the server [3].

Computation: to compute $(\mathcal{P},\ \Delta)$, where $\mathcal{P} = (f,\ \tau_1,\ ...,\ \tau_n)$, the server first looks for the corresponding data $(m_1,\ ...,\ m_n)$ and tags $(\sigma_1,\ ...,\ \sigma_n)$ according to the labeling previously sent by $\mathcal{C}$. Next, $\mathcal{S}$ computes $m = f(m_1,\ ...,\ m_n)$ and $\sigma \leftarrow \mathsf{Eval}(\mathsf{ek}, f, \sigma_1, ..., \sigma_n)$, and sends $(m,\ \sigma)$ to $\mathcal{C}$.

Verification: given the result $(m,\ \sigma)$ sent by $\mathcal{S}$, the client checks that m is the correct output of the multi-labeled program $(\mathcal{P},\ \Delta)$ by running $\mathsf{EffVer}(\mathsf{sk}, \mathsf{VK}_{\mathcal{P}},\ \Delta,\ m,\ \sigma)$. By the amortized efficiency property of the homomorphic MAC, we obtain that $\mathcal{C}$ achieves amortized *input-independent efficiency* - and thus also *efficiency* (2.b) - in verifying the delegated computations.

Fig.2.3 shows a graphic sketch of the previous protocol.

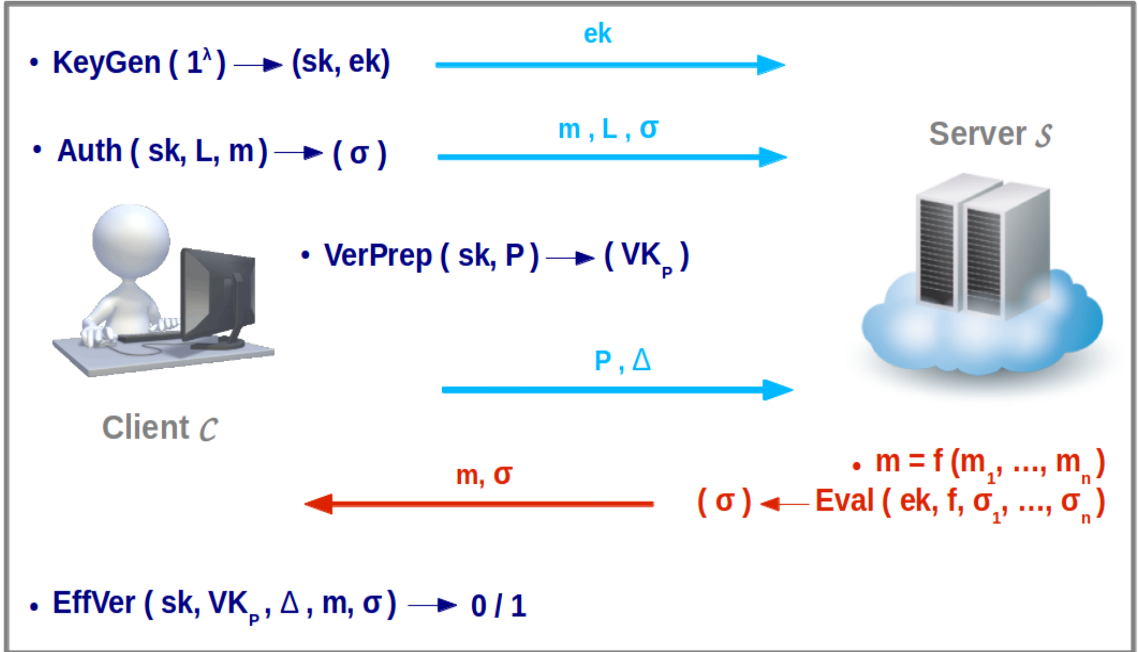

Figure 2.3: Protocol for verifiable delegation of computations on outsourced data using a homomorphic MAC scheme with efficient verification.

[3] While in general the description of $\mathcal{P}$ may be large, here we assume the case in which $\mathcal{P}$ has a succinct description, e.g., “daily variance of the air pollution levels at every 5 minutes”. Hence, the cost of communicating $\mathcal{P}$ can, in fact, be ignored.

2.4 An approach to solve *privacy*

Now we turn to the problem of achieving privacy when delegating computation.

In Sec. 2.3 we have seen how a standard cryptographic tool such as digital signatures can not solve the integrity problem. Here we briefly show how again another standard cryptographic tool, such as *encryption*, cannot help to solve the privacy issue.

Let's consider the same initial example of the medical data. Someone might be tempted to solve the *privacy* problem by using (standard) encryption: in order to prevent leakage of sensitive information to the cloud service provider, the user A first encrypts the data to produce ciphertexts. He then uploads the ciphertexts to the cloud server S (probably without having to keep a copy of the data due to its limited local storage capacity). When, later, A wishes to derive some information from its data, such as the presence of a disease, it sends an instruction to S, specifying a program $\mathcal{P}$ to be executed on his data.

Now, S is not able to read the encrypted data and it can carry on the computation only having the A's secret key. A possible solution could be that S sends the ciphertexts back to A so that it could compute $\mathcal{P}$ by itself. Once again, this is not an efficient solution because to obtain a simple result, S has to send back a huge amount of data and the client has to store them, decrypt them and then make the computation. Even in this case, a standard cryptographic tool cannot be of help. As we show in the next section, the question about how to solve privacy in this setting, can be solved in principle with a very powerful cryptographic tool that is called *Fully Homomorphic Encryption (FHE)*.

2.4.1 Fully homomorphic encryption

FHE is a public key encryption scheme, defined as usual, by a key generation algorithm, an encryption algorithm and a decryption algorithm, but with the addition of a new algorithm, the *evaluation* algorithm. Such algorithm has the capability to execute a function f over ciphertexts in such a way that what one obtains is another ciphertext; this ciphertext, when decrypted, gives the output of the function over the original data. In this sense the scheme is homomorphic, because when one operates over the ciphertexts (that are encodings of the inputs) is like one is performing the same operation homomorphically on the original inputs.

As usually, we want that a FHE scheme must be secure, in particular it should satisfy the standard notion of *semantic security*.

Definition 2.3 (Semantic security). *Let* $\mathcal{HE}=(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *be a (homomorphic) encryption scheme, and* $\mathcal{A}$ *be a PPT adversary. Consider the following*

experiment:

Experiment $\mathbf{Exp}^{\mathsf{SS}}_{\mathcal{HE},\mathcal{A}}(\lambda)$
$b \leftarrow_{\mathcal{R}} \{0,1\}$; $(\mathsf{pk},\mathsf{sk}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$
$(m_0, m_1) \leftarrow \mathcal{A}(\mathsf{pk})$
$c \leftarrow_{\mathcal{R}} \mathsf{Enc}(\mathsf{pk}, m_b)$
$b' \leftarrow \mathcal{A}(c)$
If $b' = b$ *return 1. Else return 0.*

and define $\mathcal{A}$*'s advantage as* $\mathbf{Adv}^{\mathsf{SS}}_{\mathcal{HE},\mathcal{A}}(\lambda)=\Pr[\mathbf{Exp}^{\mathsf{SS}}_{\mathcal{HE},\mathcal{A}}(\lambda)=1]-\frac{1}{2}$. *Then we say that* $\mathcal{HE}$ *is* semantically-secure *if for any PPT algorithm* $\mathcal{A}$ *it holds* $\mathbf{Adv}^{\mathsf{SS}}_{\mathcal{HE},\mathcal{A}}(\lambda)= \mathsf{negl}(\lambda)$.

So, if FHE gives us semantic security, we achieve privacy, while efficiency is guaranteed by another property of HE called compactness, that basically means that the ciphertext is not growing with the complexity of the function. Program independence is also guaranteed by the definition (when you are encrypting the plaintext you are not specifying any function).

While the first FHE scheme proposed by Gentry in 2009 ([16]) basically consisted of theoretical proofs of concepts and was not practical, in the recent years there have been significant advances ([7], [8]).

In what follows we recall the formal definitions of FHE.

Formally, a *fully homomorphic (public-key) encryption* (FHE) scheme is a tuple of PPT algorithms $\mathsf{FHE} = (\mathsf{FHE.ParamGen}, \mathsf{FHE.KeyGen}, \mathsf{FHE.Enc}, \mathsf{FHE.Dec}, \mathsf{FHE.Eval})$ defined as follows.

$\mathsf{FHE.ParamGen}(1^\lambda)$: defines the parameters for the scheme, such as plaintext space $\mathcal{M}$, ciphertext space, keyspace, randomness distributions, etc. The output of $\mathsf{ParamGen}$ is assumed to be input to any subsequent algorithm.

$\mathsf{FHE.KeyGen}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{evk}, \mathsf{dk})$: outputs a public encryption key pk, a public evaluation key evk, and a secret decryption key dk.

$\mathsf{FHE.Enc}(\mathsf{pk}, m) \rightarrow c$: encrypts message $m \in \mathcal{M}$ under public key pk. It outputs a ciphertext c.

$\mathsf{FHE.Dec}(\mathsf{dk}, c) \rightarrow m$: decrypts the ciphertext c using dk to a plaintext $m \in \mathcal{M}$.

$\mathsf{FHE.Eval}(\mathsf{evk}, g, c_1, ..., c_t) \rightarrow c^*$: given the evaluation key evk, a circuit $g : \mathcal{M}^t \rightarrow \mathcal{M}$, and a set of t ciphertexts c_1, ..., c_t, deterministically compute and output a ciphertext c^*.

An FHE should also satisfy the following properties.

Encryption Correctness. For all $m \in \mathcal{M}$ we have:

$$\Pr\Big[\mathsf{FHE.Dec}_{\mathsf{dk}}(\mathsf{FHE.Enc}_{\mathsf{pk}}(m)) = m \mid (\mathsf{pk}, \mathsf{evk}, \mathsf{dk}) \leftarrow \mathsf{FHE.KeyGen}(\lambda)\Big] = 1$$

Evaluation Correctness. For $(\mathsf{pk}, \mathsf{evk}, \mathsf{dk}) \leftarrow \mathsf{FHE.KeyGen}(\lambda)$, any ciphertexts c_1, ...,c_t such that $\mathsf{FHE.Dec}_{\mathsf{dk}}(c_i) = m_i \in \mathcal{M}$, and any circuit $g : \mathcal{M}^t \to \mathcal{M}$, we have

$$\mathsf{FHE.Dec}_{\mathsf{dk}}(\mathsf{FHE.Eval}_{\mathsf{evk}}(g, c_1, ..., c_t)) = g(m_1, ..., m_t)$$

Succinctness. The ciphertext size is bounded by some fixed polynomial in the security parameter, and is independent of the size of the evaluated circuit or the number of inputs it takes. I.e. there exists some polynomial p such that, for any $(\mathsf{pk}, \mathsf{evk}, \mathsf{dk}) \leftarrow \mathsf{FHE.KeyGen}(\lambda)$, the output size of $\mathsf{FHE.Enc}_{\mathsf{pk}}$ and of $\mathsf{Eval}_{\mathsf{evk}}$ is bounded by p, for any choice of their inputs.

Semantic security. An FHE is a *semantically secure* public-key encryption scheme, where we consider the evaluation key evk as a part of the public key. I.e. for any PPT attacker $\mathcal{A}$:

$$|\Pr[\mathcal{A}(\lambda, \mathsf{pk}, \mathsf{evk}, c_0) = 1] - \Pr[\mathcal{A}(\lambda, \mathsf{pk}, \mathsf{evk}, c_1) = 1]| \leq \mathsf{negl}(\lambda)$$

where the probability is over $(\mathsf{pk}, \mathsf{evk}, \mathsf{dk}) \leftarrow \mathsf{FHE.KeyGen}(\lambda)$, $c_b \leftarrow \mathsf{FHE.Enc}_{\mathsf{pk}}(m_b)$, $m_0, m_1 \leftarrow \mathcal{M}$, and the coins of $\mathcal{A}$.

2.5 An approach to solve integrity *and* privacy

Summarizing, in the two previous sections we have seen the Homomorphic MACs as possible solution to solve the integrity problem without guaranteeing privacy (the fact that the server sends the correct result does not prevent the server to learn information about the client data) and FHE as possible solution to solve the privacy issue without guaranteeing integrity (the fact that the server operates over encrypted data does not prevent the server to cheat, sending incorrect results).

A possible solution that aims to guarantee integrity and privacy has been recently proposed by D. Fiore, R. Gennaro and V. Pastro in [12]. The authors, in [12], studied the task of verifiable delegation on encrypted data. They improved previous definitions by adding *verification queries*. Their contribution was:

- *theoretical*, with the definition of a *generic* VC scheme with *privacy in the presence of verification queries* and with the generic construction of a scheme to obtain "integrity and privacy" for arbitrary computations;

- *practical*, with the constructions of highly efficient schemes for delegation of various classes of specific functions with applications, i.e. verifiable statistics (average, variance, covariance, correlation error, etc.) and distance measures over encrypted data.

In the next chapter, using as starting point the article of [12], we present our new theoretical contribution, showing a *generic* and *flexible* model for representing *secure* outsourcing schemes.

Chapter 3

A unifying model for secure outsourcing

We concluded the previous chapter seeing that new interesting research works are trying to solve the problem of achieving both integrity and privacy.

Following the ideas present in one of these recent works ([12]), our contribution is:

1. *theoretical*, with the definition of an Outsourcing scheme (see Sec. 3.1 and Sec. 3.2) that aims to provide a *flexible* and *generic* model to allow the outsourcing of data on a remote server and the delegation of a computation over such data. Starting from the VC scheme of [12] we have studied new definitions in order to capture the case in which one wants only privacy, only integrity or both; the new Outsourcing scheme is a re-elaborated version of the VC scheme of [12] adapted to the multi-labeled model presented in Sec. 2.3.1.2;

2. *practical* (see Sec. 3.3), with the redefinition of one of the efficient schemes proposed in [12] using the new theoretical construction of the previous point; even in this case, starting from the scheme in [12], we have re-elaborate it using the definitions of our new Outsourcing scheme.

3.1 The Outsourcing scheme

An Outsourcing scheme is a tuple of algorithms (KeyGen, Encode, Compute, Decode) satisfying five properties: *encoding correctness, computing correctness, succinctness, security* and *privacy*.

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{enck}, \mathsf{compk}, \mathsf{deck})$**:** given the security parameter λ, the key generation algorithm outputs an *encoding key* enck, a *computing key* compk and a *decoding key* deck.

$\mathsf{Encode}(\mathsf{enck}, \mathsf{L}, m) \to \sigma_m$**:** given the encoding key enck, a multi-label $\mathsf{L} = (\Delta, \tau)$ and a message $m \in \mathsf{M}$, it outputs an encoding σ_m which is given to the server to compute with.

$\mathsf{Compute}(\mathsf{compk}, f, \boldsymbol{\sigma}) \to \sigma_y$**:** on input the computing key compk, a function $f : \mathcal{M}^n \to \mathcal{M}$ and a vector of encodings $\boldsymbol{\sigma} = (\sigma_{m_1}, ..., \sigma_{m_n})$, the server computes an encoded version, σ_y, of the function's output $y = f(m_1, ..., m_n)$.

$\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta, \sigma_y) \to (acc, y)$**:** given the decoding key deck, a multi-labeled program $\mathcal{P}_\Delta = ((f, \tau_1, ..., \tau_n), \Delta)$, and the tag σ_y, the decoding algorithm converts the server's output into a bit acc and a message y. If $acc = 1$ we say the client accepts $y = \mathcal{P}_\Delta(m_1, ..., m_t)$, if $acc = 0$ we say the client rejects.

A discussion on the Outsourcing scheme model

At a glance, this new model could appear just as an improved copy of the Homomorphic MACs for Multi-Labeled Programs (Sec. 2.3.1.2) with the addition of the privacy property (shown below) and the modification of the algorithms names. Actually, it shows very interesting characteristics that maybe could not seem clear at this point, but that will be clearer in the next chapters, especially when it will be presented the framework to implement such model and some related applications.

How was this scheme born? In this chapter we have seen two possible solutions to solve the integrity and privacy problems in the scenario in which one wants to outsource some data on a remote server and then delegate a computation over such data. The idea has been to create a model that incorporated all such solutions giving the possibility to choose from time to time the desired property (integrity, privacy or both). Namely, if we need integrity and we have a VC scheme based on the definition of *Homomorphic MACs for Multi-Labeled Programs* (Sec. 2.3.1.2) we can use the Outsourcing scheme to model such VC scheme; if we need privacy and we have a homomorphic encryption scheme, we can use the Outsourcing scheme to model such encryption scheme; if finally, we need privacy and integrity, we can again use the Outsourcing and we will see how.

This idea is strictly related to the choice of the algorithms names. In the multi-labeled model(Sec. 2.3.1.2), they were KeyGen, Auth standing for *authentication*, Eval standing for *evaluation* and Ver standing for *verification*.

If one wants to be coherent with the idea presented above, it is obvious that these names are no longer correct because when one speaks about encryption, usually one refers to encryption or decryption algorithms, and not to authentication or

verification algorithms. Of course it is difficult to find names whose meanings well map for both encryption and verifiable computation; for this reason they have been chosen generic names somehow *wrappers*. We have chosen the names of Encode, $\mathsf{Compute}$ and Decode; essentially, the therm *encode* represents the idea to "convert" a message (or a generic information) into something that could be, for example, a ciphertext if one is working with encryption, or maybe an authentication tag if one is working with verifiable computation using homomorphic MACs; in other words, an *encoding* operation returns a piece of information that is what is given to the untrusted cloud. The same consideration is valid for the therm *decode* that represents the operation to restore the original value of something; operation that can be a simple decryption or a verification, according to the application context. Regarding the compute word, it has been chosen because, apart from the context, this algorithm implies always a computation, whatsoever nature it is. Finally, the KeyGen algorithm is always present and the name is appropriate for each setting.

The main properties defined in Sec. 2.3.1.2 are here redefined, while the new *privacy* property is introduced, again *in the presence of encoding and "decoding" queries* by the adversary.

Encoding Correctness. Informally speaking, an $\mathsf{Outsourcing}$ scheme has encoding correctness if any tag σ_{m_i} generated by the algorithm $\mathsf{Encode}(\mathsf{enck}, \mathsf{L}_i, m_i)$ encodes m_i with respect to the identity multi-labeled program $\mathcal{I}_{\mathsf{L}_i}$. More formally, we say that an $\mathsf{Outsourcing}$ scheme satisfies encoding correctness if for any message $m \in \mathcal{M}$, all keys $(\mathsf{enck}, \mathsf{compk}, \mathsf{deck}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$, any multi-label $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^*)^2$, and any tag $\sigma_m \leftarrow_{\mathcal{R}} \mathsf{Encode}(\mathsf{enck}, \mathsf{L}, m)$, we have that $\mathsf{Decode}(\mathsf{deck}, \mathcal{I}_{\mathsf{L}}, \sigma) = (1, m)$ holds with probability 1.

Computation Correctness. This property aims at capturing that if the computation algorithm is run on a vector of encodings $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$ such that each σ_i encodes some message m_i as the output of a multi-labeled program $(\mathcal{P}_i, \Delta)$, then the encoding σ produced by $\mathsf{Compute}$ must authenticate $f(m_1, ..., m_n)$ as the output of the composed program $(f(\mathcal{P}_1, ..., \mathcal{P}_n), \Delta)$. More formally, let us fix a tuple of keys $(\mathsf{enck}, \mathsf{compk}, \mathsf{deckk}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$, a function $g : \mathcal{M}^t \to \mathcal{M}$ and any set of message/program/encoding triples $\{(m_i, \mathcal{P}_{\Delta,i}, \sigma_{y_i})\}_{i=1}^t$ such that all multi-labeled programs $\mathcal{P}_{\Delta,i} = (\mathcal{P}_i, \Delta)$ (i.e., share the same data set identifier Δ) and $\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_{\Delta,i}, \sigma_{y_i}) = (1, m_i)$. If $m^* = g(m_1, ...m_n)$, $\mathcal{P}^* = g(\mathcal{P}_1, ..., \mathcal{P}_t)$, and $\sigma_y^* = \mathsf{Compute}(\mathsf{compk}, f, (\sigma_1, ..., \sigma_t))$, then $(\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta^*, \sigma_y^*) = (1, m^*)$ holds with probability 1.

Succinctness. The size of an encoding is bounded by some fixed polynomial in the security parameter, which is independent of the number n of inputs taken by the evaluated circuit.

Security. An $\mathsf{Outsourcing}$ scheme has to satisfy the following notion of unforgeability. Let $\mathsf{Outsourcing}$ be a scheme as defined above and let $\mathcal{A}$ be an adversary. $\mathsf{Outsourcing}$ is said to be unforgeable if for every PPT adv. $\mathcal{A}$, we have $\Pr[\mathsf{OutsourcingUF-CMA}_{\mathcal{A},\mathsf{Outsourcing}}(\lambda) = 1] \leq \epsilon(\lambda)$ where $\epsilon(\lambda)$ is a negligible function. The experiment $\mathsf{OutsourcingUF-CMA}_{\mathcal{A},\mathsf{Outsourcing}}(\lambda)$ is the one defined below.

SETUP The challenger generates $(\mathsf{enck}, \mathsf{compk}, \mathsf{deck}) \leftarrow_{\mathcal{R}} \mathsf{KeyGen}(1^\lambda)$ and gives pk to $\mathcal{A}$.

ENCODING QUERIES The adversary can adaptively ask for encodings on multi-labels and messages of its choice. Given a query (L, m) where $\mathsf{L} = (\Delta, \tau)$, the challenger proceeds as follows: If (L, m) is the first query with data set identifier Δ, then the challenger initializes an empty list $T_\Delta = \emptyset$ for data set identifier Δ. If T_Δ does not contain a tuple $(\tau, \cdot)$ (i.e., the multi-label (Δ, τ) was never queried), the challenger computes $\sigma_m \leftarrow_{\mathcal{R}} \mathsf{Encode}(\mathsf{enck}, \mathsf{L}, m)$, returns σ_m to $\mathcal{A}$ and updates the list $T_\Delta \leftarrow T_\Delta \cup (\tau, m)$. If $(\tau, m) \in T_\Delta$ (i.e., the query was previously made), then the challenger replies with the same encoding generated before. If T_Δ contains a tuple (τ, m') for some message $m' \neq m$, then the challenger ignores the query.

DECODING QUERIES The adversary has access to a decoding oracle as follows: given a query $(\mathcal{P}_\Delta, \sigma)$ from $\mathcal{A}$, the challenger replies with *only* the acceptance bit of the output of $\mathsf{Decode}(\mathsf{sk}, \mathcal{P}_\Delta, \sigma)$.

FORGERY The adversary terminates the experiment by returning a forgery $(\mathcal{P}^*_{\Delta^*},\sigma^*)$ for some $\mathcal{P}^*_{\Delta^*} = (\mathcal{P}^*, \Delta^*)$ and $\mathcal{P}^* = (f^*, \tau^*_1,...,\tau^*_n)$. Notice that, equivalently, $\mathcal{A}$ can implicitly return such a tuple as a verification query $(\mathcal{P}^*_{\Delta^*}, \sigma^*)$ during the experiment.

The following steps are almost equal to those in Sec. 2.3.1.2; therefore the reader can refer to such section to review the notion of well-defined programs with respect to a list T_Δ.

To define the output of the experiment $\mathsf{OutsourcingUF-CMA}$, we said it outputs 1 if and only if $\mathsf{Decode}(\mathsf{deck},\mathcal{P}^*_{\Delta^*},\ \sigma^*) = (1, y^*) \wedge y^* \neq \mathcal{P}^*_{\Delta^*}(m_1, ..., m_n)$ and one of the following conditions holds:

- *Type 1 Forgery:* no list T_{Δ^*} was created during the game, i.e., no message m has been authenticated with respect to a data set identifier Δ^* during the experiment;
- *Type 2 Forgery:* $\mathcal{P}^*$ is well-defined w.r.t. T_{Δ^*} and $m^* \neq f^*(\{m_j\}_{(\tau_j,m_j)\in T_{\Delta^*}})$, i.e., m^* is not the correct output of the labeled program $\mathcal{P}^*$ when executed on previously authenticated messages $(m_1, \ldots, m_n)$;
- *Type 3 Forgery:* $\mathcal{P}^*$ is not well-defined w.r.t. T_{Δ^*}.

As already done in Sec. 2.3.1.2, we provide the following proposition equivalent to the Prop. 2.1 but here adapted to the new model, presenting only the statement. The proof traces that of the Prop. 2.1.

Proposition 2.2 *Let $\lambda \in \mathbb{N}$ be the security parameter, let $p > 2^\lambda$ be a prime number, and let $\{f_\lambda\}$ be a family of arithmetic circuits over $\mathbb{Z}_p$ whose degree is bounded by some polynomial $d = poly(\lambda)$. If for any adversary $\mathcal{B}$ producing a Type 2 forgery we have that* $\Pr[\mathsf{OutsourcingUF-CMA}_{\mathcal{B},\mathsf{Outsourcing}}(\lambda) = 1] \leq \epsilon$*, then for any adversary $\mathcal{A}$ producing a Type 3 forgery it holds* $\Pr[\mathsf{OutsourcingUF-CMA}_{\mathcal{A},\mathsf{Outsourcing}}(\lambda) = 1] \leq \epsilon + d/p$.

Now it is the moment to present the real new contribute of the model, i.e. the privacy property. Again, it is described through a game between an honest challenger and an adversary. We stress that this property is still valid in presence of encoding and decoding queries; basically it is an extension of the usually semantic security notion, just embedded in this new model.

Privacy is defined based on a typical indistinguishability argument that guarantees that no information about the inputs is leaked.

Intuitively, an Outsourcing scheme is private when the public outputs of the encoding algorithm Encode over two different inputs are indistinguishable (i.e., nobody can decide which encoding is the correct one for a given input).

Privacy. An Outsourcing scheme has to satisfy the following notion of privacy. Let Outsourcing be a scheme as defined above and let $\mathcal{A}$ be an adversary. Outsourcing is said to be private if for every PPT adversary $\mathcal{A}$, we have $\Pr[\mathsf{PrOutsourcing}_{\mathcal{A},\mathsf{Outsourcing}}(\lambda) = 1] \leq 1/2 + \epsilon(\lambda)$ where $\epsilon(\lambda)$ is a negligible function. The experiment $\mathsf{PrOutsourcingPR}_{\mathcal{A},\mathsf{Outsourcing}}(\lambda)$ is the one defined below.

The SETUP step, the ENCODING QUERIES, the DECODING QUERIES and the FORGERY are the same of the security experiment described before.

CHALLENGE The adversary selects two inputs m_0, m_1 of its choice. The challenger randomly selects $b \leftarrow_{\mathcal{R}} \{0,1\}$ and then gives to the adversary the encoding of the randomly selected one of the two inputs, m_b. The adversary must guess which one was encoded.

To define the output of the experiment, we say that the adversary $\mathcal{A}$ outputs b', and the experiment outputs 1 if and only if $b' = b$.

3.1.1 Outsourcing schemes with efficient decoding

The next step is to define the meaning of *efficiency* for the Outsourcing scheme.

As for the homomorphic MAC for multi-labeled programs, an Outsourcing scheme satisfies the efficiency property if it possible to decode an encoding σ against a multi-labeled program $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ in *less time than that required to compute* $\mathcal{P}$. Again, this efficiency property is possible in the *amortized sense* exploiting the concept of multi-label; indeed the decoding is more efficient if the same program $\mathcal{P}$ is executed over different data sets.

Definition 2.3 *Let* Outsourcing = *(*KeyGen*,* Encode*,* Compute*,* Decode*) be a scheme as defined previously.* Outsourcing *satisfies the efficiency property if there exist two additional algorithms (*OfflineDecode*,* OnlineDecode*) working as follows*:

$\mathsf{OfflineDecode}(\mathsf{deck}, \mathcal{P}) \rightarrow \mathsf{OnDK}_\mathcal{P}$**:** on input the decoding key deck and a labeled program $\mathcal{P} = (f, \tau_1, ..., \tau_n)$, this algorithm generates a concise online decoding key $\mathsf{OnDK}_\mathcal{P}$. We stress that this online decoding key *does not depend on any data set identifier* Δ.

$\mathsf{OnlineDecode}(\mathsf{deck}, \mathsf{OnDK}_\mathcal{P}, \Delta, \sigma_y)$**:** given the decoding key deck, an online decoding key $\mathsf{OnDK}_\mathcal{P}$, a data set identifier Δ and a tag σ_y, the efficient decoding algorithm outputs an acceptance bit acc and a message y. If $acc = 1$ we say the client accepts $y = \mathcal{P}_\Delta(m_1, ..., m_n)$, if $acc = 0$ we say the client rejects.

The above algorithms are required to satisfy the following two properties:

Correctness. Let $(\mathsf{enck}, \mathsf{compk}, \mathsf{deck}) \leftarrow_\mathcal{R} \mathsf{KeyGen}(1^\lambda)$ be honestly generated keys, and $(\mathcal{P}_\Delta\ \sigma_s)$ be any program/encoding pair with $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ such that $\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta, \sigma_s) = (1, y)$ with $y = \mathcal{P}_\Delta(m_1, ..., m_n)$. Then, for every $\mathsf{OnDK}_\mathcal{P} \leftarrow_\mathcal{R} \mathsf{OfflineDecode}(\mathsf{deck}, \mathcal{P})$, we have $\Pr[\mathsf{OnlineDecode}(\mathsf{deck}, \mathsf{OnDK}_\mathcal{P}, \Delta, \sigma_s) = 1] = 1$.

Amortized efficiency. Let $\mathcal{P}_\Delta = (\mathcal{P}, \Delta)$ be a program, let $(m_1, ..., m_n) \in \mathcal{M}^n$ be any vector of inputs, and let $t(n)$ be the time required to compute $\mathcal{P}(m_1, ..., m_n)$. If $\mathsf{OnDK}_\mathcal{P} \leftarrow_\mathcal{R} \mathsf{OfflineDecode}(\mathsf{deck}, \mathcal{P})$, then the time required for $\mathsf{OnlineDecode}(\mathsf{deck}, \mathsf{OnDK}_\mathcal{P}, \Delta, \sigma_s)$ is $O(1)$, i.e., independent of n.

As already mentioned for the homomorphic MAC for multi-labeled programs, notice that in this efficiency requirement, it is not included the time needed to compute $\mathsf{OnDK}_\mathcal{P}$. The reason is, since $\mathsf{OnDK}_\mathcal{P}$ is independent of Δ, the same $\mathsf{OnDK}_\mathcal{P}$ can be re-used in many decodings involving the same labeled program $\mathcal{P}$ but many different Δ. In this sense, the cost of computing $\mathsf{OnDK}_\mathcal{P}$ is *amortized* over many decodings of the same function on different data sets.

3.2 A generic construction of the Outsourcing scheme

The goal of this section is to define a generic scheme that, starting from a *not private* Outsourcing scheme could provide privacy.

Let suppose now that we have an Outsourcing scheme like the one described in the previous section that, however, *does not guarantee privacy* but allows to solve integrity and let assume we have also a FHE, that is a tool that allows to solve privacy. The new question is: can we combine them together in order to obtain a composition that provides both properties?

A correct and secure answer is achievable following the approach in which one runs the Outsourcing scheme on top of the FHE scheme.

Informally, in this case to "encode" m you use the Encode method of Outsourcing over the encryption of m obtained by the Enc algorithm of the FHE scheme. Now, the encoding σ_m one obtains is not revealing something about m because it is an *encoding of a ciphertext*. The cloud runs the computation algorithm Compute of Outsourcing over the FHE Eval over f, producing an encoding σ_s. Fianlly, the Decode algorithm is also run respect to the FHE Eval over the f, but it is first checked that σ_s is correct and only if this test passes then one decrypts; in some sense one will be sure that what one is decrypting is something that one obtains correctly by applying the evaluation algorithm of the FHE.

In retrospect, this is a very simple solution because it is a kind of the equivalent of what do you do in the standard so called *Authenticated Encryption* ([5]) where the rule is *Encrypt-then-MAC* instead of the opposite, while here it is *Encrypt-then-Encode* instead of the opposite.

The generic Outsourcing scheme

In this section it is described the generic solution to outsource computation over encrypted data. It is assumed the existence of a FHE scheme, and a Outsourcing scheme to outsource the computation of generic functions.

Let FHE = (FHE.ParamGen, FHE.KeyGen, FHE.Enc, FHE.Dec, FHE.Eval) be an FHE scheme as defined in Sec. 2.4.1. Also let Outsourcing = (KeyGen, Encode, Compute, Decode) be a scheme which is correct, secure, and outsourceable, as defined in Sec. 3.1. In particular note that the Outsourcing scheme does not need to be private, and that the security is guaranteed in the presence of decoding queries. It is here described a new scheme PrOutsourcing = (PrKeyGen, PrEncode, PrCompute, PrDecode) (for private Outsourcing) which uses the above two tools as follows.

PrKeyGen(1^λ) $\to$ ($\mathsf{K}_1, \mathsf{K}_2$):
- Run FHE.KeyGen(1^λ) to generate (pk, dk, evk) for FHE.
- Run KeyGen(1^λ) to generate the enck, compk, deck for Outsourcing.
- Set K_1 = (pk, evk, compk) and K_2 = (K_1, dk, enck, deck).

PrEncode(K_2, L, m) $\to \sigma_m$:
- Compute c_m = FHE.Enc(pk, m).

- Run $\mathsf{Encode}(\mathsf{enck}, \mathsf{L}, c_m)$ to get σ_m.

$\mathsf{PrCompute}(\mathsf{K}_1,\ f, \boldsymbol{\sigma}) \rightarrow (\sigma_s)$:
- Run $\mathsf{Compute}(\mathsf{compk},\ f,\ \boldsymbol{\sigma})$ to compute σ_s. Note that σ_s is an encoding of $c^* = \mathsf{FHE.Eval}(\mathsf{evk}, f, \boldsymbol{\sigma})$.

$\mathsf{PrDecode}(\mathsf{K}_2, \mathcal{P}_\Delta, \sigma_s) \rightarrow (acc, y)$:
- Run $\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta, \sigma_s)$ to get (acc, c). If $acc = 0$, reject. If $acc = 1$, decrypt $y = \mathsf{FHE.Dec}(\mathsf{dk}, c)$.

Even in this case, we provide only the statement of the following theorem equivalent to that present in [12], but here adapted to the new model.

Theorem 1. *If* FHE *is a semantically secure FHE, and* Outsourcing *is a correct, secure, and outsourceable scheme, then* PrOutsourcing *is a correct, secure, outsourceable, and "private" scheme.*

3.3 An efficient realization of the Outsourcing scheme: verifiable computation of linear functions on encrypted data

The main goals of this thesis is to implement, using the framework that we will present in the next chapter, one of the efficient schemes conceived in [12] that we re-elaborated using our new Outsourcing scheme and that we have called $\mathsf{Outsourcing}_{lin}$.

$\mathsf{Outsourcing}_{lin}$ (see Sec. 3.3.3) allows to compute multi-variate polynomials of degree 1 over the ring $\mathbb{Z}_{2^k}$. In the next sections we will focus on the redefinition of the scheme in [12] using our new Outsourcing scheme presented in 3.1. To better understanding the $\mathsf{Outsourcing}_{lin}$ scheme, before to define it, we propose a review of some essential and related mathematical notions, in order to also familiarize with the notation (see Sec. 3.3.1). Later on in the chapter, we will also show the Joye-Libert cryptoystem (see Sec. 3.3.2) that is the homomorphic encryption scheme employed in the $\mathsf{Outsourcing}_{lin}$ scheme.

3.3.1 Some facts of number theory

Quadratic Residues Modulo a Prime

In this section we recall some basic facts of number theory. Most of these definitions are taken from [17], where it is also possible to find proofs that here are

omitted. Considering a group $\mathbb{G}$, an element y of $\mathbb{G}$ is a *quadratic residue* if there exists an element x of $\mathbb{G}$ such that $x^2 = y$. An element that is not a quadratic residue is called a *quadratic non-residue*. If we consider the case of $\mathbb{G} = \mathbb{Z}_p^*$, the set of elements from $\mathbb{Z}_p$ that are relatively prime to p, y is a quadratic residue modulo p if it is congruent to a perfect square modulo p; i.e, if there exists an $x \in \mathbb{Z}_p^*$ such that $y = x^2 \bmod p$.

Proposition 3.1 *Let consider a prime $p > 2$. Every quadratic residue in $\mathbb{Z}_p^*$ has exactly two square roots.*

The above proposition implies that *half the elements of $\mathbb{Z}_p^*$ are quadratic residues.* The set of quadratic residues modulo p is here denoted by $\mathcal{QR}_p$, while the set of quadratic non-residue is here denoted by $\mathcal{QNR}_p$. We can also say that, for $p > 2$ prime

$$|\mathcal{QR}_p| = |\mathcal{QNR}_p| = |\mathbb{Z}_p^*|/2 = (p-1)/2$$

We recall now the *Jacobi symbol of x modulo p, $\mathcal{J}_p(x)$.*

The Jacobi symbol is a generalization of the *Legendre symbol.* For any integer a and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k} \text{ where } n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}$$

$\left(\frac{a}{p}\right)$ represents the Legendre symbol, defined for all integers a and all odd primes p by

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p} \\ +1 & \text{if } a \not\equiv 0 \pmod{p} \text{ and for some integer } x, a \equiv x^2 \pmod{p} \\ -1 & \text{if there is no such } x \end{cases}$$

The Legendre and Jacobi symbols are indistinguishable exactly when the lower argument is an odd prime, in which case they have the same value.

Let consider a prime $p > 2$, and $x \in \mathbb{Z}_p^*$. Then

$$\mathcal{J}_p(x) = \begin{cases} +1 & \text{if } x \text{ is a quadratic residue modulo } p \\ -1 & \text{if } x \text{ is } \textit{not} \text{ a quadratic residue modulo } p \end{cases}$$

The notation of Jacobi symbol can be extended for any x relatively prime to p by defining $\mathcal{J}_p(x) = \mathcal{J}_p([x \bmod p\ ])$.

Proposition 3.2 *Let consider a prime* $p > 2$*. Then* $\mathcal{J}_p(x) = x^{\frac{p-1}{2}}$.

From the previous proposition we can easily obtain a polynomial-time algorithm that tells us when a given element $x \in \mathbb{Z}_p^*$ is a quadratic residue and when not.

Algorithm 3.1 ***Deciding quadratic residuosity modulo a prime***

Input Prime p; element $x \in \mathbb{Z}_p^*$
Output $\mathcal{J}_p(x)$

$z := [x^{\frac{p-1}{2}} \bmod p]$;
if $z = 1$ **return** x *is a quadratic residue*
else return x *is a quadratic non-residue*

Quadratic Residues Modulo a Composite

Let analyse now the group $\mathbb{Z}_N^*$. We can characterize the quadratic residues modulo N using the results of the previous section together with the Chinese remainder theorem: it says that $\mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ and it guarantees a correspondence $y \leftrightarrow (y_p, y_q)$, i.e. $y_p = [y \bmod p]$ and $y_q = [y \bmod q]$.

Proposition 3.3 *Let consider* $N = pq$ *with* p, q *distinct primes, and let* $y \in \mathbb{Z}_N^*$ *with* $y \leftrightarrow (y_p, y_q)$*. Then* y *is a quadratic residue modulo* N *if and only if* y_p *is a quadratic residue modulo* p *and* y_q *is a quadratic residue modulo* q*.*

By using similar observations made for the quadratic residues modulo p, here one can easily seen that exactly $1/4$ of the elements of $\mathbb{Z}_N^*$ are quadratic residues. Note that since $y \in \mathbb{Z}_N^*$ is a quadratic residue if and only if y_p, y_q are quadratic residues, there is a one-to-one correspondence between $\mathcal{QR}_N$ and $\mathcal{QR}_p \times \mathcal{QR}_q$. Thus, the fraction of quadratic residues modulo N is

$$\frac{|\mathcal{QR}_N|}{|\mathbb{Z}_N^*|} = \frac{|\mathcal{QR}_p| \cdot |\mathcal{QR}_q|}{|\mathbb{Z}_N^*|} = \frac{\frac{p-1}{2} \cdot \frac{q-1}{2}}{(p-1)\,(q-1)} = 1/4$$

In the previous section, it has been defined the Jacobi symbol $\mathcal{J}_p(x)$ for $p > 2$ prime. We can also define it for the case of $N = pq$, a product of distinct odd primes in the following way: for any x relatively prime to N,

$$\mathcal{J}_N(x) = \mathcal{J}_p(x) \cdot \mathcal{J}_q(x) = \mathcal{J}_p([x \bmod p]) \cdot \mathcal{J}_q([x \bmod q\,]):$$

We refer to $\mathcal{J}_N^{+1}$ as the set of elements in $\mathbb{Z}_N^*$ having Jacobi symbol $+1$, and to $\mathcal{J}_N^{-1}$ as the set of elements in $\mathbb{Z}_N^*$ having Jacobi symbol -1.

Hence, we can also assert that *if x is a quadratic residue modulo N, then $\mathcal{J}_N(x) = +1$*. If x is a quadratic residue modulo N, then $[x \bmod p]$ and $[x \bmod q]$ are quadratic residues modulo p and q, respectively; this means that $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$. So $\mathcal{J}_N(x) = +1$.

However, one has to note that $\mathcal{J}_N(x) = +1$ it is also possible when $\mathcal{J}_p(x) = \mathcal{J}_q(x) = -1$; that is, when both $[x \bmod p]$ and $[x \bmod q]$ are *not* quadratic residues modulo p and q, and so x is not a quadratic residue modulo N.

Let introduce therefore the notation $\mathcal{QNR}_N^{+1}$ for this set of elements:

$$\mathcal{QNR}_N^{+1} = \{x \in \mathbb{Z}_N^* | \mathcal{J}_N(x) = +1, \text{ but } x \text{ is not a quadratic residue modulo } N\}$$

Proposition 3.4 *Let $N = pq$ with p, q distinct, odd primes. Then:*

- Exactly half the elements of $\mathbb{Z}_N^*$ are in $\mathcal{J}_N^{+1}$;
- $\mathcal{QR}_N$ is contained in $\mathcal{J}_N^{+1}$;
- Exactly half the elements of $\mathcal{J}_N^{+1}$ are in $\mathcal{QR}_N$ (the other half are in $\mathcal{QNR}_N^{+1}$).

$\mathcal{J}_N^{+1}$		$\mathcal{J}_N^{-1}$
$\mathcal{QR}_N$	$\mathcal{QNR}_N^{+1}$	$\mathcal{QNR}_N^{-1}$

Table 3.1: Distribution of the quadratic residues modulo N in $\mathbb{Z}_N^*$.

Quadratic Residuosity Assumption

We can now extend the Alg. 3.1 of the previous section to work modulo a composite number N but *only if the factorization of N is known.*

Algorithm 3.2 ***Deciding quadratic residuosity modulo a composite***

Input Composite $N = pq$; the factors p and q; $x \in \mathbb{Z}_N^*$
Output $\mathcal{J}_N(x)$

$z_1 := \mathcal{J}_p(x) = [x^{\frac{p-1}{2}} \bmod p]$;
$z_2 := \mathcal{J}_q(x) = [x^{\frac{q-1}{2}} \bmod q]$;
if $z_1 = z_2 = +1$ **return** '*x is a quadratic residue*
else return *x is a quadratic non-residue*

When the factorization of N is unknown, however, there is no immediate way of efficiently computing the Jacobi symbol modulo N, or efficiently deciding whether a given element x is a quadratic residue modulo N or not. Somewhat surprisingly, a highly non-trivial polynomial-time algorithm is known for computing $\mathcal{J}_N(x)$ without the factorization of N. This leads to a partial test of quadratic residuosity: if, for a given input x it holds that $\mathcal{J}_N(x) = -1$, then x cannot possibly be a quadratic residue. This test says *nothing* in case $\mathcal{J}_N(x) = -1$, and it is a reasonable cryptographic assumption that no polynomial-time algorithm for deciding quadratic residuosity in this case (that performs better than random guessing) exists.

We now formalize this assumption. Let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$, and p and q are n-bit primes except with probability negligible in n.

Definition 3.1 *We say "deciding quadratic residuosity is hard relative to* GenModulus*" if for all probabilistic, polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl such that*

$$|\Pr[\mathcal{A}(N, \mathsf{qr}) = 1] - \Pr[\mathcal{A}(N, \mathsf{nqr}) = 1]| \leq \mathsf{negl}(\mathrm{n}),$$

where in each case the probabilities are taken over the experiment in which GenModulus*(1^n) is run to give (N,p,q),* qr *is chosen at random from $\mathcal{QR}_N$ and* qnr *is chosen at random from $\mathcal{QNR}_N^{+1}$.*

The quadratic residuosity assumption is simply the assumption that there exists a GenModulus relative to which deciding quadratic residuosity is hard. It is easy to see that if deciding quadratic residuosity is hard relative to GenModulus, then factoring is hard relative to GenModulus as well.

The Goldwasser-Micali (GM) Encryption Scheme is a public encryption scheme for single bit-messages based on this *Quadratic Residuosity* assumption, i.e. *the hardness of distinguishing quadratic residues from (certain) quadratic non-residues modulo a composite*. The GM cryptosystem has *homomorphic properties*, in the sense that if c_0, c_1 are the encryptions of bits m_0, m_1, then $c_0c_1 \bmod N$ will be an encryption of $m_0 \oplus m_1$

$$c_0 \cdot c_1 = E(m_0) \cdot E(m_1) = E(m_0 \oplus m_1)$$

where $\oplus$ denotes addition mod 2, (i.e. exclusive-or).

3.3.2 The Joye-Libert cryptosystem

At the EUROCRYPT 2013 conference, Marc Joye and Benoît Libert presented a paper with the title "*Efficient Cryptosystems From 2^k-th Power Residue Symbols*",

[18], to show a generalization of the Goldwasser-Micali cryptosystem, using 2^k-th power residues.

The new cryptosystem is efficient in both bandwidth and speed. Further they inherit the useful features of the GM cryptosystem (like its homomorphic property) and they are shown to be secure under a similar complexity assumption.

n^{th}- power residues

First we recall some useful definitions from [18].

Joye and Libert extend the quadratic residue notion defining the n-th power residue symbol.

Let $N \in \mathbb{N}$. For each integer $n \geq 2$, they define $(\mathbb{Z}_N^*)^n = \{x^n | x \in \mathbb{Z}_N^*\}$ the set of *n^{th}-power residues modulo N*. If the relation $a = x^n$ has no solution in $\mathbb{Z}_N^*$ then a is called a *n^{th}-power non-residue modulo N*. Suppose that p is an odd prime. For any integer a with $\gcd(a, p) = 1$, it is easily verified that a is a nth-power residue modulo p if and only if

$$a^{\frac{p-1}{gcd(n,p-1)}} \equiv 1 \pmod{p}$$

There are several ways to generalize the Legendre symbol defined in Sec. 3.3.1. Joye and Libert considered the n-th powein an his r residue symbol for a divisor n of $(p-1)$.

Definition 3.2 *Let p be an odd prime and let $n \geq 2$ such that $n | p-1$. Then the symbol*

$$\left(\frac{a}{p}\right)_n = a^{\frac{p-1}{n}} mods\ p$$

is called the n-th power residue symbol modulo p.

Quadratic Residuosity

To emphasize that the Quadratic Residuosity assumption should hold for moduli $N = pq$ with $p, q \equiv 1 \pmod{2^k}$ Joye and Libert will refer to it as the k-QR assumption. Formally:

Definition 3.3 (Quadratic Residuosity Assumption) *Let* RSAGen *be a probabilistic algorithm which, given a security parameter λ, outputs primes p and q such that $p, q \equiv 1$ (mod 2^k), and their product $N = pq$. The* QuadraticResiduosity *($k-$QR) assumption asserts that the function $\boldsymbol{Adv}_{\mathcal{D}}^{k\text{-QR}}(1^\lambda)$, defined as the distance*

$$\left| \Pr[\mathcal{D}(x, N) = 1 | x \leftarrow_{\mathcal{R}} \mathbb{QR}_N] - \Pr[\mathcal{D}(x, N) = 1 | x \leftarrow_{\mathcal{R}} \mathbb{J}_N \setminus \mathbb{QR}_N] \right|$$

is negligible for any probabilistic polynomial-time distinguisher $\mathcal{D}$; the probabilities are taken over the experiment of running $(N, p, q) \leftarrow \mathsf{RSAGen}(1^\lambda)$ and choosing at random $x \in \mathbb{QR}_N$ and $x \in \mathbb{J}_N \setminus \mathbb{QR}_N$.

They also introduced a new assumption for RSA moduli $N = pq$ when $p, q \equiv 1 \pmod 4$. Since -1 is a square modulo p and q, the square roots of any element of $\mathbb{QR}_N$ all have the same Jacobi symbol modulo N. The new assumption, which they called the *Squared Jacobi Symbol* (SJS) assumption, posits the infeasibility of determining whether $\left(\frac{y}{N}\right) = 1$ or -1 given (x, N) where $x = y^2 \bmod N$. Formally:

Definition 3.4 (Squared Jacobi Symbol Assumption) *Let* RSAGen *be a probabilistic algorithm which, given a security parameter λ, outputs primes p and q such that $p, q \equiv 1 \pmod{2^k}$, and their product $N = pq$. The Squared Jacobi Symbol ($k-$SJS) assumption asserts that the function $\boldsymbol{Adv}_{\mathcal{D}}^{k-\mathsf{SJS}}(1^\lambda)$, defined as the distance*

$$\left|\Pr[\mathcal{D}(y^2 \ mod\ N, N) = 1 | y \leftarrow_{\mathcal{R}} \mathbb{J}_N] - \Pr[\mathcal{D}(y^2 \ mod\ N, N) = 1 | y \leftarrow_{\mathcal{R}} \bar{\mathbb{J}}_N\right|$$

is negligible for any probabilistic polynomial-time distinguisher D*; the probabilities are taken over the experiment of running $(N, p, q) \leftarrow \mathsf{RSAGen}(1^\lambda)$ and choosing at random $y \in \mathbb{J}_N$ and $y \in \bar{\mathbb{J}}_N$.*

The setting of the Joye-Libert cryptosystem is basically the same as for the Goldwasser-Micali cryptosystem. The only additional requirement is that primes p and q are chosen congruent to 1 modulo 2^k where k denotes the bit-size of the message being encrypted.

The scheme

In detail, the Joye-Libert (JL) Encryption Scheme is the tuple (KeyGen, Encrypt, Eval, Decrypt) defined as follows.

$\mathsf{KeyGen}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{evk}, \mathsf{dk})$**:** it takes a security parameter λ and randomly produce primes p, $q \equiv 1 (\mathrm{mod}\ 2^k\ )$, setting $N = pq$ and picking $y \in \mathbb{J}_N \setminus \mathbb{QR}_N$. The output is the tuple composed of the secret decryption key $\mathsf{dk} = \{p\}$, the public encryption key $\mathsf{pk} = \{N, y, k\}$ and the public evaluation key $\mathsf{evk} = \{N, y, k\}$.

$\mathsf{Encrypt}(\mathsf{pk}, m) \rightarrow c$**:** let consider an element $m \in \mathcal{M}$ to encrypt (seen as an integer in $\{0, ..., 2^k - 1\}$) where $\mathcal{M} = \{0, 1\}^k$. The algorithm generates a random $x \in \mathbb{Z}_N^*$ and outputs the ciphertext $c = y^m x^{2^k} \bmod N$.

$\mathsf{Eval}(\mathsf{evk}, g, c_1, ..., c_t) \rightarrow c^*$**:** given the public evaluation key, a circuit $g : \mathcal{M}^t \rightarrow \mathcal{M}$, and a set of t ciphertexts $c_1, ..., c_t$, the algorithm deterministically compute and output a ciphertext c^*. Since that g is a circuit that can only manage additions and multiplications by constants, every g function can be computed

using the two elementary functions addition and multiplication by constant: $\mathsf{Add}(c_1, c_2) \rightarrow c$, where c_1 and c_2 are two ciphertexts and $c = c_1 * c_2 \bmod N$, and $\mathsf{CMult}(c_3, a) \rightarrow c'$ where c_3 is a ciphertext, a is a constant element in $\mathcal{M} = \{0,1\}^k$ and $c' = c_3^a \bmod N$.

$\mathsf{Decrypt}(\mathsf{dk}, c) \rightarrow m$**:** let $c \in \mathbb{Z}_N^*$ a ciphertext and $\mathsf{dk} = \{p\}$ the private key; the algorithm first computes $z = \left(\frac{c}{p}\right)_{2^k}$ and then finds $m \in \{0, ..., 2^k - 1\}$ such that the relation

$$\left[\left(\frac{y}{p}\right)_{2^k}\right]^m = z \text{ (mods } p)$$

holds.

The case $k = 1$ of the JL cryptosystem corresponds to the Goldwasser-Micali cryptosystem which has indistinguishable encryptions under the standard Quadratic Residuosity assumption. For $k \geq 2$ the scheme provides *indistinguishable encryptions under the* $k - \mathsf{QR}$ *and* $k - \mathsf{SJS}$ *assumptions*. We do not provide here the proof and the discussion about security analysis; the reader can refer to [18] for more details.

The GM cryptosystem is computationally efficient but somewhat wasteful in bandwidth as $k \cdot log_2 N$ bits are needed to encrypt a k-bit message. The JL scheme is instead more bandwidth-efficient; only $log_2 N$ bits are needed for encrypting a k-bit message in typical applications.

Similarly to the GM cryptosystem, the JL cryptosystem enjoys the additive homomorphic encryption property, i.e. if c_1 and c_2 denote two ciphertexts corresponding to k-bit plaintexts m_1 an m_2, respectively, then $c_1 \cdot c_2 \pmod N$ is an encryption of the message $m_1 + m_2 \pmod{2^k}$.

As last comment, we highlight that, for security reasons, is necessary that $\frac{1}{4} log_2 N - k_1 > k_2$, or equivalently, $k_1 < \frac{1}{4} log_2 N - k_2$ where k_1 is the bit-size of the message space and k_2 is the security parameter.

In the rest of the thesis, we will refer to the above formula as the *security check* of the Joye-Libert cryptosystem.

Decryption algorithms

One of the main advantages of the proposed cryptosystem is that it provides an efficient (and practical) way to recover the message, even for large values of k. Fig. 3.1 shows the original decryption algorithm proposed by Joye and Libert.

Algorithm 3.3 ***Decryption algorithm (1)***

Input Ciphertext c, private key p (and public-key elements y and k)
Output Plaintext $m = (m_{k-1}, ..., m_0)_2$

$m \leftarrow 0; B \leftarrow 1$
for $i = 1$ to k **do**
  $z \leftarrow \left(\frac{c}{p}\right)_{2^i}; t \leftarrow \left(\frac{y}{p}\right)_{2^i}^m \text{ mods } p$
  if $(t \neq z)$ **then** $m \leftarrow m + B$
  $B \leftarrow 2B$
end for
return m

Figure 3.1: Original decryption algorithm of Joye and Libert.

The message $m \in \{0,1\}^k$ is viewed as a k-bit integer given by its binary expansion $m = \sum_{i=1}^{k-1} m_i 2^i$, with $m_i \in \{0,1\}$. Given $c = y^m x^{2^k} \bmod N$, we have

$$\left(\frac{c}{p}\right)_{2^i} = \left(\frac{y^{\sum_{j=0}^{i-1} m_j 2^j}}{p}\right)_{2^i} = \left(\frac{y}{p}\right)_{2^i}^{\sum_{j=0}^{i-1} m_j 2^j} (\text{mods } p)$$

since $y^m x^{2^k} = y^{\sum_{j=0}^{i-1} m_j 2^j} \cdot (y^{\sum_{j=1}^{k-1} m_j 2^{j-1}} x^{2^{k-1}})^{2^i}$, for $1 \leq i \leq k$. As a result, m can be recovered bit by bit using p, starting from the rightmost bit. The algorithm uses an accumulator B which contains the successive powers of 2.

Joye-Libert optimized decryption algorithms

Actually, the part of the JL cryptosystem concerning the decryption algorithms has been subject of special interest. Indeed, in a full version of the paper appeared later respect to the original version of the Joye-Libert paper, the authors added a section containing some optimized decryption algorithms.

First modified decryption algorithm Let $N = pq$ where p and q are prime, and $p, q \equiv 1(\mod 2^k)$ but $p, q \not\equiv 1(\mod 2^{k+1})$. In this case, we can write $p = 2^k p' + 1$ and $q = 2^k q' + 1$ for some odd integers p' and q'. therefore as $p = 2^k p' + 1$ with p' odd and since $y \in \mathbb{J}_N \setminus \mathbb{QR}_N$, it is easily seen that

$$y^{2^{k-1} p'} \equiv -1 \pmod p$$

The proof of this statement is straightforward since we have $y^{2^{k-1}p'} \equiv y^{(p-1)/2} \equiv \frac{y}{p} \equiv -1 \pmod p$. Consider now the ciphertext $c = y^m x^{2^k} \mod N$ of messages $m = \sum_{i=0}^{k-1} m_i 2^i$ with $m_i \in \{0,1\}$. If, for $1 \le j \le k$, we define

$$\lambda_j = 2^{k-j} p'$$

and

$$C_j = c^{\lambda_j} \mod p,$$

then, we have

$$C_j \equiv (y^m x^{2^k})^{\lambda_j} \equiv y^{m 2^{k-j} p'} \equiv y^{(m \mod 2^j) 2^{k-j} p'} \equiv y^{(m_{j-1} 2^{j-1} + \sum_{i=0}^{j-2} m_i 2^i) 2^{k-j} p'}$$
$$\equiv y^{m_{j-1} 2^{k-1} p'} y^{(m \mod 2^{j-1}) 2^{k-j} p'} \equiv (-1)^{m_{j-1}} y^{(m \mod 2^{j-1}) \lambda_j} \pmod p$$

Hence, it follows that

$$\left(\frac{c}{y^{m \bmod 2^{j-1}}}\right)_j^{\lambda} \equiv (-1)^{m_{j-1}} \mod p$$

Starting from these formulas, they derived the first new algorithm with a private key that now consists of the pair (p', D) where $D \equiv y^{-p'} \mod p$. Fig. 3.2 shows the first modified decryption algorithm of Joye and Libert.

Algorithm 3.4 ***Decryption algorithm (2)***

Input Ciphertext c, private key (p', D) (and public-key elements y and k)
Output Plaintext $m = (m_{k-1}, ..., m_0)_2$

$m \leftarrow 0$; $\mathsf{B} \leftarrow 1$; $\mathsf{D} \leftarrow D$
$\mathsf{C} = c^{p'} \mod p$
for $j = 1$ to $k - 1$ **do**
  $z \leftarrow \mathsf{C}^{2^{k-j}} \mod p$;
  if $(z \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$; $\mathsf{C} \leftarrow \mathsf{C} \cdot \mathsf{D} \mod p$
  $\mathsf{B} \leftarrow 2\mathsf{B}$; $\mathsf{D} \leftarrow \mathsf{D}^2 \mod p$
end for
if $(\mathsf{C} \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$
return m

Figure 3.2: First modified decryption algorithm of Joye and Libert.

Second modified decryption algorithm The second proposed algorithm is just a change of the previous, where D is precomputed instead of being explicitly included in the private key. Paying something more in terms of storage, we obtain better performances. Now, the decryption key is defined by the tuple $(p',D[1],...,D[k-1])$ where $D[j] = D^{2^{j-1}} \bmod p$ for $1 \leq j \leq k-1$. Fig. 3.3 shows the second modified algorithm of Joye and Libert.

Algorithm 3.5 ***Decryption algorithm (3)***

Input Ciphertext c, private key $(p', D[1], ...D[k-1])$ (and public-key elements y and k)
Output Plaintext $m = (m_{k-1}, ..., m_0)_2$

$m \leftarrow 0; \mathsf{B} \leftarrow 1$
for $j = 1$ to $k-1$ **do** $\mathsf{D}[j] \leftarrow D[j]$
$\mathsf{C} = c^{p'} \bmod p$
for $j = 1$ to $k-1$ **do**
  $z \leftarrow \mathsf{C}^{2^{k-j}} \bmod p$;
  if $(z \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$; $\mathsf{C} \leftarrow \mathsf{C} \cdot \mathsf{D}[j] \bmod p$
  $\mathsf{B} \leftarrow 2\mathsf{B}$
end for
if $(\mathsf{C} \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$
return m

Figure 3.3: Second modified decryption algorithm of Joye and Libert.

Our new decryption algorithms

Actually, Joye and Libert proposed four optimized decryption algorithms. By analysing such new algorithms during the study for this thesis and by trying to implement them, we discovered that, among the four new algorithms two were incorrect. After an exchange of e-mails, the authors confirmed my observation. From that moment I started an investigation, in collaboration with Dario Fiore in an attempt to discover new algorithms. This phase led to the definition of two new algorithms, one of them performing even better of the *correct* improved algorithms presented by JL.

In the previous part I have described only the two *correct* algorithms of JL, while now we are going to start the description of our two new decryption algorithms (Fig. 3.4 and Fig. 3.5). The decryption key is still defined by the tuple $(p',D[1],...,D[k-1])$ where $D[j] = D^{2^{j-1}} \bmod p$ for $1 \leq j \leq k-1$.

Algorithm 3.6 ***New decryption algorithm (5)***

Input Ciphertext c, private key $(p', D[1], ...D[k-1])$ (and public-key elements y and k)
Output Plaintext $m = (m_{k-1}, ..., m_0)_2$

$m \leftarrow 0; \mathsf{B} \leftarrow 1$
for $j = 1$ to $k-1$ **do** $\mathsf{D}[j] \leftarrow D[j]$
$\mathsf{C}[0] = c^{p'} \bmod p$
for $j = 1$ to $k-1$ **do** $\mathsf{C}[j] \leftarrow \mathsf{C}[j-1]^2 \bmod p$
for $j = 1$ to $k-1$ **do**
  if $(\mathsf{C}[k-j] \neq 1)$ **then**
    $m \leftarrow m + \mathsf{B}$
    for $i = k-1$ to $j+1$ **do** $\mathsf{C}[k-i] \leftarrow \mathsf{C}[k-i-1]^2 \bmod p$
  end if
  $\mathsf{B} \leftarrow 2\mathsf{B}$
end for
if $(\mathsf{C}[0] \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$
return m

Figure 3.4: Our first new decryption algorithm.

Algorithm 3.7 ***New decryption algorithm (4)***

Input Ciphertext c, private key $(p', D[1], ...D[k-1])$ (and public-key elements y and k)
Output Plaintext $m = (m_{k-1}, ..., m_0)_2$

$m \leftarrow 0; \mathsf{B} \leftarrow 1$
for $j = 1$ to $k-1$ **do** $\mathsf{D}[j] \leftarrow D[j]$
$\mathsf{C}[0] = c^{p'} \bmod p$
for $j = 1$ to $k-1$ **do** $\mathsf{C}[j] \leftarrow \mathsf{C}[j-1]^2 \bmod p$
for $j = 1$ to $k-1$ **do**
  if $(\mathsf{C}[k-j] \neq 1)$ **then**
    $m \leftarrow m + \mathsf{B}$
    for $i = j+1$ to k **do** $\mathsf{C}[k-i] \leftarrow \mathsf{C}[k-i] \cdot \mathsf{D}[k-i+j] \bmod p$
  end if
  $\mathsf{B} \leftarrow 2\mathsf{B}$
end for
if $(\mathsf{C}[0] \neq 1)$ **then** $m \leftarrow m + \mathsf{B}$
return m

Figure 3.5: Our second new decryption algorithm.

The Alg. 3.7 is the real new contribution. As we will see in the Chapter 6, it provides significantly better performances than those provided by the algorithms of Joye and Libert.

3.3.3 The $\mathsf{Outsourcing}_{lin}$ scheme for computing linear functions over the ring $\mathbb{Z}_{2^k}$

We are almost ready to *redefine* the central scheme of this work. The basic idea of the scheme is to combine the Joye-Libert linearly-homomorphic encryption scheme with a linearly homomorphic MAC with efficient verification. The authors in [12] designed a MAC that allows to authenticate *linear computations* over the group $\mathbb{Z}_N^*$ which is the ciphertext space of the encryption scheme of Joye-Libert.

In particular, one of the more interesting tools to achieve efficiency is represented by a new pseudorandom function (PRF) with amortized closed-form efficiency whose security relied on the Decisional Diffie-Hellman (DDH) assumption in the subgroup of 2^k-residues of $\mathbb{Z}_N^*$.

Amortized closed-form efficient pseudorandom function

[1] A PRF consists of two algorithms ($\mathsf{F.KG}$, F) such that the key generation $\mathsf{F.KG}$ takes as input the security parameter 1^λ and outputs a secret key K and some public parameters pp that specify domain $\mathcal{X}$ and range $\mathcal{R}$ of the function, and the function $\mathsf{F}_K(x)$ takes input $x \in \mathcal{X}$ and uses the secret key K to compute a value $R \in \mathcal{R}$. As usual, a PRF must satisfy the pseudorandomness property. Namely, we say that ($\mathsf{F.KG}$, F) is *secure* if for every PPT adversary $\mathcal{A}$ we have that:

$$\left| \Pr[\mathcal{A}^{\mathsf{F}_\mathsf{K}(\cdot)}(1^\lambda, \mathsf{pp}) = 1] - \Pr[\mathcal{A}^{\Phi(\cdot)}(1^\lambda, \mathsf{pp}) = 1] \right| \leq \epsilon(\lambda)$$

where $\epsilon(\lambda)$ is negligible, $(K, \mathsf{pp}) \leftarrow \mathsf{F.KG}(1^\lambda)$, and $\Phi : \mathcal{X} \to \mathcal{R}$ is a random function.

In other words, it is not required for the element $\mathsf{F}_\mathsf{K}(\cdot)$ to be indistinguishable from the random function $\Phi(\cdot)$; rather, the *behaviour* of any PPT adversary $\mathcal{A}$ which is given oracle access to the function $\mathsf{F}_\mathsf{K}(\cdot)$ must be indistinguishable from the behaviour of the same adversary when given oracle access to the function $\Phi(\cdot)$.

To put this yet in another way we can imagine to have two worlds: in the first world the adversary queries the function $\mathsf{F}_\mathsf{K}(\cdot)$ while in the second world the adversary's queries are answered by the truly random function $\Phi(\cdot)$. Here by *truly*

[1] Much of this section has been copied verbatim from [12].

random function we mean a black box which outputs a fresh random value on each invocation, except that it is *consistent*, i.e. if queried twice on the same value, it always returns the same output. Now we say that the security property is guaranteed if, although the outputs given by $\mathsf{F}_\mathsf{K}(\cdot)$ are clearly correlated while $\Phi(\cdot)$'s answers are completely independent, the behaviour of the adversary is essentially the same, so that it is not possible to tell these two worlds apart.[2].

The following definition of pseudorandom functions with amortized closed-form efficiency is taken from [3], which extend closed-form-efficient PRFs in [4].

Definition 3.5 (Amortized Closed-Form Efficiency [3]) *Consider a computation* Comp *that takes as input* n *random values* $R_1, ..., R_n \in R$ *and a vector of* m *arbitrary values* $z = (z_1, ..., z_m)$, *and assume that the computation of* $\mathsf{Comp}(R_1, ..., R_n, z_1, ..., z_m)$ *requires time* $t(n, m)$. *Let* $\mathsf{L} = (\mathsf{L}_1, ..., \mathsf{L}_n)$ *be arbitrary values in the domain* $\mathcal{X}$ *of* F *such that each can be interpreted as* $\mathsf{L}_i = (\Delta, \tau_i)$. *We say that a PRF* $(\mathsf{F.KG}, \mathsf{F})$ *satisfies* amortized closed-form efficiency *for* $(\mathsf{Comp},\ \mathsf{L})$ *if there exist algorithms* $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\tau}$ *and* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}$ *such that:*

1. *Given* $\omega \leftarrow \mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\tau}(K, z)$, *we have that*
$$\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \omega) = \mathsf{Comp}(\mathsf{F}_K(\Delta, \tau_1), ..., \mathsf{F}_K(\Delta, \tau_n), z_1, ..., z_m);$$
2. *the running time of* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \omega)$ *is* $o(t)$.

A Realization Based on DDH over 2^k-Residues in $\mathbb{Z}_N^*$

Here it is described a closed-form efficient PRF based on the DDH assumption in the subgroup of 2^k-residues of $\mathbb{Z}_N^*$.

Let $f : \mathbb{Z}_{2^k}^t \to \mathbb{Z}_{2^k}$ be a linear function $f(Z_1, ..., Z_t) = \sum_{i=1}^t f_i \cdot Z_i$. We define $\hat{f} : (\mathbb{Z}_N^*)^t \to \mathbb{Z}_N^*$ as the compilation of f on $\mathbb{Z}_N^*$ elements as:

$$\hat{f}(A_1, ..., A_t) = \prod_{i=1}^{t} A_i^{f_i} \bmod N$$

Below it is described the PRF with amortized closed-form efficiency for $\mathsf{Comp}(R_1, ..., R_t, f) = \hat{f}(R_1, ..., R_t)$:

- $\mathsf{F.KG}(1^\lambda)$. Let $N = pq$ be the product of two *k-quasi-safe primes*[3] $p = 2^k p' + 1$ and $q = 2^k q' + 1$. Let $\mathcal{R}_k$ be the following subgroup of $\mathbb{Z}_N^*$, $\mathcal{R}_k = \{x^{2^k} : x \in$

---

[2]Extracted from [10]

[3]A safe prime is a prime number of the form $2p+1$, where p is also a prime; we can equivalently say that a safe prime q is such that $(q-1)/2$ is also a prime. More generally, a d-quasi-safe prime q is such that $(q-1)/2^d$ is also a prime.

$Z_N^*\}$, and let $g \in \mathcal{R}_k$ be a generator.
- Choose two seeds K_1, K_2 for a family of PRFs $\mathsf{F}'_{K_{1,2}} : \{0,1\}^* \to \mathbb{Z}_{p'q'}$.
- Output $K = (K_1, K_2)$ and $\mathsf{pp} = (N, k)$. These parameters define a function F with domain $\mathcal{X} = \{0,1\}^* \times \{0,1\}^*$ and range $\mathbb{Z}_N^*$, as described below.

- $\mathsf{F}_\mathsf{K}(\Delta, \tau)$.
  - Generate values $v \leftarrow \mathsf{F}'_{K_1}(\tau)$ and $b \leftarrow \mathsf{F}'_{K_2}(\Delta)$.
  - Output $R = g^{vb} \bmod N$.

- $\mathsf{CFEval}^{\mathsf{off}}_{\tau}(K, f)$. Parse $K = (K_1, K_2)$ as a secret key for the PRF, and f as a function $(\mathbb{Z}_{2^k})^t \to \mathbb{Z}_{2^k}$.
  - For $i = 1$ to t, compute $v_i \leftarrow \mathsf{F}'_{K_1}(\tau_i)$.
  - Next, compute $\rho \; gets \; f(v_1, ..., v_t) \bmod p'q'$.
  - Finally, output $\omega_f = \rho$.

- $\mathsf{CFEval}^{\mathsf{on}}_{\Delta}(K, \omega_f)$. Parse $K = (K_1, K_2)$ as a secret key and $\omega_f = \rho$ as in the previous algorithm. The online evaluation algorithm does the following:
  - Generate $b \leftarrow \mathsf{F}'_{K_2}(\Delta)$.
  - Output $W = g^{\rho b} \bmod N$.

The function above can be proven secure from the DDH assumption in $\mathcal{R}_k$, that we recall below.

Definition 3.6 (DDH) *Let $N = pq$ be the product of two k-quasi-safe primes $p = 2^k p' + 1$ and $q = 2^k q' + 1$. Define $\mathcal{R}_k = \{x^{2^k} : x \in Z_N^*\}$, and let $g \in \mathcal{R}_k$ be a generator. Let a, b, $c \leftarrow \mathbb{Z}_{p'q'}$ be chosen uniformly at random. We define the advantage of an adversary $\mathcal{A}$ in solving the DDH problem as*

$$\boldsymbol{Adv}^{ddh}_{\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(N, k, g, g^a, g^b, g^{ab}) = 1] - \Pr[\mathcal{A}(N, k, g, g^a, g^b, g^c) = 1]|$$

We say that the DDH assumption holds in $\mathcal{R}_k$ if for every PPT algorithm $\mathcal{A}$ we have that $\boldsymbol{Adv}^{ddh}_{\mathcal{A}}(\lambda)$ is negligible.

Theorem 3.1. *If the DDH assumption holds for $\mathcal{R}_k$, and F' is a family of pseudorandom functions, then the function F described above is a pseudorandom function with amortized closed-form efficiency for $\mathsf{Comp} = \hat{f}$ as defined above.* [4]

The $\mathsf{Outsourcing}_{lin}$ scheme for linear functions

Now, we are really ready to present the scheme. Following the description of the pseudorandom function introduced in the last section, it is possible to define two

[4] For the proof, refer to [12].

different schemes, depending on the use of the amortized closed-form efficiency property.

A full description of the scheme $\mathsf{Outsourcing}_{lin}$ in absence of the ACF-efficiency property follows:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{enck}, \mathsf{compk}, \mathsf{deck})$:

- Sample two large-enough k-quasi-safe primes p, q, such that $p = 2^k p' + 1$ and $q = 2^k q' + 1$. Set $N = pq$, and sample $y \leftarrow_{\mathcal{R}} \mathcal{J}_N$. Let us call $\mathcal{R}_k = \{x^{2^k} : x \in \mathbb{Z}_N^*\}$
- Sample a random $\alpha \leftarrow_{\mathcal{R}} \mathbb{Z}^*_{\phi(N)}$, and run $(K, \mathsf{pp}) \leftarrow_{\mathcal{R}} \mathsf{F.KG}(1^\lambda)$ to obtain the seed and the public parameters of an ACF-efficient PRF $\mathsf{F}_K : \{0,1\}^* \times \{0,1\}^* \to \mathcal{R}_k$.
- Output the encoding key $\mathsf{enck} = (k, N, y, \alpha, K)$, the computing key $\mathsf{compk} = (N)$ and the decoding key $\mathsf{deck} = (k, p, p', N, y, \alpha)$.

$\mathsf{Encode}(\mathsf{enck}, \mathsf{L}, m) \to \sigma_m$:

- The multi-label $\mathsf{L} = (\Delta, \tau)$ is the identifier of the input $m \in \mathbb{Z}_{2^k}$.
- Sample $r \leftarrow_{\mathcal{R}} \mathbb{Z}_N^*$ and compute $c = y^m r^{2^k} \bmod N$. Next compute $R \leftarrow \mathsf{F}_K(\Delta, \tau)$, and compute $t = c^\alpha \cdot R \bmod N$.
- Set $\sigma_m = (c, t)$.

$\mathsf{Compute}(\mathsf{compk}, f, \boldsymbol{\sigma}) \to (\sigma_y)$:

- Let $f = (f_1, ..., f_n) \in (\mathbb{Z}_{2^k})^n$ be a linear function and let $\boldsymbol{\sigma} = (\sigma_{m_1} = (c_1, t_1), ..., \sigma_{m_n} = (c_n, t_n))$.
- First, compute $C = \prod_{i=1}^n c_i^{f_i} \bmod N$ to homomorphically evaluate f over the ciphertexts (c_i).
- Second, compute $T = \prod_{i=1}^n t_i^{f_i} \bmod N$ to homomorphically evaluate f over the authentication tags (t_i).
- Output $\sigma_y = (C, T)$.

$\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta, \sigma_y) \to (acc, m')$:

- For $i = 1$ to n: $W_i \leftarrow \mathsf{F}_K(\Delta, \tau_i)$.
- Compute $W = \prod_{i=1}^n W_i^{f_i} \bmod N$.
- Next, if the following equation is satisfied set $acc = 1$ (accept). Otherwise, set $acc = 0$ (reject).

$$T = C^\alpha \cdot W \bmod N$$

- If $acc = 1$, then compute $(\frac{C}{p})_{2^k} = C^{p'} \bmod N = z$, and find $m' \in \{0,1\}^k$ such that $[(\frac{y}{p})_{2^k}]^{m'} = z$ (see refsubsec:3.3.2 for details). If $acc = 0$, set $m' = 0$. Finally, return (acc, m').

A full description of the scheme $\mathsf{Outsourcing}_{lin}$ in presence of the ACF-efficiency property follows:

$\mathsf{KeyGen}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$:

- Sample two large-enough k-quasi-safe primes p, q, such that $p = 2^k p' + 1$ and $q = 2^k q' + 1$. Set $N = pq$, and sample $y \leftarrow_{\mathcal{R}} \mathcal{J}_N$. Let us call $\mathcal{R}_k = \{x^{2^k} : x \in \mathbb{Z}_N^*\}$
- Sample a random $\alpha \leftarrow_{\mathcal{R}} \mathbb{Z}_{\phi(N)}^*$, and run $(K, \mathsf{pp}) \leftarrow_{\mathcal{R}} \mathsf{F.KG}(1^\lambda)$ to obtain the seed and the public parameters of an ACF-efficient PRF $\mathsf{F}_K : \{0,1\}^* \times \{0,1\}^* \rightarrow \mathcal{R}_k$.
- Compute a concise verification information for f by using the offline closed-form efficient algorithm of F, i.e., $\omega_f \leftarrow \mathsf{CFEVal}_\tau^{\mathsf{off}}(\mathsf{K}, \mathsf{f})$.
- Output the encoding key $\mathsf{enck} = (k, N, y, \alpha, K)$, the computing key $\mathsf{compk} = (N)$ and the decoding key $\mathsf{deck} = (k, p, p', N, y, \alpha, \omega_f)$.

$\mathsf{Encode}(\mathsf{enck}, \mathsf{L}, m) \rightarrow \sigma_m$:

- The multi-label $\mathsf{L} = (\Delta, \tau)$ is the identifier of the input $m \in \mathbb{Z}_{2^k}$.
- Sample $r \leftarrow_{\mathcal{R}} \mathbb{Z}_N^*$ and compute $c = y^m r^{2^k} \bmod N$. Next compute $R \leftarrow \mathsf{F}_K(\Delta, \tau)$, and compute $t = c^\alpha \cdot R \bmod N$.
- Set $\sigma_m = (c, t)$.

$\mathsf{Compute}(\mathsf{compk}, f, \boldsymbol{\sigma}) \rightarrow (\sigma_y)$:

- Let $f = (f_1, ..., f_n) \in (\mathbb{Z}_{2^k})^n$ be a linear function and let $\boldsymbol{\sigma} = (\sigma_{m_1} = (c_1, t_1), ..., \sigma_{m_n} = (c_n, t_n))$.
- First, compute $C = \prod_{i=1}^n c_i^{f_i} \bmod N$ to homomorphically evaluate f over the ciphertexts (c_i).
- Second, compute $T = \prod_{i=1}^n t_i^{f_i} \bmod N$ to homomorphically evaluate f over the authentication tags (t_i).
- Output $\sigma_y = (C, T)$.

$\mathsf{Decode}(\mathsf{deck}, \mathcal{P}_\Delta, \sigma_y = (C, T)) \rightarrow (acc, y)$:

- Parse $\mathsf{pp} = (p, \alpha, K, \omega_f)$ as the secret key where ω_f is the concise verification information for f.
- First, run the online closed-form efficient algorithm of F, to compute $W \leftarrow \mathsf{CFEVal}_\Delta^{\mathsf{on}}(K, \omega_f)$.
- Next, if the following equation is satisfied set $acc = 1$ (accept). Otherwise, set $acc = 0$ (reject).

$$T = C^\alpha \cdot W \bmod N$$

- If $acc = 1$, then compute $(\frac{C}{p})_{2^k} = C^{p'} \bmod N = z$, and find $m' \in \{0,1\}^k$ such that $[(\frac{y}{p})_{2^k}]^{m'} = z$ (see Sec. 3.3.2.2 for details). If $acc = 0$, set $m' = 0$. Finally, return (acc, m').

As we already said in Sec. 3.3.2.2, we have not provide the discussion about security analysis of the JL cryptosystem; in such analysis, Joye-Libert define the

Gap-2^k-Residuosity assumption that we here propose in order to define the next theorem.

Definition 3.7 (Gap-2^k-Residuosity assumption) *Let $N = pq$ be the product of two quasi-safe primes $p = 2^k p' + 1$ and $q = 2^k q' + 1$. Define $\mathcal{R}_k = \{x^{2^k} : x \in \mathbb{Z}_N^*\}$. Let $x \leftarrow_{\mathcal{R}} \mathcal{R}_k$ and $y \leftarrow \mathcal{J}_N \setminus QR_N$. We say that the Gap-$2^k$-Residuosity assumption holds if for every PPT adversary $\mathcal{A}$ the following advantage is negligible:*

$$\boldsymbol{Adv}^{Gap-2^k-res}(1^\lambda) = \Big| \Pr[\mathcal{A}(N,k,x)=1] - \Pr[\mathcal{A}(N,k,y)=1] \Big|$$

Theorem 3.2 *If F is a pseudorandom function and the Gap-2^k-Residuosity assumption holds, then $\mathsf{Outsourcing}_{lin}$ is correct, secure and private.*

The proof can be obtained modifying the proof present in [12].

Chapter 4

A *Framework* for implementing Outsourcing schemes

4.1 Framework architecture design

The goal of this chapter is to present the *framework* designed to implement the model described in Sec. 3.1.

When one speaks about a *framework*, one means a complete support architecture, or better a structure, used in computer science for a software development process, with the goal to simplify the life of the programmer. More technically, a framework can be a set of abstract classes together with the relations among them[?]. In this sense, we can refer to this thesis work as to a framework, since basically it consists of some "*interfaces*" that define a behaviour and that can be implemented every time by a specific scheme.

As already explained in the Sec. 3.1, the Outsourcing scheme is born to create a model that allows to be employed for doing verifiable computation *together* with homomorphic encryption, but at the same time that could be usable for doing *only* verifiable computation or *only* homomorphic encryption. The idea behind the framework is essentially the same; we want to provide a *common interface that could be used to implement all those several outsourcing schemes.*

This chapter is organized as follows: we first present the software tools employed to build the framework (see Sec. 4.1.1), then we provide an high level description of the structure of the framework (see Sec. 4.1.2).

4.1.1 Support tools

The code of the software has been written in C++, using some external libraries: `NTL`[1] and `Crypto++® Library 5.6.2`[2].

`NTL (A Library for doing Number Theory)` is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. It provides high quality implementations of state-of- the-art algorithms for:

- arbitrary length integer arithmetic and arbitrary precision floating point arithmetic;
- polynomial arithmetic over the integers and finite fields (is one of the fastest available anywhere, and has been used to set "world records" for polynomial factorization and determining orders of elliptic curves);
- lattice basis reduction (is also one of the best available anywhere, in terms of both speed and robustness).

NTL can be used in conjunction with `GMP (the GNU Multi-Precision library)` for enhanced performance. GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface. The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc. GMP is carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is achieved by using fullwords as the basic arithmetic type, by using fast algorithms, with highly optimised assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed.

NTL provides a clean and consistent interface to a large variety of classes representing mathematical objects; indeed, the library consists of a number of *software modules* (classes).

For the purpose of this work, just the module (class) `ZZ` it has been employed. It is used to represent signed, *arbitrary length integers*. Routines are provided for

[1] http://www.shoup.net/ntl/

[2] http://www.cryptopp.com/

all of the basic arithmetic operations, as well as for some more advanced operations (such as Greatest Common Divisors (GCDs), Jacobi symbols, modular arithmetic, and primality testing). The space is automatically managed by the constructors and destructors. This module also provides routines for generating small primes, and fast routines for performing modular arithmetic on single-precision numbers.

`Crypto++ Library` is a free C++ class library of cryptographic schemes. Currently the library contains algorithms for authenticated encryption schemes, high speed stream ciphers, AES and AES candidates, block cipher modes of operation (ECB, CBC, CBC Ciphertext Stealing, CFB, OFB, Counter Mode), hash functions and many other things. The library has been basically used to build the pseudo-random function of the cryptographic scheme implemented, and for this goal, AES with few modes of operation (ECB and Counter Mode) and with SHA-3 (Keccak), added in the last release, have been employed.

4.1.2 High level description

The step from the theoretical model towards the birth of the framework has been quick and quite easy. Essentially, the core of the framework is represented by six *abstract base classes* (C++ doesn't provide the `interface` keyword as Java or C# do), each one containing a *pure virtual function* reflecting one of the algorithms of the Outsourcing scheme presented in Sec. 3.1.

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error. Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions; in this way we say that it supports the interface declared by the ABC.

In order to define the virtual functions inside the six ABCs, some other support classes have been defined. Some of them are basically empty classes, providing just a *default constructor* and the *destructor*, and they should be extended every time that a specific scheme has to be implemented (the child classes of such specific implementation will inherit the required base classes of the framework specifying the behaviour by adding the own members, data and functions); the other classes, instead, contain also already defined data members and functions. Such classes are born to be employed as they are and it is not possible to inherit them.

In the following part of this section I will present the structure of the framework, describing each class with the support of *pseudo* UML class diagrams[3] and, when

[3] In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of

possible, showing part of the code.

In an effort to facilitate the understanding of the composition of the framework, we show the Table 4.1 containing all its classes. This table is composed of three columns containing the six ABCs (*interfaces*), the classes that should be inherited and the classes that do not have to be inherited, respectively.

Abstract classes (*interfaces*)	**Classes can be extended**	**Classes do not extend**
`SchemeKeyGen`	`EncodKey`	`Keys`
`SchemeEncode`	`ComputKey`	`OnlineDecodKey`
`SchemeCompute`	`DecodKey`	`Decoding`
`SchemeDecode`	`OnDecodKey`	`Program`
`SchemeOfflineDecode`	`Label`	
`SchemeOnlineDecode`	`Message`	
	`Encoding`	
	`Program_function`	
	`Comput_inputs`	
	`My_clock`	

Table 4.1: Framework content.

Why several ABCs?

To be thorough, I would like to motivate the choice of define many ABCs, *each with only one virtual function*, instead of *a single ABC containing all the virtual functions*. This choice is coherent with the model in Sec. 3.1; indeed, there are many players acting different roles, e.g. a client that (can generate the keys) encodes the inputs and decodes the result, a remote server that computes the result and sends it to the user.

For this reason, we want to guarantee the possibility to *distribute* the algorithms of the model among the different acting nodes, e.g. instead of assign to the user the role of generate the keys, there could be a specific process in charge of this task, as well as the decoding algorithm could be executed by an user different from that who asked the computation.

If there was only one ABC with all the virtual functions, every node should implement all these virtual functions (maybe providing some blank implementations),

static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

even if it was interested in using just one of them. Using the configuration *"one abstract class —one virtual function"* we increase the *modularity*, giving to every node participating to the model the chance to choose which of the functions implement.

Key generation

The first ABC I am going to present is the `SchemeKeyGen` that contains the `KeyGen` pure virtual function, Fig.4.1. It is the function corresponding to the KeyGen algorithm of the Outsourcing scheme (Sec. 3.1); it takes a `long` parameter, the *security parameter*, and outputs an object of the `Keys` class.

```
1  class SchemeKeyGen{
2      public:
3          virtual Keys* KeyGen(long sec_param)=0;
4  };
```

Figure 4.1: The `KeyGen` pure virtual function inside the `SchemeKeyGen` abstract class.

In the Outsourcing scheme the output of the KeyGen algorithm consists of the three keys enck, compk and deck; in the framework they become the three classes `EncodKey`, `ComputKey` and `DecodKey`), one for each of the algorithms (Encode, Compute and Decode). When one will have to implement a specific scheme, one will have to extend the three key classes, or just some of them, specifying the key structure by adding specific data and function members.

The three keys are stored in an object of the not abstract class `Keys` that provides also the `set` and `get` functions to manage them. So, the `KeyGen` virtual function has the task of generating all the keys for the encoding, computing and decoding phase, using them to build the `Keys` object that will be returned by the function. Fig.4.2 shows a pseudo UML class diagram for the above classes.

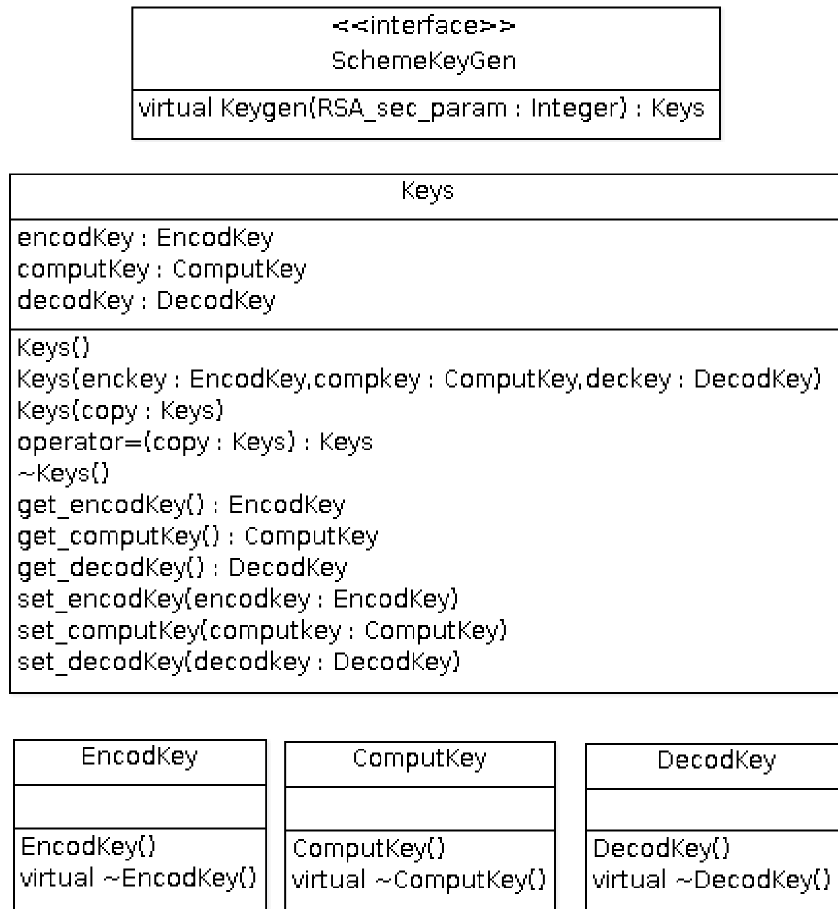

Figure 4.2: Pseudo UML class diagram for the `SchemeKeyGen` ABC, the `Keys` class and the `EncodKey`, `ComputKey` and `DecodKey` classes.

Encoding

The next ABC is the `SchemeEncode` containing the pure virtual function `Encode`, Fig.4.3; it takes as parameter the `EncodKey` (contained in the `Keys` object provided by the keys generation function), a *multi label*, i.e. two objects of the `Label` class, *delta* representing a *data set* and *tau_i* associated with the *message belonging to that data set*, and the *message* to encode (`Message` object). The output produced is an *encoding* (`Encoding` object) to give to the remote server to compute with. Actually, the `SchemeEncode` ABC provides also an overloaded `Encode` function that takes as parameters only the `EncodKey` and the `Message` objects (that are the minimum essentials elements of a generic encoding algorithm, i.e. think of a FHE).

```
1  class SchemeEncode{
2      public:
3          virtual Encoding* Encode(const EncodKey& encKey, const
               Label& delta, const Label& tau_i, const Message& m)=0;
4          virtual Encoding* Encode(const EncodKey& encKey, const
               Message& m)=0;
5  };
```

Figure 4.3: The `Encode` pure virtual function inside the `SchemeEncode` abstract class.

Note that such function `Encode` is slightly different from the theoretical Encode algorithm of the Outsourcing scheme due to the fact that the multi-label $\mathsf{L} = (\Delta, \tau)$ (Sec. 2.3.1.1) is represented in the framework with a couple of `Label` object, where a `Label` object simply contains a `string`. The Label class can be used as it is or it can be extended to be personalized. A `Message` object represents instead the message to encode. The `Message` class is another class that will be surely extended by each specific implementation in order to define the nature of the messages space. The `Encoding` object symbolizes the σ_m output of the Encode algorithm of the Outsourcing scheme. The `Encoding` class will have to be also extended in order to fill it with the members that allow to define the encodings space. Fig.4.4 shows a pseudo UML class diagram for the above classes.

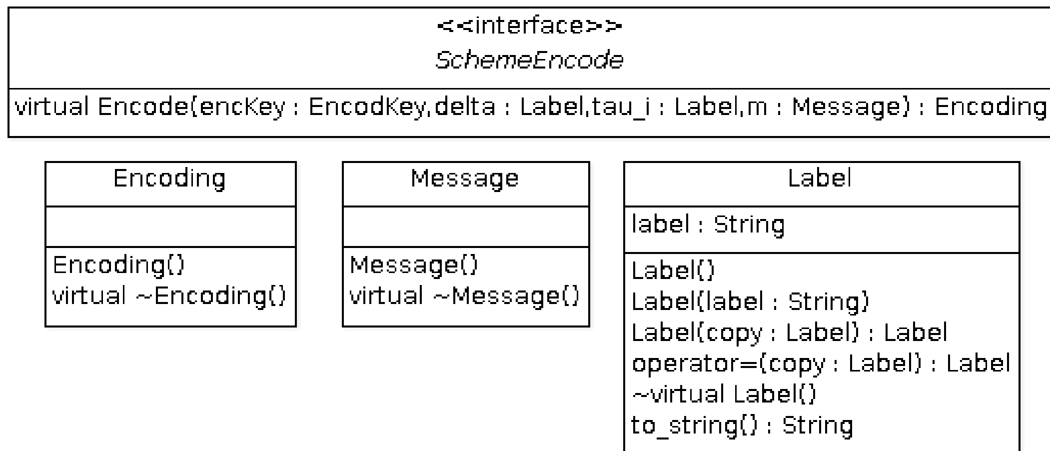

Figure 4.4: Pseudo UML class diagram for the `SchemeEncode` ABC, the `Encoding`, the `Message` and the `Label` classes.

As we have already said, the method `Encode` inside the `SchemeEncode` ABC is

an *overloaded* method. There could be many schemes that do not provide for a *multi-label* parameter (i.e. the Joye-Libert cryptosystem where the encoding phase is basically an encryption that does not require any label). We can say that the *mandatory* parameters are the `EncodKey` and the `Message`, while the two `Label` objects (*multi-label*) are *optional*. This lead to an overloaded version of the method that receive only two parameters, the `EncodKey` and the `Message`.

Computation

The pure virtual function that should be implemented by the server is the `Compute` function, Fig.4.5, inside the `SchemeCompute` ABC. The function takes the `ComputKey` (contained in the `Keys` object provided by the keys generation function) as first parameter, a `Program_function` object, and a `Comput_inputs` object over which the `Program` has to make the computation. `Compute` outputs an own *encoding* (object `Encoding`) of the result.

```
1  class SchemeCompute{
2      public:
3          virtual Encoding* Compute(const ComputKey& compKey, const
               Program_function& funct, const Comput_inputs& inputs)=0;
4  };
```

Figure 4.5: The `Compute` pure virtual function inside the `SchemeCompute` abstract class.

Note that the `Compute` function takes the same parameters of the Compute algorithm of the Outsourcing scheme; the `Program_function` object represents the function to be computed and every specific implementation will extend such class specifying the own function. The `Comput_inputs` represents the list of encodings to give to the function as inputs; a `Comput_inputs` object contains a vector of `Encoding` objects together with the methods to fill such vector, to retrieve the elements, etc. Fig.4.6 shows a pseudo UML class diagram for the above classes.

Decoding

The `Encoding` returned by the server is used in the *decoding* step by the `Decode` pure virtual function, Fig.4.7, inside the `SchemeDecode` ABC; such function, using the `DecodKey` (contained in the `Keys` object provided by the keys generation function), a `Program` object and a `Label` object, produces a `Decoding` object. Actually,

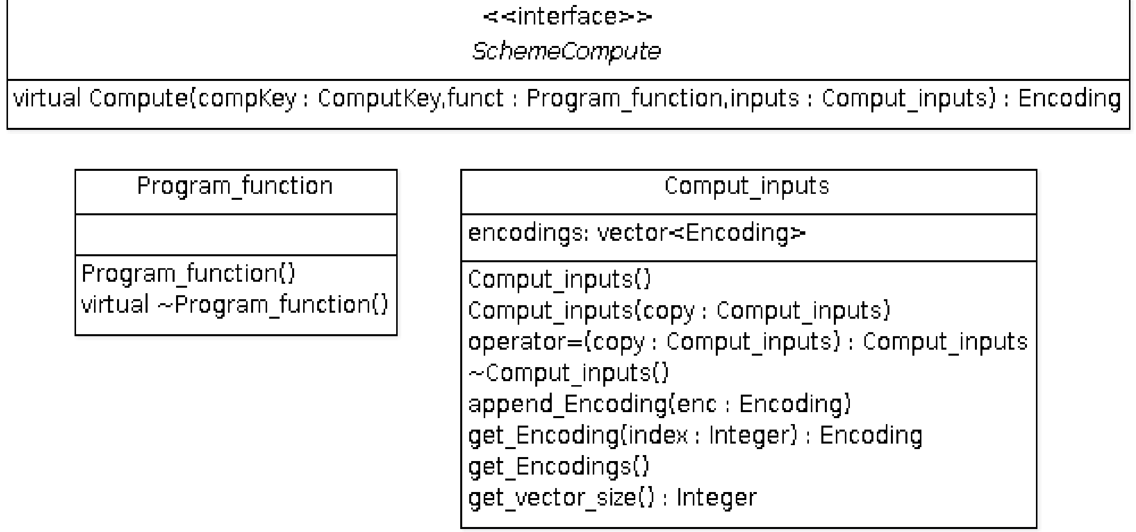

Figure 4.6: Pseudo UML class diagram for the `SchemeCompute` ABC, the `Program_function` and the `Comput_inputs` classes.

the `SchemeDecode` ABC provides also an overloaded `Decode` function that takes as parameters only the `DecodKey` and the `Encoding` objects (that are the minimum essentials elements of a generic decoding algorithm, i.e. think of a FHE).

```
1  class SchemeDecode{
2      public:
3          virtual Decoding* Decode(const DecodKey& decodKey, const
              Program& prog, const Label& delta, const Encoding&
              encod)=0;
4          virtual Decoding* Decode(const DecodKey& decodKey, const
              Encoding& encod)=0;
5  };
```

Figure 4.7: The `Decode` pure virtual function inside the `SchemeDecode` abstract class.

Remember that a *labeled program* $\mathcal{P}$ (Sec. 2.3.1.1) is defined by a tuple $(f, \tau_1, ..., \tau_n)$ where $f : \mathcal{M}^n \to \mathcal{M}$ is a function on n variables, and each $\tau_i \in \{0,1\}^*$ is the label of the i-th variable input of f. In the framework, a labeled program $\mathcal{P}$ becomes an object of the `Program` class that contains the *function* (`Program_function` object) to be computed and the labels (`Label` objects) associated to the inputs of

the function. In this way, the *multi-labeled program* $\mathcal{P}_{\Delta}$, present in the Decode algorithm of the Outsourcing scheme, is represented with a couple of `Program` and `Label` objects. Finally, an object of the `Decoding` class contains the *acceptance bit* (boolean element) and a `Message` object. Fig.4.8 shows a pseudo UML class diagram for the above classes.

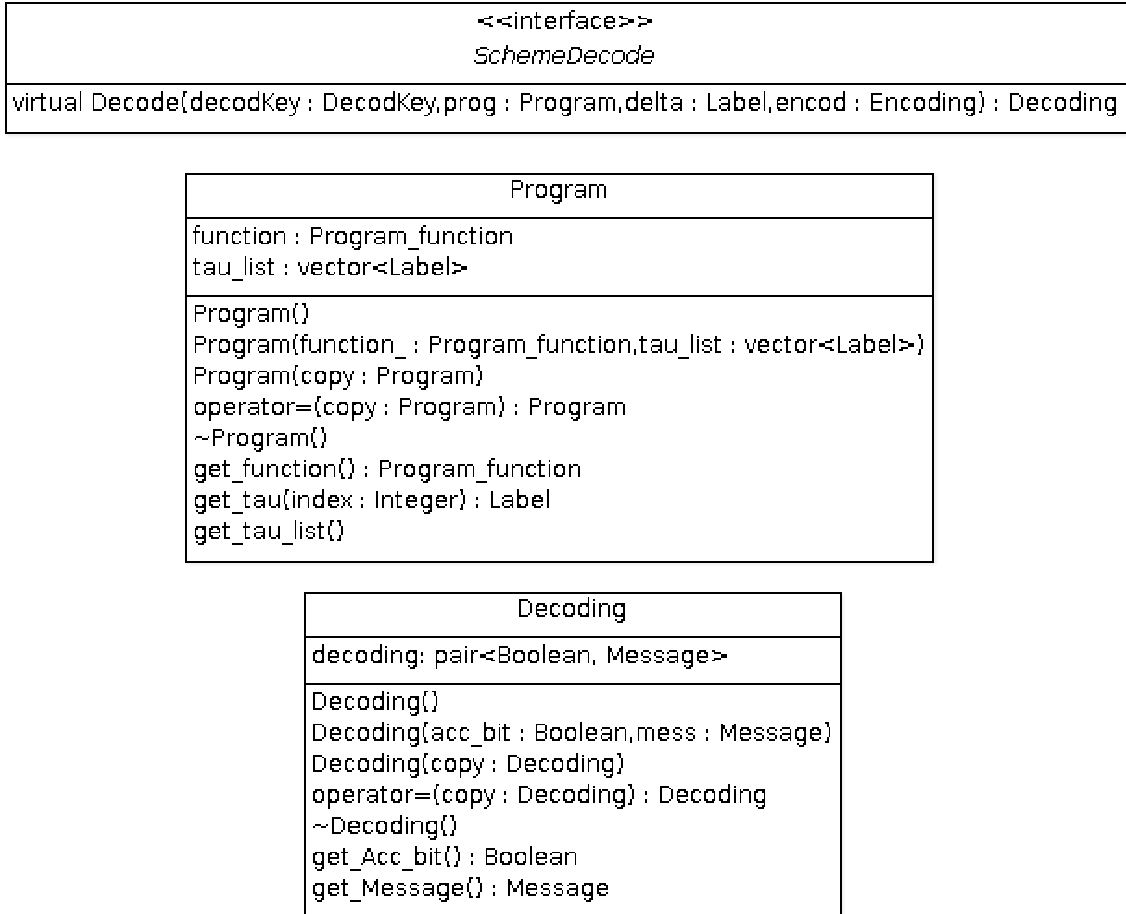

Figure 4.8: Pseudo UML class diagram for the `SchemeDecode` ABC and the `Decoding` class.

The same observation made for the `Encode` virtual function inside the `SchemeEncode` ABC remains valid for the `Decode` virtual function inside the `SchemeDecode` ABC. It is an *overloaded* method indeed there could be many schemes that do not provide for a *multi-labeled program*, i.e. the couple composed by a `Program` and a `Label` object (i.e. the Joye-Libert cryptosystem where the decoding phase is basically a decryption that does not require any multi-labeled program). We can say that the *mandatory* parameters are the `DecodKey` and the `Encoding`, while the `Label` and

the `Program` are *optional*. This lead to an overloaded version of the method that receive only two parameters, the `DecodKey` and the `Encoding`.

According to the theoretical model, in order to guarantee the efficient verification property in the amortized sense, the *decoding phase* can be split into two steps; so, the `Decode` function can be replaced with the `OfflineDecode` and `OnlineDecode` pure virtual functions. The first, Fig.4.9, receives as parameters a `DecodKey` object (contained in the `Keys` object provided by the keys generation function) and a `Program` object and outputs an `OnlineDecodKey` object. Such `OnlineDecodKey` object is then passed to the second method, Fig.4.10, together with a `Label` object and the `Encoding` object generated by the server to finally produce the `Decoding` object.

```
1  class SchemeOfflineDecode{
2      public:
3          virtual OnlineDecodKey* OfflineDecode(const DecodKey&
               decodKey, const Program& prog)=0;
4  };
```

Figure 4.9: `OfflineDecode` pure virtual function inside the `SchemeOfflineDecode` abstract class.

```
1  class SchemeOnlineDecode{
2      public:
3          virtual Decoding* OnlineDecode(const OnlineDecodKey&
               onlineDecodKey, const Label& delta, const Encoding&
               encod)=0;
4  };
```

Figure 4.10: `OnlineDecode` pure virtual function inside the `SchemeOnlineDecode` abstract class.

An `OnlineDecodKey` object contains a `DecodKey` object and an `OnDecodKey` object. The latter is an element dependent on the specific implementation, so one needs to extend the `OnDecodKey` class. Fig.4.11 shows a pseudo UML class diagram for the above classes.

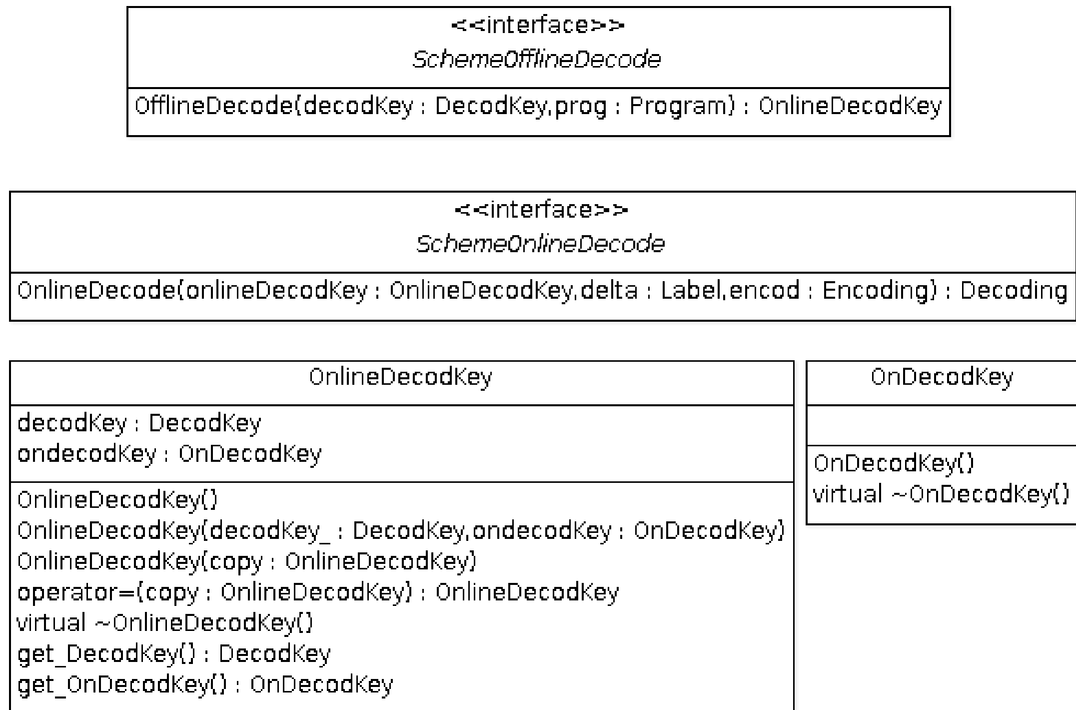

Figure 4.11: Pseudo UML class diagram for the `SchemeOfflineDecode` and the `SchemeOnlineDecode` ABCs, the `OnlineDecodKey` and the `OnDecodKey` classes.

A clock utility

We conclude the description of the framework describing the last utility class. It is about a *clock* class that can be used by each developer that wants to test the timing performance of own code. Such class has been called `My_clock` and every object represents a clock; every clock object has two function `start()` and `stop()` to respectively be called at the begin and at the end of the code one wants to evaluate. Every computed time interval is stored by the clock object in a vector, and only when the `write_log()` function is called, the content of the vector is written into a log file. Of course, every object clock has been provided with a security check to avoid errors like stopping a clock not yet started or like starting a clock already started. In all these cases, an exception is thrown.

Assuming that a clock is employed to evaluate many times the same code, maybe for testing reasons, it is also possible asking to the clock to compute the average, the variance and the standard deviation of all the values computed; to do this, one needs to pass to the class constructor a ”true” set boolean value. A time interval is retrieved by using the `GetTme()` function present in the NTL `tools` module. Such function returns the number of seconds of CPU time used by the process. Probably

there exist many other ways, more or less accurate, to get the temporal information in C++, but the choice of using the `GetTime()` just seemed the simplest and the best one, since that it belongs to the NTL library already used in the framework, and because it provides a good accuracy guaranteeing cross-platform usability.

Final remarks

The `My_clock` class description ends the presentation of the framework. Table 4.2 shows the mapping between the Outsourcing scheme algorithms and the pure virtual functions contained in the six ABCs of the framework.

Outsourcing **scheme algorithms**
$\mathsf{KeyGen}(\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$
$\mathsf{Encode}(\mathsf{sk}, \mathsf{L}, m) \rightarrow \sigma_m$
$\mathsf{Compute}(\mathsf{pk}, f, \boldsymbol{\sigma}) \rightarrow (\sigma_y)$
$\mathsf{Decode}(\mathsf{sk}, \mathcal{P}_\Delta, \sigma_y) \rightarrow (acc, y)$
$\mathsf{OfflineDecode}(\mathsf{sk}, \mathcal{P}) \rightarrow \mathsf{OnDK}_\mathcal{P}$
$\mathsf{OnlineDecode}(\mathsf{sk}, \mathsf{OnDK}_\mathcal{P}, \Delta, \sigma_s) \rightarrow (acc, y)$
Framework virtual functions
`virtual Keys*` **KeyGen**`(long RSA_sec_param)=0;`
`virtual Encoding*` **Encode**`(const EncodKey& encKey, const Label& delta, const Label& tau_i, const Message& m)=0;`
`virtual Encoding*` **Compute**`(const ComputKey& compKey, const Program& prog, const Comput_inputs& inputs)=0;`
`virtual Decoding*` **Decode**`(const DecodKey& decodKey, const Program& prog, const Label& delta, const Encoding& encod)=0;`
`virtual OnlineDecodKey*` **OfflineDecode**`(const DecodKey& decodKey, const Program& prog)=0;`
`virtual Decoding*` **OnlineDecode**`(const OnlineDecodKey& onlineDecodKey, const Label& delta, const Encoding& encod)=0;`

Table 4.2: Equivalence between the Outsourcing scheme algorithms and the Framework virtual functions.

Chapter 5

Implementations using our Framework

This chapter of the thesis is dedicated to the description of some interesting implementations that use the framework presented in the previous chapter.

The primary goal of this work is to implement the $\mathsf{Outsourcing}_{lin}$ scheme described in Sec. 3.3.3.2. In order to do this, we first present the implementation of the Joye-Libert cryptosystem (see Sec. 5.1), highlighting in this way how it is also possible to employ the framework to implement even just a simple homomorphic encryption scheme. In Sec. 5.2 we therefore implement the $\mathsf{Outsourcing}_{lin}$ scheme.

5.1 Implementation of the Joye-Libert homomorphic encryption scheme

The first thing to do is understanding which interfaces of the framework we need to implement and which classes we need to use or to extend. As we have already seen in Sec. 3.3.2.2, the scheme consists of the following algorithms: generation of the keys, encryption, decryption and evaluation. There is no an efficient verification property to satisfy (implementing the `SchemeOfflineDecode` and `SchemeOnlineDecode` ABCs). So, only four ABCs of the framework must be implemented:

- `SchemeKeyGen`;
- `SchemeEncode`;
- `SchemeCompute`;

- `SchemeDecode`.

Regarding the classes, we do not need to use or to extend all those classes related to the efficient verification algorithms (`OnlineDecodKey`, `OnDecodKey`, etc.). So, the classes to extend are `EncodKey`, `ComputKey`, `DecodKey`, `Message`, `Encoding` (and `Program_function`) while the classes to simply use are `Label` and `Decoding`. Actually the content of the `ComputKey` class is the same of `EncodKey`, because in the case of the Joye-libert cryptosystem the evaluation key is represented by the encoding key.

Note that, in the context of this cryptosystem, the names of the classes acquire specific meanings: *encode* means *encrypt* while *decode* means *decrypt*; in the same way, an *encoding* object will represent an *encryption* object and a *decoding* object will represent a *decryption* object.

5.1.1 Key generation

Recall from Sec. 3.3.2.2 that $\mathsf{KeyGen}(1^\lambda)$ takes a security parameter λ, randomly generates primes p, $q \equiv 1 \pmod{2^k}$ and sets $N = pq$ and picks $y \in \mathbb{J}_N \setminus \mathbb{QR}_N$. The public and secret keys are $\mathsf{pk} = \{N, y, k\}$ and $\mathsf{sk} = \{p\}$, respectively.

The most interesting part of this algorithm is surely represented by the generation of the two random primes p and q.

Before to analyze this part, I will show how the framework has been employed to obtain the KeyGen algorithm.

What we need is to implement the `KeyGen` virtual function of the `SchemeKeyGen` "interface" of the framework. For this reason, one can create a concrete class, i.e. called `JL_13_SchemeKeyGen`, in which define a body for the `KeyGen` virtual function.

We could call the classes that inherit `EncodKey` and `DecodKey`, as `JL_13_EncodeKey` and `JL_13_DecodeKey`, respectively.

The `JL_13_EncodeKey` class will contain:

- the NTL ZZ variables where store the public key elements (N, y, k, 2^k);
- an additional constructor receiving as parameter the public key elements;
- a classic get function for each variable to retrieve.

Despite redundant,we chose to store also the value 2^k in order to just increase the efficiency and to avoid its re-computation every time that one needs it.

The `JL_13_DecodeKey` class will contain:

- all the NTL ZZ variables for the secret (p) and public key elements (N, y, k) for the decryption phase;

- an additional constructor receiving as parameter the secret and public key elements;

- a classic get function for each variable to retrieve.

Random primes generation

Here we recall some basic facts from [1] about prime numbers that are useful in our work.

Definition 5.1 An integer n which is believed to be prime on the basis of a probabilistic primality test is called a *probable prime*.

A primality test is a test to determine whether or not a given number is prime, as opposed to actually decomposing the number into its constituent prime factors (which is known as prime factorization). Primality tests come in two varieties: deterministic and probabilistic. Deterministic tests determine with absolute certainty whether a number is prime. Probabilistic tests can potentially (although with very small probability) falsely identify a composite number as prime (although not vice versa). However, they are in general much faster than deterministic tests.

One commonly faced problem in cryptography is how to generate a prime number of a given length in bits. The prime number theorem (PNT) describes the asymptotic distribution of the prime numbers among the positive integers. It formalizes the intuitive idea that primes become less common as they become larger. It asserts that the proportion of (positive) integers $\leq x$ that are prime is approximately $1/\ln x$. Since half of all integers $\leq x$ are even, the proportion of *odd* integers $\leq x$ that are prime is approximately $2/\ln x$.

From this fact we can determine a way to select a random k-bit (probable) prime; it will be enough picking random k-bit odd integers n until one can be declared “prime” by the MILLER-RABBIN(n, t) algorithm for an appropriate value of the security parameter t.

The Miller-Rabin test is the probabilistic primality test used most in practice, also know as the *strong pseudoprime test*. It takes as inputs an odd integer $n \geq 3$ and a security parameter $t \geq 1$ (a parameter that determines the accuracy of the test), and outputs an answer “prime” or “composite” to the question: “Is n prime?”.

In general, if a random k-bit odd integer n is divisible by a small prime, it is less computationally expensive to rule out the candidate n by trivial division than by using the Miller-Rabin test. Since the probability that a random integer n has a small prime divisor is relatively large, before applying the Miller-Rabin test, the

candidate n should be tested for small divisors below a pre-determined bound B. This can be done by dividing n by all the primes below B, or by computing greatest common divisor of n and (pre-computed) products of several of the primes $\leq B$. The discussion about how finding the optimal trial division bound B lies outside of this context (for more details refers to [1]).

Algorithm 5.1 Random search for a prime using the Miller-Rabin test

---

RANDOM-SEARCH(k,t)
INPUT: an integer k, and a security parameter t
OUTPUT a random k-bit probable prime

---

1. Generate an odd k-bit integer n at random.
2. Use trial division to determine whether n is divisible by any add odd prime $\leq B$. If it is then go to step 1.
3. If the Miller-Rabin test with n and t outputs "prime" then return n. Otherwise, go to step 1.

Since the Miller-Rabin test does not provide a mathematical proof that a numbers is indeed prime, the number n returned by Alg. 5.1 is a probable prime.

It is important, therefore, to have an estimation of the probability that n is in fact composite. We denote this probability by $p_{k,t}$. In [1] the authors provide some mathematics formulas to establish some upper bounds on $p_{k,t}$.

(i) $p_{k,t} < k^2 4^{2-\sqrt{k}}$ for $k \geq 2$;

(ii) $p_{k,t} < k^{3/2} 2^t t^{-1/2} 4^{2-\sqrt{tk}}$ for $(t = 2,\ k \geq 88)$ or $(3 \leq t \leq k/9,\ k \geq 21)$;

(iii) $p_{k,t} < \frac{7}{20} k 2^{-5t} + \frac{1}{7} k^{15/4} 2^{-k/2-2t} + 12k 2^{-k/4-3t}$ for $k/9 \leq t \leq k/4,\ k \geq 21$;

(iv) $p_{k,t} < \frac{1}{7} k^{15/4} 2^{-k/2-2t}$ for $t \geq k/4,\ k \geq 21$.

Actually, there exist more accurate formulas that the authors don't provide and that have been used to produce the values in Table 5.1 that can be used to fix the number of *rounds* t for the Miller-Rabin test.

k	t	k	t	k	t	k	t	k	t
100	27	500	6	900	3	1300	2	1700	2
150	18	550	5	950	3	1350	2	1750	2
200	15	600	5	1000	3	1400	2	1800	2
250	12	650	4	1050	3	1450	2	1850	2
300	9	700	4	1100	3	1500	2	1900	2
350	8	750	4	1150	3	1550	2	1950	2
400	7	800	4	1200	3	1600	2	2000	2
450	6	850	3	1250	3	1650	2	2050	2

Table 5.1: Smallest values of t for which $p_{k,t} \leq (\frac{1}{2})^{80}$.

As we have already said, unfortunately the authors don't provide the improved formulas, so for the security parameters greater than 80 we have used the standard formulas showed before in order to make the keys generation algorithm parametric on the bit-size of the prime and on the security parameter.

Bit-size of the composite N

In order to determine the bit-size of the composite N we followed recommendations from a document of the European Union Agency for Network and Information Security in [2].

It is a report with cryptographic guidelines supporting the security measures required to protect personal data in online systems. It discusses about conservative guiding principles, based on current state-of-the-art research, addressing construction of new systems with a long life cycle. In the section 3.6 of such report the authors provide a Key Size Analysis (Fig. 5.1) essentially following the NIST equivalence [19] between the different key sizes; we propose here a part of this section:

> Providing key sizes for long term use is somewhat of a hit-and-miss affair, for a start it assumes that the algorithm you are selecting a key size for is not broken in the mean time. So in providing key sizes for specific application domains we make an *implicit* assumption that the primitive, scheme or protocol which utilises this key size is not broken in the near future. All primitives and schemes marked as suitable for future use in this document we have confidence will remain secure for a significant period of time. Making this assumption still implies a degree of choice as to key size however. The AES block cipher may remain secure for the next fifty years, but one is likely to want to use a larger key size for data which one wishes to secure for fifty years as opposed to, say, five

years. Thus in providing key size guidelines we make two distinct cases for schemes relevant for future use. The first cases is for security which you want to ensure for *at least* ten years (which we call *near term*), and secondly for security for thirty to fifty years (which we call *long term*). Again we reiterate these are purely key size guidelines and they do not guarantee security, nor do they guarantee against attacks on the underlying mathematical primitives.

			Future System Use	
	Parameter	Legacy	Near Term	Long Term
Symmetric Key Size	k	80	128	256
Hash Function Output Size	m	160	256	512
MAC Output Size	m	80	128	256*
RSA Problem	$\ell(n) \geq$	1024	3072	15360
Finite Field DLP	$\ell(p^n) \geq$	1024	3072	15360
	$\ell(p), \ell(q) \geq$	160	256	512
ECDLP	$\ell(q) \geq$	160	256	512
Pairing	$\ell(p^{k \cdot n}) \geq$	1024	3072	15360
	$\ell(p), \ell(q) \geq$	160	256	512

Figure 5.1: Key Size Analysis. A $*$ notes the value could be smaller due to specific protocol or system reasons, the value given is for general purposes.

The recommendations for the RSA problem actually follow state of the art research in factoring algorithms. Since the assumptions used for our schemes (e.g. the Joye-Libert cryptosystem and the $\mathsf{Outsourcing}_{lin}$ scheme) are related to factoring we follow these recommendations. For this reason, in the code we will show in the following, we will refer to the security parameter with the name `RSA_sec_param`.

A discussion about the number of *rounds* t for the Miller-Rabin test As we have said before, in order to determine the number t for the Miller Rabin test, we should use the formulas presented above. In this section, we show that we can actually focus just on one formula and use it to individuate the best value t. Considering our implementation, we have just seen that the composite number N can assume three different values depending on the security parameter; once that we determine N, we know that the bit-size of the two primes p and q (equal to bit size of N divided by 2), respectively.

Let's consider the formula (ii) ; we can use such formula if $t \leq k/9$ where k is the bit size of the number of which we want to check the primality. In our case we know k that is the bit size of p (and q), and we can affirm that the desired value of

t (for each security parameter) is always in this interval, so we can use the formula (ii) . Table 5.2 shows the computed values of t for each bit size of Nk (namely of p and q).

bit size of N	t
1024	5
3072	4
15360	3

Table 5.2: Number or rounds t for the Miller Rabin test depending on the bit size of N.

Code

All the previous considerations have been used to randomly generates the two primes p, $q \equiv 1 \pmod{2^k}$. Fig. 5.2 and Fig. 5.3 show the code of the random primes generation inside the `KeyGen` function.

As it is possible to see from the code, there is a preliminary setting phase to determine the right bit size of the numbers p' and q' so that the bit size of p and q is compliant with the bit size of N. Then, in order, there is the procedure before described to fix the number of tests to pass to the function `ProbPrime` and the security check (Sec. 3.3.2.2). `long ProbPrime(const ZZ& n, long NumTrials = 10)` is a function of the NTL ZZ class and it performs up to `NumTrials` Miller-Rabin tests (after some trial division) to say if `n` is a probable prime; it essentially implements the Alg. 5.1. `void RandomLen(ZZ& x, long num_bits)` is another function of the NTL ZZ class and it generates a pseudo-random number `x` with precisely `num_bits` bits, or 0 if `num_bits` ≤ 0.

The last part of the `KeyGen` function picks a $y \in \mathbb{J}_N \setminus \mathbb{QR}_N$. This result is obtained by implementing the Alg. 3.2. The code of such implementation is shown in Fig. 5.4.

`void RandomBnd(ZZ& x, const ZZ& n)` is a NTL ZZ class function that outputs a pseudo-random number in the range $0..n-1$, or 0 if $n \leq 0$. `ZZ GCD(const ZZ& a, const ZZ& b)` is also a NTL ZZ class function that computes the Greatest Common Divisor (GCD) between a and b; it uses a binary GCD algorithm. `long Jacobi(const ZZ& a, const ZZ& n)` is another NTL ZZ class function that computes the Jacobi symbol of a and n, assuming $0 \leq a < n$, and n odd.

```
1  Keys* JL_13_SchemeKeyGen:: KeyGen(long sec_param )
2  {
3      //...
4      long N_factoring_sec_par_bit_size=0;
5      switch(RSA_sec_param){
6          case(80):
7              N_factoring_sec_par_bit_size= 1024;
8                          miller_rabin_tests=5;
9                      break;
10                 case(128):
11                     N_factoring_sec_par_bit_size= 3072;
12                         miller_rabin_tests=4;
13                     break;
14                 case(256):
15                         N_factoring_sec_par_bit_size= 15360;
16                         miller_rabin_tests=3;
17                     break;
18                 default:
19                     N_factoring_sec_par_bit_size= 1024;
20                         miller_rabin_tests=5;
21                     break;
22     }
23         try{
24                 if(k_message_bit_size>=N_factoring_sec_par_bit_size/4-RSA_sec_param)
25                         throw sec_exec;
26         }
27         catch (exception& e){
28                         cout << e.what() << '\n';
29                         return NULL;
30         }
31     ZZ seed(GetTime()*rand());
32     SetSeed(seed);
33     long bit_size_p_q_prime=(N_factoring_sec_par_bit_size/2) -
           k_message_bit_size;
34     //...
```

Figure 5.2: Part (a) of the KeyGen virtual function inside the JL_13_SchemeKeyGen class.

```
36
37      ZZ two_to_the_k;
38      power2(two_to_the_k, k_message_bit_size);
39      ZZ p,p_prime;
40      RandomLen(p_prime,bit_size_p_q_prime);
41      while(!ProbPrime(two_to_the_k*p_prime+1,miller_rabin_tests)){
42          RandomLen(p_prime,bit_size_p_q_prime);
43      };
44      p=two_to_the_k*p_prime+1;
45      ZZ q,q_prime;
46      RandomLen(q_prime,bit_size_p_q_prime);
47      while(!ProbPrime(two_to_the_k*q_prime+1,miller_rabin_tests)){
48          RandomLen(q_prime,bit_size_p_q_prime);
49      };
50      q=two_to_the_k*q_prime+1;
51      //...
```

Figure 5.3: Part (b) of the `KeyGen` function inside the `JL_13_SchemeKeyGen` class.

5.1.2 Encoding

In this case, what we need is the `Encode` virtual function of the `SchemeEncode` "interface" of the framework. As always, one can create a concrete class, i.e. called `JL_13_SchemeEncode`, in which define a body for the `Encode` virtual function, specifically the overloaded `Encode` function that takes as parameters only the `EncodKey` and the `Message` objects. There are no special observations to make about the encoding (encryption) step. The code simply reflects the algorithm showed in the scheme description in Sec. 3.3.2.2. The code is shown in Fig. 5.5.

We could call the class that inherits the `Message` class as `JL_13_Message` class. It will contain:

- a NTL ZZ object where to store the value to encrypt;
- an additional constructor receiving as parameter directly a NTL ZZ object representing the value to encrypt;
- an additional constructor receiving as parameter a `long` representing the bit size of the integer to encrypt; in this case, inside the constructor it will be called an other function of the class to generate a random NTL ZZ integer of the right size representing the value to encrypt;
- a classic get function to retrieve the NTL ZZ object value.

```
61      //...
62      ZZ y, gcd, b1, b2;
63      do{
64          b1=0; b2=0;
65          do{
66              RandomBnd(y,N); //y=pseudo-random number in the range
                    0..N-1, or 0 if N <= 0
67              gcd=GCD(y,N);
68          }
69          while(gcd!=1);
70          b1 = Jacobi(y%p,p);
71          b2 = Jacobi(y%q,q);
72      }
73      while(b1!=-1 or b2!=-1);
74      //...
75      JL_13_EncodKey* encrypKey= new
            JL_13_EncodKey(N,y,k_message_bit_size);
76      JL_13_DecodKey* decrypKey= new
            JL_13_DecodKey(p,y,k_message_bit_size,p_prime);
77      ComputKey* computKey=NULL;
78      Keys* keys= new Keys(encrypKey,computKey,decrypKey);
79      return keys;
80  }
```

Figure 5.4: Part (c) of the `KeyGen` function inside the `JL_13_SchemeKeyGen` class.

The class `JL_13_Encoding` is the class that inherits the `Encoding` class of the framework. It will contain:

- a NTL ZZ object where store the value to decrypt;
- an additional constructor receiving as parameter directly a NTL ZZ object representing the value to decrypt (ciphertext);
- a classic get function to return the NTL ZZ object value;

`void PowerMod(ZZ& x, const ZZ& a, const ZZ& e, const ZZ& n)` and `void MulMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n)` are two NTL ZZ class routines among those performing arithmetic mod n, where $n > 1$. The first computes $x = a^e \mod n$ while the latter computes $x = (a * b) \mod n$. All arguments (other than exponents) are assumed to be in the range $0..n-1$. The `PowerMod` function checks this and raises an error if this does not hold.

```
1  Encoding* JL_13_SchemeEncode:: Encode(const EncodKey& encKey, const
       Message& mess)
2  {
3      //...
4      JL_13_Encoding* encoding=NULL;
5      ZZ m=(dynamic_cast<const JL_13_Message&>(mess)).to_ZZ();
6      ZZ x,gcd;
7      do{
8                 RandomBnd(x,N);
9          gcd=GCD(x,N);
10         }
11     while(gcd!=1);
12     ZZ y_to_the_m;
13     PowerMod(y_to_the_m,y,m,N);
14     ZZ x_to_the_two_to_the_k;
15     PowerMod(x_to_the_two_to_the_k,x,two_to_the_k,N);
16     ZZ c;
17     c=MulMod(y_to_the_m, x_to_the_two_to_the_k, N);
18         //...
19         encoding= new JL_13_Encoding(c);
20         return encoding;
21 }
```

Figure 5.5: Part of the `Encode` function inside the `JL_13_SchemeEncode` class.

Our implementation requires that the application generates a certain number of encodings (of random messages) that will be inserted in an object of the class `Comput_inputs`; such object will be than passed to the `Compute` function we will examine in the following. Fig. 5.6 shows the code inside a possible `main` application.

5.1.3 Computation

In this case, what we need is the `Compute` virtual function of the `SchemeCompute` “interface” of the framework. As always, one can create a concrete class, i.e. called `JL_13_SchemeCompute`, in which define a body for the `Compute` virtual function. There are no special observations to make about the computation (evaluation) step. Fig. 5.7 shows the code.

In this our implementation of JL, we have chosen to implement the composition of the two basic functions Add and CMult presented in the Eval algorithm. Namely,

```
1  int i;
2  int num_mess;
3  vector<JL_13_Message> messages;
4  for(num_mess=1;num_mess<=2;num_mess++){
5          JL_13_Message my_message(k_message_bit_size);
6          messages.push_back(my_message);
7  }
8          const JL_13_EncodKey* encodkey=dynamic_cast<const
               JL_13_EncodKey*>(keys->get_encodKey());
9          if (encodkey==0) {
10                 cout << "Null pointer on main: dynamic_cast of
                       EncodKey*\n";
11                 exit(EXIT_FAILURE);
12         }
13         Comput_inputs comput_inputs;
14         Encoding* cipher;
15         for(num_mess=0;num_mess<(int)messages.size();num_mess++){
16                 cipher=encrypt.Encode(*encodkey,
                       messages[num_mess]);
17               comput_inputs.append_Encoding(cipher);
18         }
19 ...
```

Figure 5.6: A portion of a possible `main` application.

a set given $\mathbf{a}=(a_1, ..., a_n)$ of n random elements belonging to the message space, and considering n ciphertexts, $\mathbf{c}=(c_1, ..., c_n)$ the function f we chose outputs $c_1^{a_1} \cdot c_2^{a_2}, ..., c_n a_n$.

We have to extend the `Program_function` class to contain the n coefficients a_i. The extended class will contain a vector of NTL ZZ elements representing the coefficients of the function and all the support methods to get information about the vector (i.e. size, elements indexes, etc.).

5.1.4 Decoding

In this case, what we need is the `Decode` virtual function of the `SchemeDecode` "interface" of the framework. As always, one can create a concrete class, i.e. called `JL_13_SchemeDecode`, in which define a body for the `Decode` virtual function, in particular the overloaded function `Decode` that does not take as parameters the `Program` and the `Label` objects. There are no special observations to make about

```
1  Encoding* JL_13_SchemeCompute:: Compute(const ComputKey& compKey,
       const Program_function& funct,const Comput_inputs& vec)
2  {
3          JL_13_Encoding* encoding=NULL;
4          ZZ N=(dynamic_cast<const JL_13_ComputKey&>(compKey)).get_N();
5          JL_13_Program_function function(dynamic_cast<const
               JL_13_Program_function&>(funct));
6          ZZ c(1);
7          int n;
8          ZZ c_i,fi,N;
9          int ciphers=vec.get_vector_size();
10         JL_13_Encoding* enc;
11         for(n=0;n<ciphers;n++){
12                 fi=function.get_coefficient(n);
13                 enc=(dynamic_cast<JL_13_Encoding*>(vec.get_Encoding(n)));
14                 c_i=enc->get_Ciphertext();
15                 PowerMod(power, c_i, fi, N);
16                 MulMod(c,c,power,N);
17         }
18         encoding = new JL_13_Encoding(c);
19         return encoding;
20 }
```

Figure 5.7: The `Compute` function inside the `JL_13_SchemeCompute` class.

the decoding (decryption) step. The code simply reflects the algorithm Alg. 3.7 showed in Sec. 3.3.2.4 where we have introduced our new decryption algorithms. Fig. 5.8 and Fig. 5.9 show such code.

5.2 Implementation of the $\mathsf{Outsourcing}_{lin}$ scheme

In this section I am going to present the implementation of the $\mathsf{Outsourcing}_{lin}$ scheme described in Sec. 3.3.3.2. Such implementation will allow to show a complete use of the framework where all the classes are employed.

Recall that the $\mathsf{Outsourcing}_{lin}$ scheme consists of all the possible phases: generation of the keys, encoding, computation and decoding. So, all the ABCs of the framework must be implemented.

```
1  Decoding* JL_13_SchemeDecode:: Decode(const DecodKey& decodKey, const
       Encoding& encod)
2  {
3      Decoding* decoding=NULL;
4          JL_13_DecodKey JL_decodKey(dynamic_cast<const
               JL_13_DecodKey&>(decodKey));
5          JL_13_Encoding JL_encoding(dynamic_cast<const
               JL_13_Encoding&>(encod));
6          ZZ p = JL_decodKey.get_p();
7          long k_message_bit_size = JL_decodKey.get_k_message_bit_size();
8          ZZ y = JL_decodKey.get_y();
9          ZZ p_prime=JL_decodKey.get_p_prime();
10         ZZ c = JL_encoding.get_Ciphertext();
11         ZZ m(0);
12         ZZ A(1);
13         ZZ B(1);
14         ZZ D;
15         PowerMod(D,operator%(y,p),operator-(p_prime),p);
16         int i,j;
17         std::vector<ZZ> Ds;
18         Ds.push_back(m); //just tu put 0 in the element of index 0
19         for(j=1;j<=k_message_bit_size-1;j++)
20         {
21             Ds.push_back(PowerMod(D,power2_ZZ(j-1),p));
22         }
23         //...
```

Figure 5.8: Part (a) of the `Decode` function inside the `JL_13_SchemeDecode` class.

5.2.1 Implementation of the amortized closed-form efficient pseudorandom function

The most interesting part of the implementation of the $\mathsf{Outsourcing}_{lin}$ is surely represented by the amortized closed-form efficient pseudorandom function (ACF-efficient PRF) described in Sec. 3.3.3.1.

The goal of the following section is to describe the realization of the generic PRF (without caring about the ACF-efficiency property) showing before the theoretical construction and then the practical implementation. As already touched on in the Chap.4, the PRF has been implemented using the Crypto++ library.

```
31          vector<ZZ> C;
32          C.push_back(PowerMod(operator%(c,p),p_prime,p));
33          for(j=1;j<=k_message_bit_size-1;j++)
34          {
35              C.push_back(PowerMod(C[j-1],2,p));
36          }
37          for(j=1;j<=k_message_bit_size-1;j++)
38          {
39              if((operator==(1,C[k_message_bit_size-j]))==0) //are
                    different
40              {
41                  operator+=(m,B);
42                  for(i=j+1;i<=k_message_bit_size;i++)
43                  {
44                      C[k_message_bit_size-i]=MulMod(C[k_message_bit_size-i],
                            Ds[k_message_bit_size-i+j],p);
45                  }
46                  }
47                  operator*=(B,2);
48          }
49          if((operator==(1,C[0]))==0) //if 1!=C[0]
50              operator+=(m,B);
51          JL_13_Message* mess= new JL_13_Message(m);
52          decoding= new Decoding(true,mess);
53          return decoding;
54  }
```

Figure 5.9: Part (b) of the `Decode` function inside the `JL_13_SchemeDecode` class.

The pseudorandom function theoretical construction

Pseudo random functions are a very powerful cryptographic tool. *Practical* PRFs typically have fixed input and output size, often roughly equal to the security parameter λ (say, both being 128 bits).

The first consideration about the PRF described in Sec. 3.3.3.1 is that we need a standard PRF mapping arbitrary-long strings into integers in $\mathbb{Z}_{p'q'}$; i.e. $F : \{0,1\}^* \to \mathbb{Z}_{p'q'}$.

To achieve the best efficiency, our goal is to realize the PRF using the *Advanced Encryption Standard* (AES) block cipher.

Let's assume therefore to have this *ideal* object AES (Fig. 5.10) that, given a

security parameter λ, uses a key of λ bits, takes an input of 128 bits and produces an output of 128 bits.

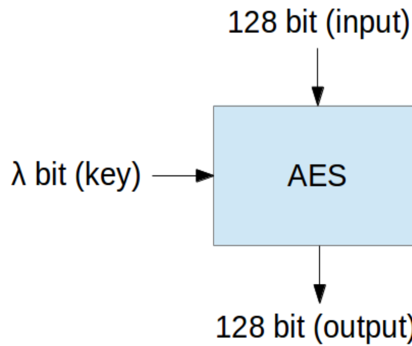

Figure 5.10: AES *ideal* object.

Due to the fact that our PRF, starting from a string of bit of *arbitrary length* ($\{0,1\}^*$), outputs an integer in $\mathbb{Z}_N$, specifically in $\mathbb{Z}_{p'q'}$, we need to solve the following two problems:

1. how to pass to our ideal AES block an arbitrary length input;
2. how to obtain as output an integer in $\mathbb{Z}_N$.

How to obtain an arbitrary length input

Formally, in order to solve the first issue (1), we will first realize a function $f_\lambda : \{0,1\}^* \rightarrow \{0,1\}^{128}$ (Fig. 5.11).

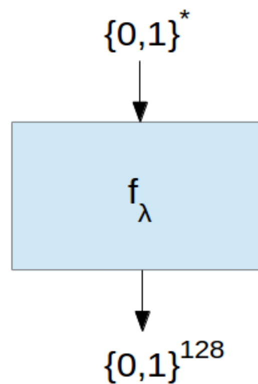

Figure 5.11: Function f_λ.

Dealing with inputs of arbitrary length can be easily achieved by employing a *cryptographic hash function*. Very informally, a hash function is any function that can be used to map digital data of *arbitrary size* to digital data of fixed size. The input data is often called the message, and the hash value is often called the *message digest* or simply the *digest*.

In our construction, the bit-size of the message digest is set equal to the *double* of the AES key bit-size, i.e. 2λ. This choice is related to an important property that a hash function has to have, namely the *collision resistance* property. Such property affirms that it should be difficult to find two different messages m_1 and m_2 such that $hash(m_1) = hash(m_2)$. Such a pair is called a cryptographic *hash collision*. This property is sometimes referred to as *strong collision resistance*. If we want to guarantee the security level given by λ we have to fix the digest bit-size equal to 2λ; otherwise collisions may be found by a birthday attack [1].

Formally what we are obtaining in this way is a function $h : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$ (Fig. 5.12); and this is different from what we wanted. Indeed we have solved the problem of the arbitrary length input but, on the other hand, the output of this new function h is $2\lambda > 128$, since $\lambda \geq 80$ in our setting.

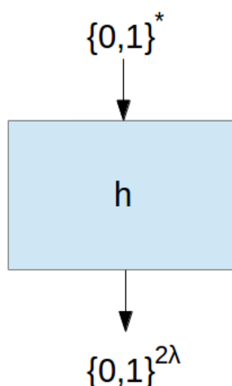

Figure 5.12: Function h.

This leads to the new question of how obtaining an output of 128 bits. This obstacle has been overcome by applying the AES block cipher in the *cascade mode*.

[2] Using cascade mode

[1] A birthday attack is a type of cryptographic attack that exploits the mathematics behind the birthday problem in probability theory. Refer to https://en.wikipedia.org/wiki/Birthday_attack for more details.

[2] This paragraph has been copied verbatim from [11].

By exploiting the cascade mode, we want to realize, in a secure way, a new PRF that starting from a string of 2λ bit, outputs a string of 128 bit, using our elementary PRF (Fig. 5.10).

We want to build a PRF function $f' : \{0,1\}^L \to \{0,1\}^l$ using a different PRF family $\{f_j\}$. Specifically, we do not care as much about the input size of f_j (but the large the better), let use call it b, but care that the output size is l and the key size k is at most l. Consider now a message m and split it into $m_1, ..., m_n$, except now each chuck is of size b, so that $L = b \cdot n$. The initial key j to f' is chosen at random from $\{0,1\}^l$, and then we inductively define values $x_0, ..., x_n \in \{0,1\}^l$ as follows:

$$x_0 = j$$
$$x_i = f_{x_i - 1}(m_i)$$

Finally the output $f'(m_1, ..., m_n) = x_n$ (Fig. 5.13). To describe it differently, $f_j(m_1)$ determines the PRF key x_1 to be used in the next round with input m_2, which in turn defines the PRF key x_2 to be used with the next input block m_3, and so on. This construction is called *cascade* or *Merkle-Damgard*. Notice, it really works for any input size $b \geq 1$, at the price of using L/b evaluations of the underlying PRF f_j (so larger b yields more efficiency).

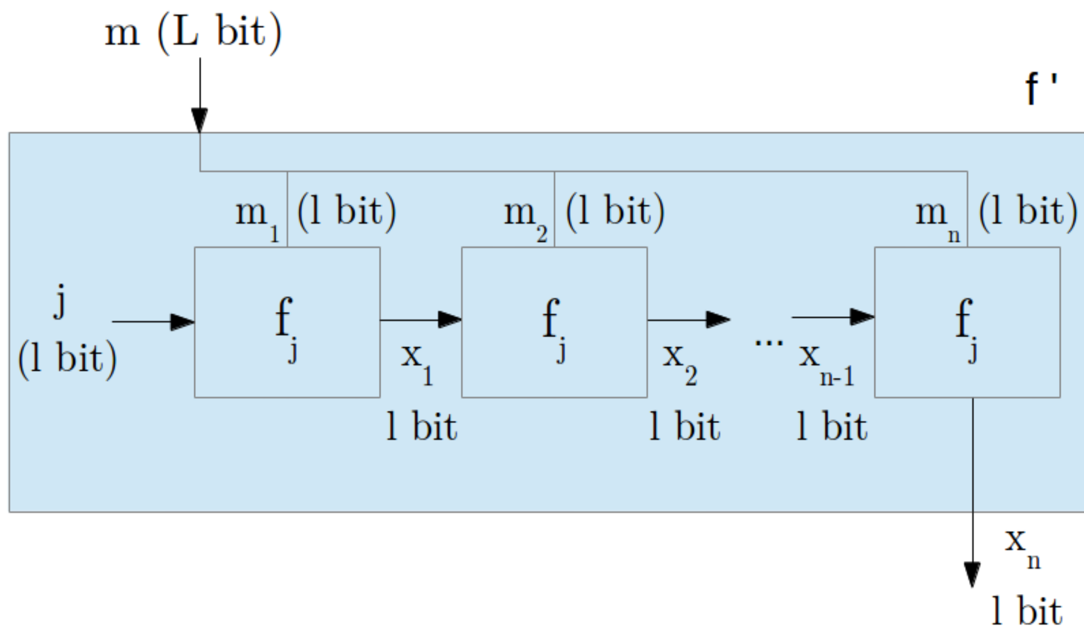

Figure 5.13: Cascade Mode.

The intuition behind this construction is quite simple. First, since all x_i's are PRF outputs, they are computationally indistinguishable from random. Second,

the very first block i separating two L-bit messages m and u would result in two computationally independent PRF keys x_i derived after the i-th call to f, and from this point on evaluating f' on m and u looks totally independent.

Theorem 5.1 *The cascade construction defines a PRF from L bits to l bits.* [3]

In our specific case the PRF f_t is represented by the AES block cipher, so the output size l is equal to 128 bit (the AES block cipher bit-size), the input size b is again 128, the L value is the message digest bit-size and the l value is again 128; the initial key t is the first element k_1 (k_2) of the seed K_1 (K_2), and it is still 128 bits. Formally, our function will be $f' : \{0,1\}^{2\lambda} \rightarrow \{0,1\}^{128}$ (Fig. 5.14).

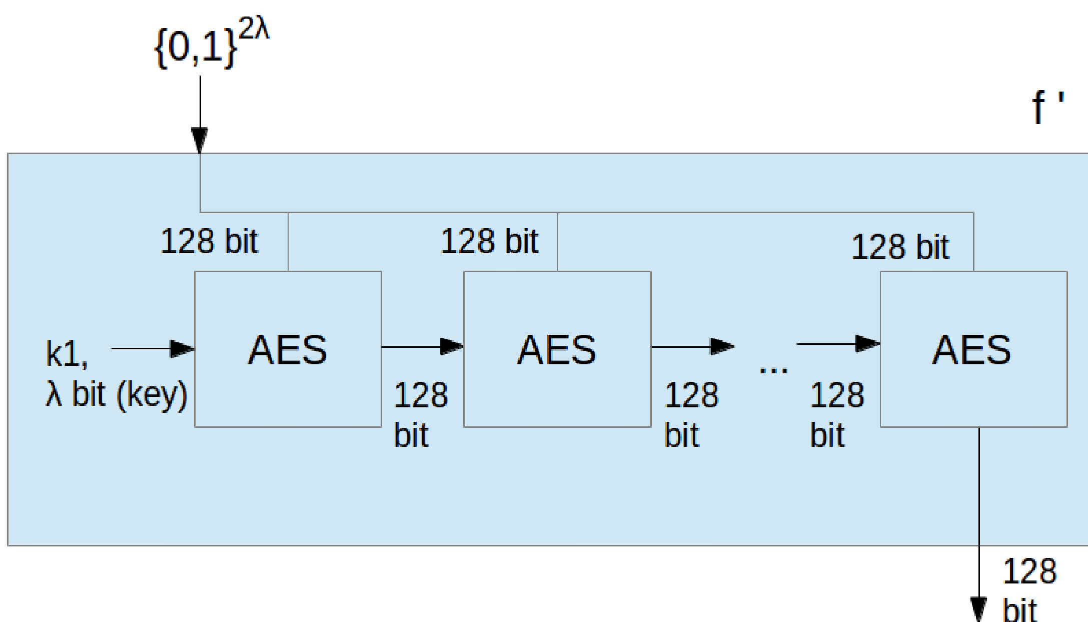

Figure 5.14: AES Cascade Mode in our construction.

So if the AES key bit-size is 128, e.g. in the case of the security parameter equal to 128 bit, we have that the hash function outputs a digest of 256 bit (there not exists a hash function with digest 160), so it will be split into two block of 128 bits; the first one will become the input of a first AES block cipher (with key k_1) that will output some intermediate 128 bits; such intermediate 128 bits will be the new key for a second AES block cipher that will take as input the second block of 128 bits of the message digest.

[3] We do not give the proof, based on a lemma that it is her not provided. The reader can refer to [11].

We have finally reach the goal. The initial function $f_\lambda : \{0,1\} * \lambda \rightarrow \{0,1\}^{128}$ can be realized as the composition of the two function $h : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$ and $f' : \{0,1\}^{2\lambda} \rightarrow \{0,1\}^{128}$ (Fig. 5.15).

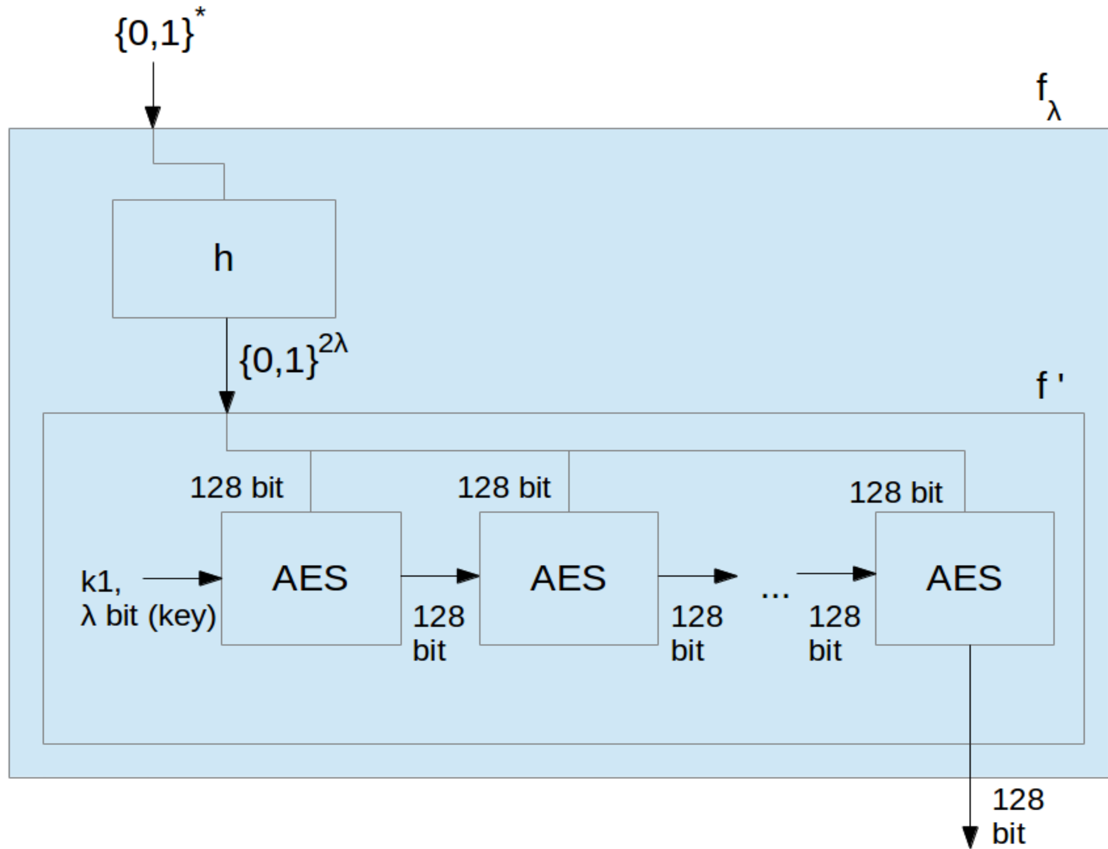

Figure 5.15: Construction of function f_λ.

How to obtain as output an integer in $\mathbb{Z}_N$

Let consider the general case of N, knowing that in the specific case of our PRF, $N = p'q'$. The final step is to obtain the output integer in $\mathbb{Z}_N$. Considering that in our application N is always larger than 2^{128}, we need to expand the 128 bits string obtained with f to an l bits string, for some $l > logN$; such string will then be converted to an integer, and finally we apply the arithmetic module operator. Formally, in order to solve such issue, we would like to define a function $g : \{0,1\}^{128} \rightarrow \{0,1\}^l$ (Fig. 5.16).

The bit-size of the product N will be always much greater than 128 bit. So we have to determine how the bit-size l has to be greater than N to securely apply the modular reduction operator. Formally, if $f(x)$ is the output of the function

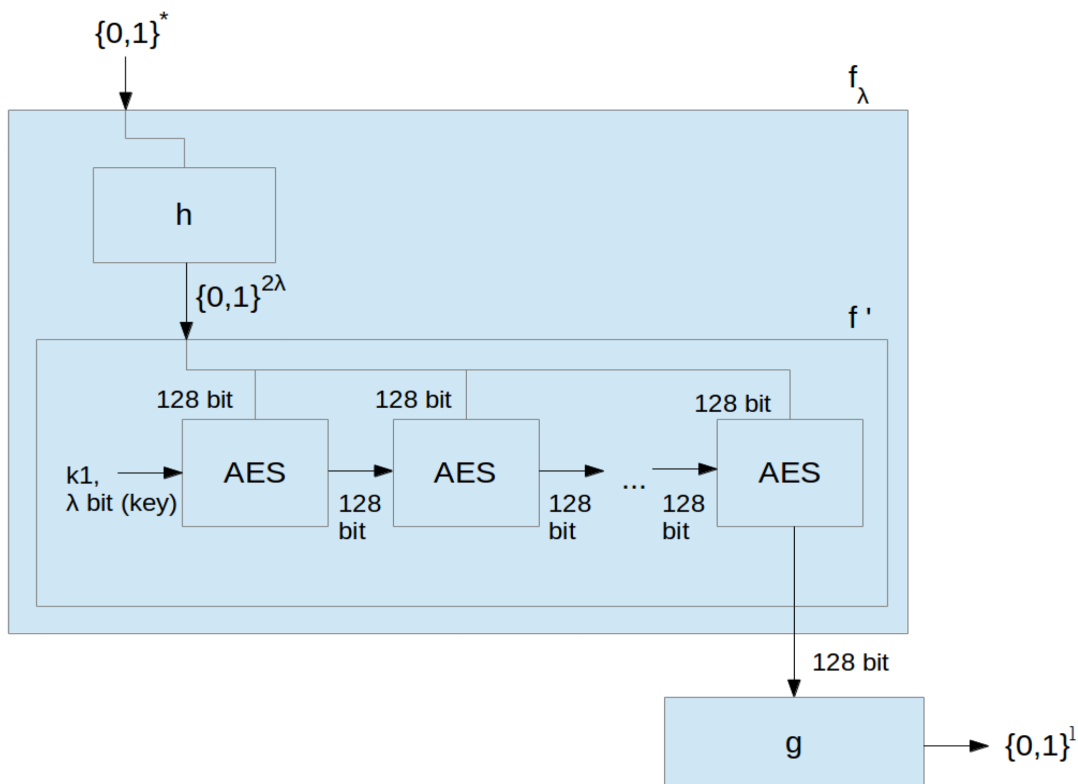

Figure 5.16: Function f_λ + function g.

previously built, applied on an arbitrary length input string x, and g is the function we are looking for, we would build our final function $F : \{0,1\}^* \rightarrow \mathbb{Z}_N$ as $F(x) = g(f(x)) \bmod N$.

HOW TO DETERMINE THE l OUTPUT BIT-SIZE

As a first step we first study what is a good value of l that maintains security. Let us first recall the notion of *Statistical Distance.*

Definition 5.2 (Statistical Distance). *Let X, Y be two random variables over a finite set $\mathcal{U}$. The statistical distance between X and Y is defined as*

$$\mathsf{SD}[X,Y] = \frac{1}{2}\sum_{u\in\mathcal{U}}\Big|Pr[X = u] - Pr[Y = u]\Big|$$

The explanation of why we chose this measure follows. Very informally, considering $\mathbb{Z}_N$, assuming that $\mathcal{R} = \{0,1\}^l$ is the range of the PRF and that $y \leftarrow_{\mathcal{R}} \{0, ...,2^l - 1\}$ is randomly chosen, we have to guarantee that the element $z = y \bmod N$ is chosen with the same distribution with which we pick an element $z \leftarrow_{\mathcal{R}} \mathbb{Z}_N$.

Formally,

Lemma 5.1 *Let consider N and l, with $l, N \in \mathbb{N}$. Let be $N = 2^{l_N}$ and $l \geq l_N$. Consider the following two experiments: Experiment* $\mathbf{ExpA}(N)$
$z \leftarrow_{\mathcal{R}} \mathbb{Z}_N$

Experiment $\mathbf{ExpB}(N, l)$
$y \leftarrow_{\mathcal{R}} \{0, ..., 2^l - 1\}$; $z = y \bmod N$.

Then

$$\mathsf{SD}[A, B] = \frac{1}{2}\sum_{x=0}^{N-1} \left| Pr[z = x | z \leftarrow \mathbf{ExpA}(N)] - Pr[z = x | z \leftarrow \mathbf{ExpB}(N, l)] \right| = \frac{2^{l_N}}{2 \cdot 2^l}$$

Proof
Let define $C = 2^l$.

$$\mathsf{SD}[A, B] = \frac{1}{2}\sum_{x=0}^{N-1} \left| \frac{1}{N} - \left(\frac{1}{C} + \frac{C/N}{C} \right) \right| = \frac{1}{2}\sum_{x=0}^{N-1} \left| \frac{1}{N} - \frac{1}{C} - \frac{1}{N} \right| = \frac{N}{2C} = \frac{2^{l_N}}{2 \cdot 2^l}$$

Corollary 5.1 *Let consider a security parameter λ and let* $\mathbf{ExpA}(\lambda, N)$ *and* $\mathbf{ExpB}(\lambda, N, l)$ *two experiments as defined in* Lemma 5.1. *Let be $l - l_N + 1 \geq \lambda$. Then, from the* Lemma 5.1 *follows that* $\mathsf{SD}[A, B] =$
$$\frac{1}{2}\sum_{x=0}^{N-1} \left| Pr[z = x | z \leftarrow \mathbf{ExpA}(\lambda, N)] - Pr[z = x | z \leftarrow \mathbf{ExpB}(\lambda, N, l)] \right| \leq \frac{1}{2^\lambda} = \mathsf{negl}(\lambda)$$

Therefore, using the above result, the bit-size of the PRF output can be easily determined by adding the security parameter λ to the bit-size of the number N, i.e. set $l = l_N + \lambda$.

Now that we know the output bit-size length l, we have to construct the function g. To this end we use the block cipher AES in *Counter mode* (CNT).

CTR uses a running counter block. Each block of plain text is encrypted independently, rather than with the results of a previous set of rounds. The counter includes a nonce and an initial counter block. The Key and Counter are encrypted, and then the result is XOR'd with the plain text (Fig. 5.17). The mode does not require padding the plain text to the block size of the cipher.

The counter has the size of the cipher's block size. In the case of default AES, this would be 16 bytes. NIST Special Publication 800-38A [?] specifies two methods for using the CTR mode. The first is a counter which is made up of a nonce and counter. The nonce is random, and the remaining bytes are counter bytes (which are incremented). For example, a 16 byte block cipher might use the high 8 bytes as a nonce, and the low 8 bytes as a counter; the second method uses the entire block cipher size (16 bytes in the case of AES) as a monotonically increasing value. If we

inspect the Crypto++ library source code (`modes.h`), we can see that the library uses this method, which means the entire byte block is treated as counter bytes.

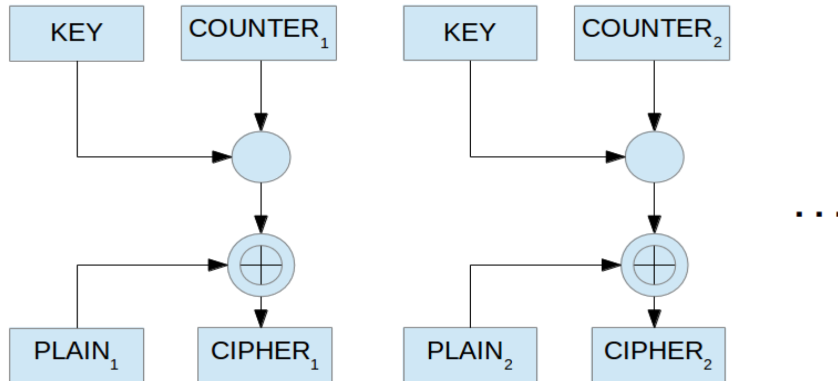

Figure 5.17: CTR Mode.

Basically, through the use of the CTR mode we would to obtain a Pseudorandom Number Generator (PRG). A PRG is a deterministic polynomial-time computable function $G : \{0,1\}^k \rightarrow \{0,1\}^{p(k)}$ (defined for all $k > 0$) that *stretches* a short random seed $x \in \{0,1\}^k$ into a longer output $G(x)$ of length $p(k) > k$ which nevertheless *looks* like a random $p(k)$-bit strings to any computationally bounded adversary (the *distinguisher*).

In our construction, the counter will be the output of the function f, the key will be the second element k'_1 (k'_2) of the seed K_1 (K_2); the plaintext will be an empty message (all 0s bit) of bit-size equal to the PRF output bit-size. At the end, the l bits produced by the CTR mode will be the final output; they will be converted in an integer.

As a last comment we have to say that in the Crypto++ library there not exists the ideal object AES described in Fig.5.10. In `Crypto++` When we use symmetric ciphers, we have to choose a cipher and a mode of operation. The cipher chosen has been the AES while the mode has been the Electronic Cookbook (ECB). ECB is the simplest of the methods. A message is broken into blocks, and each block is combined with the key (Fig. 5.18).

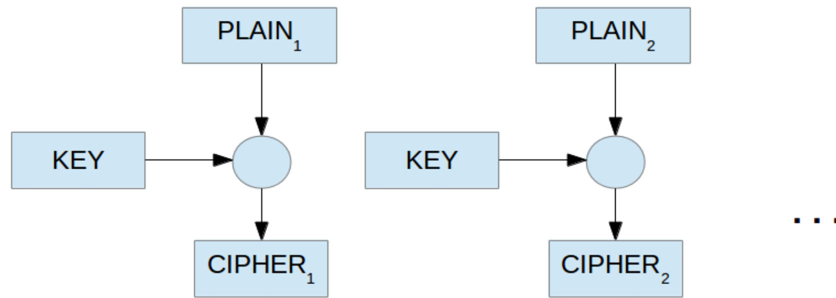

Figure 5.18: ECB Mode.

Fig. 5.19 shows the complete construction scheme of the PRF.

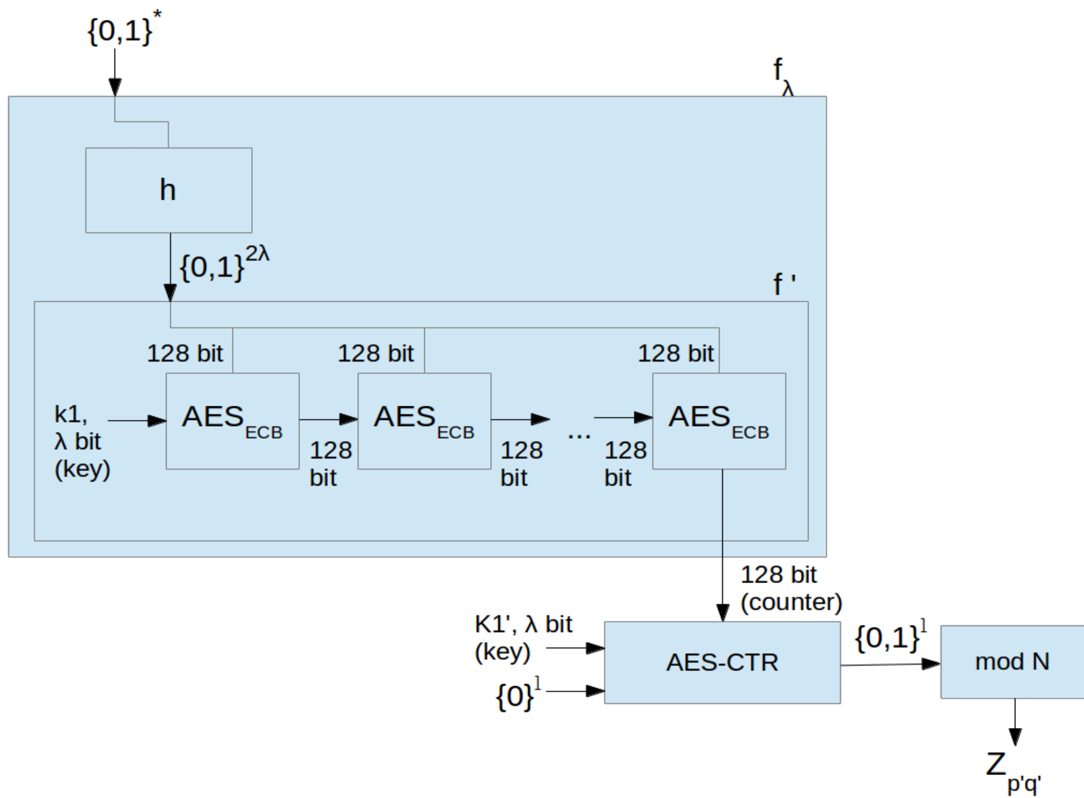

Figure 5.19: Complete construction scheme of the PRF.

Implementation of the amortized closed-form efficient pseudorandom function

The part of the implementation about the PRF consists of 4 classes:

- PRF;

- PRF_parameters;

- PRF_seeds;

- PRF_keys.

The PRF_seeds class (Fig. 5.20) represents the two seeds K_1 and K_2 of the ACF-efficient PRF; it contains:

- two C++ pair elements, each one representing a seed (K_1 and K_2) and containing a couple of array of bytes ($\{k_1, k'_1\}$ for K_1 and $\{k_2, k'_2\}$ for K_2);
- an additional constructor receiving as parameter such pair elements;
- a classic get function for each seed (pair) to retrieve.

```
1  class PRF_seeds{
2  public:
3      PRF_seeds();
4      PRF_seeds(const pair<const byte*,const byte*>& K1_, const
          pair<const byte*,const byte*>& K2_);
5      PRF_seeds(const PRF_seeds& copy);
6      PRF_seeds& operator= (const PRF_seeds& copy);
7      ~PRF_seeds();
8      pair<const byte*,const byte*> get_K1() const;
9      pair<const byte*,const byte*> get_K2() const;
10 private:
11     pair<const byte*,const byte*> K1;
12     pair<const byte*,const byte*> K2;
13 };
```

Figure 5.20: PRF_seeds class

.

The PRF_parameters class (Fig. 5.21) represents the additional parameters required by the ACF-efficient PRF to execute the computation; recall from the Sec. 3.3.3.1 that the ACF-efficient PRF needs also other parameters like the primes p and q, the product N, the bit-size k of every message to encode, the bit-size of the AES key, the bit-size of the message digest, etc.; therefore it contains:

- all such necessary parameters (almost all NTL ZZ objects plus some `int` or `long` elements);
- an additional constructor receiving as parameter such elements;
- a classic get function for each value to retrieve.

```
1  class PRF_parameters{
2  public:
3      PRF_parameters();
4      PRF_parameters(long RSA_sec_param_, const ZZ& N_, const ZZ&
          p_prime_,const ZZ& q_prime_, const ZZ& p_, const ZZ& q_, const
          ZZ& two_to_the_k_,const ZZ& g_, int aes_key_byte_size_, int
          sha3_digest_byte_size_);
5      PRF_parameters(const PRF_parameters& copy);
6      PRF_parameters& operator= (const PRF_parameters& copy);
7      ~PRF_parameters();
8      long get_RSA_sec_param() const;
9      ZZ get_N() const;
10     ZZ get_p_prime() const;
11     ZZ get_q_prime() const;
12     ZZ get_p() const;
13     ZZ get_q() const;
14     ZZ get_two_to_the_k() const;
15     ZZ get_g() const;
16     int get_aes_key_byte_size() const;
17     int get_sha3_digest_byte_size() const;
18 private:
19     long RSA_sec_param;
20     ZZ N, p_prime,q_prime,p,q,two_to_the_k,g;
21     int aes_key_byte_size, sha3_digest_byte_size;
22 };
```

Figure 5.21: `PRF_parameters` class

.

The `PRF_keys` class (Fig. 5.22) is somehow a wrapper class that contains:

- two objects: a `PRF_seeds` object and a `PRF_parameters` object;
- an additional constructor receiving as parameter such two objects;
- a classic get function for each object to retrieve.

```
1  class PRF_keys{
2  public:
3      PRF_keys();
4      PRF_keys(const PRF_parameters& prf_params,const PRF_seeds&
           prf_seeds_);
5      PRF_keys(const PRF_keys& copy);
6      PRF_keys& operator= (const PRF_keys& copy);
7      ~PRF_keys();
8      PRF_seeds get_PRF_seeds();
9      PRF_parameters get_PRF_parameters() const;
10     PRF_seeds get_PRF_seeds() const;
11 private:
12     PRF_parameters prf_params;
13     PRF_seeds prf_seeds;
14 };
```

Figure 5.22: `PRF_keys` class

.

The reason for which it has been choice this configuration is that, in general, one can choose to use a `PRF` object without caring about the generation of the seeds and/or of the other parameters; one can assume that maybe there exists someone in charge of output all the necessary keys, parameters, and than to pass them to the `PRF` object. In other words, if an user wants to employ the `PRF`, can decide to generate all the keys necessary by itself or just demand the task to someone else. Moreover, the composed `PRF_keys` object allows to demand the generation of only one part of the components (`PRF_seeds` or `PRF_parameters`) required by the `PRF` object or of both parts. Once again, the goal is to increase the modularity.

An object of the `PRF` class (Fig. 5.23) represents an ACF-efficient PRF, so it contains a `compute` method that is the method that executes the $\mathsf{F}'_{K_1}(\tau)$ and $\mathsf{F}'_{K_2}(\Delta)$ of the description in Sec. 3.3.3.1. `PRF` contains the constructor that receives a `PRF_keys` object, i.e. the parameters and the seeds, but it also contains an additional constructor that receives only a `PRF_parameters` object; in this case inside such constructor will be built the seeds. In order to guarantee the efficiency property, the `PRF` provides also two methods `CFEval_off` and `CFEval_on` that carry out the rule of the two equivalent methods seen always in Sec. 3.3.3.1.

In the following paragraphs I will describe all the methods of the `PRF` class, showing the parts of the code more significant.

ACF-efficient PRF keys generation

```
1  class PRF{
2  public:
3      PRF();
4      PRF(const PRF_keys& prf_keys_); //keys = paramaeters + seeds for aes
5      PRF(const PRF_parameters& prf_params); //only parameters, the
           constructor builds the seeds
6      PRF(const PRF& copy);
7      PRF& operator= (const PRF& copy);
8      ~PRF();
9      ZZ compute(const Label& delta, const Label& tau);
10     ZZ CFEval_off(const Program& program);
11     ZZ CFEval_on(const Label& delta, const ZZ& w_f);
12 private:
13     ZZ N,p_prime,q_prime,p,q,two_to_the_k,g;
14     long RSA_sec_param;
15     PRF_seeds prf_seeds;
16     long PRF_output_byte_size;
17     const byte* k1, k1_prime, k2, k2_prime;
18     int aes_key_byte_size;
19     int sha3_digest_byte_size;
20     SHA3_256 hash, kash2;
21     ECB_Mode< AES >::Encryption encryptor, encryptor3;
22     CTR_Mode< AES >::Encryption encryptor2, encryptor4;
23     bool destroy;
24 };
```

Figure 5.23: `PRF_keys` class

.

The Fig. 5.24 shows part of the code of the generation of the keys of the ACF-efficient PRF inside one of the constructor of the `PRF` class. Actually, the code shows just the generation of the seeds, because the parameters are generated inside the `KeyGen` function of the framework, as we will see later on.

The code is quite self-explanatory; there is a switch to determine the bit-size of the AES key and of the message digest of the hash function, depending on the security parameter. Regarding the hash function, it has been chosen the SHA-3 (Keccak)[4], added in the version 5.6.2 of `Crypto++`.

[4] https://en.wikipedia.org/wiki/SHA-3

```
1  //...
2  const byte* k1;
3  const byte* k1_prime;
4  const byte* k2;
5  const byte* k2_prime;
6  AutoSeededRandomPool xRng;
7  switch(RSA_sec_param){
8          case(80): //aes128
9                  aes_key_byte_size=16;
10             sha3_digest_byte_size=32;
11             break;
12         case(128): //aes128
13                 aes_key_byte_size=16;
14             sha3_digest_byte_size=32;
15             break;
16         case(256): //aes256
17             //...
18         default: //aes128
19             //...
20 }
21 //building seeds
22 k1 = new byte[aes_key_byte_size];
23 k1_prime= new byte[aes_key_byte_size];
24 k2 = new byte[aes_key_byte_size];
25 k2_prime = new byte[aes_key_byte_size];
26 xRng.GenerateBlock(const_cast<byte*>(k1), aes_key_byte_size);
27 xRng.GenerateBlock(const_cast<byte*>(k1_prime), aes_key_byte_size);
28 xRng.GenerateBlock(const_cast<byte*>(k2), aes_key_byte_size);
29 xRng.GenerateBlock(const_cast<byte*>(k2_prime), aes_key_byte_size);
30 PRF_output_byte_size=NumBytes(operator*(p_prime,
       q_prime))+(RSA_sec_param/8);
31 destroy=true;
32 //...
```

Figure 5.24: Part of the constructor of the `PRF` class that receives only the `PRF_parameters` as parameter.

The code in Fig. 5.24 randomly fills the byte arrays (seeds) using a `RandomNumberGenerator` of `Crypto++` that is `AutoSeededRandomPool` and that is

an `AutoSeeded` generator, i.e. it does not require a seed.[5]. Than the bit-size of the ACF-efficient PRF output is set following the Lemma 5.1. The last code line is just a simple solution to avoid C++ memory problems; indeed in the case in which the above constructor is called, the C++ `new` operator is called several times to create the seeds; therefore we need to set this boolean flag `destroy` so that inside the destructor we can correctly *free* the memory. This artifice is necessary because in the case in which the other `PRF` class constructor (that receives all the keys, so also the seeds) is called, there is no memory to free (because the C++ `new` operator is not employed).

ACF-efficient PRF `compute()` function

The `Compute()` function essentially implements the $\mathsf{F_K}(\Delta, \tau)$ seen in Sec. 3.3.3.1. This method takes as parameters two `Label` objects, Δ and τ. The `Compute()` function inside the `PRF` class describes the implementation of the theoretical construction of the ACF-efficient PRF seen in the previous section, considering the case of the security parameter equal to 80. The Fig. 5.25 shows the part of the code of the `Compute()` function related to the hash function.

```
1  SHA3_256 hash;
2  //input string for the hashing
3  string delta_string=delta.to_string();
4  //output of the hashing and input of aes
5  byte aes_input_bytes[sha3_digest_byte_size];
6  hash.CalculateDigest(aes_input_bytes, (byte*) delta_string.c_str(),
       delta_string.length());
```

Figure 5.25: Hashing of the the string Δ

The `hash` variable is an object of the `Cryptop++ SHA3_256` class and it outputs the message digest. After that, there is the AES cascade mode.

The cascade method has been realized by implementing a for cycle within which it is called the function `void cascade_mode(const byte* key, byte* input, byte* output)` Fig. 5.26; this function takes as parameters the key, the input bytes and the output bytes of each round of the cascade mode.

[5] Refer to `http://www.cryptopp.com/wiki/RandomNumberGenerator` for more details

```
1  void PRF:: cascade_mode(const byte* key, byte* input, byte* output)
2  {
3          ECB_Mode< AES >::Encryption encryptor;
4          encryptor.SetKey(key, aes_key_byte_size);
5          encryptor.ProcessData(output,input,AES::BLOCKSIZE);
6  }
```

Figure 5.26: AES cascade mode (ECB mode)

In `Cryptop++`, when we use symmetric ciphers, we have to choose a cipher and a mode. When we choose a mode, we can use it in one of two ways in Crypto++. The first method uses the cipher as a templated parameter. The second method uses external cipher objects. They are quite equivalent; in the Fig. 5.26, the block cipher (AES) is a templated parameter to the mode object (ECB) - it holds an instance of the cipher object. Next, we set the key and we push the bytes block in for encryption. Finally we use `ProcessData()` to encrypt the plain text, since it allows us to receive the result in a single line of code.

Even the counter mode has been implemented by using another function, `string ctr_mode(const byte* key, const byte* final_16_byte_aes_output)` Fig. 5.27, called by the `compute` function. It takes as parameters the key (second value of the pair K_1 and K_2) and the input for the CTR mode that is the output of the cascade mode, and it returns the final string to convert to the NTL ZZ integer.

The `ctr_mode()` function code in Fig. 5.27 deserves a little extra explanation.

Block ciphers, such as DES and AES, can be made to appear like a *stream cipher* if we use a Crypto++ adapter called a `StreamTransformationFilter`. We do this because the filter handles buffering, blocking, and padding for us. In short, it makes it easier to use the library. Crypto++ uses a Unix pipelining paradigm: data flows from source to a destination.

Above, we start with a `StringSource`, and end with a `StringSink`. The filters in between perform the buffering, blocking, and padding. If we were to visualize the block diagram of a system using a `StreamTransformationFilter`, it would be as shown in figure Fig. 5.28. Note that the `BufferedTransformation` argument to the `StreamTransformationFilter` is the Encryptor object. `BufferedTransformation` is the base class of the Encryptor and the Decryptor.

```
1  string PRF:: ctr_mode(const byte* key, const byte*
       final_16_byte_aes_output)
2  {
3          CTR_Mode< AES >::Encryption encryptor;
4          //AES Counter mode to obtain the final number
5          byte counter [AES::BLOCKSIZE];
6          memcpy (counter, final_16_byte_aes_output,
               sizeof(byte)*AES::BLOCKSIZE);
7          string message;
8          message.assign(PRF_output_byte_size,'0');
9          encryptor.SetKeyWithIV(key, aes_key_byte_size, counter);
10         string output; //the output bytes of the pseudorandom function
11         StringSource(message, true, new
               StreamTransformationFilter(encryptor,new
               StringSink(output)));
12         return output;
13 }
```

Figure 5.27: AES Counter mode

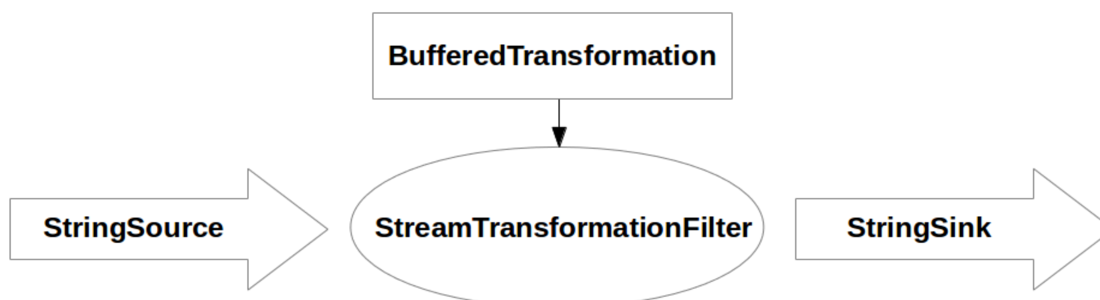

Figure 5.28: StreamTransformationFilter data flow.

Finally, the AES CTR mode output is converted in a NTL ZZ object by using the NTL ZZ function; by applying the module operator one finally obtains the output of the ACF-efficient PRF.

Inside the `compute()` function, the sequence of functions `cascade_mode-ctr_mode` is called for both strings, *delta* and τ.

`CFEval_off()` FUNCTION of the ACF-EFFICIENT PRF

The `CFEval_off()` function is the implementation of the $\mathsf{CFEval}_{\tau}^{\mathsf{off}}(K, f)$ algorithm presented in the description of the ACF-efficient PRF in Sec. 3.3.3.1. The

function takes as parameter a `Program` object. Fig. 5.29 shows the code of the implementation. First it is recovered the number of coefficients of the function (inside the `Program`) that has to be computed; than, for each τ_i inside the `Program` the ACF-efficient PRF is computed as seen before and the function (`w_f`) is computed over all the PRF values (the function is a linear combination so it is enough to multiply every coefficient of the function for the correspondent value outputted by the ACF-efficient PRF).

```
1  ZZ PRF:: CFEval_off(const Program& program){
2          ZZ w_f(0); //value to return
3          FGP_14_lin_funct_Program_function funct(dynamic_cast<const
              FGP_14_lin_funct_Program_function&>(program.get_function()));
4          int coefficients=funct.get_coefficients_number();
5          int n;
6          ZZ vi; //output of the pseudorandom function
7          for(n=0;n<coefficients;n++){
8                  //computing the PRF for each string "tau_i" inside
                      "program"
9                  //as seen in the Compute() function
10                 string output; //output of the PRF
11                 ZZ vi;
12                 ZZFromBytes(vi, (unsigned char*)output.c_str(),
                      output.size());
13                 operator%=(vi, operator*(p_prime, q_prime));
14                 AddMod(w_f, w_f,
                      MulMod(vi,funct.get_coefficient(n),operator*(p_prime,
                      q_prime)), operator*(p_prime, q_prime));
15         }
16         return w_f;
17 }
```

Figure 5.29: `CFEval_off()` function inside the `PRF` class

.

`CFEval_on()` FUNCTION of ACF-EFFICIENT PRF

The `CFEval_on()` function is the implementation of the $\mathsf{CFEval}^{\mathsf{on}}_{\Delta}(K, \omega_f)$ algorithm presented in the description of the ACF-efficient PRF in Sec. 3.3.3.1. The function takes as parameter a `Label` object and the output of the `CFEval_off()` function. Fig. 5.30 shows the code of the implementation. The function simply computes the ACF-efficient PRF of the string Δ, b and than use such value to return $W = g^{w_f \cdot b}$ (see Sec. 3.3.3.1 to recall what g is).

```
1  ZZ PRF:: CFEval_on(const Label& delta,const ZZ& w_f){
2          ZZ W(0); //value to return
3          //computing the PRF of "delta"
4          //as seen in the Compute() function
5          string output; //output of the PRF
6          ZZ b;
7          ZZFromBytes(b, (unsigned char*)output.c_str(), output.size());
8          operator%=(b, operator*(p_prime, q_prime));
9          PowerMod(W, g, operator*(w_f,b), N);
10         return W;
11 }
```

Figure 5.30: `CFEval_on()` function inside the `PRF` class

.

5.2.2 Implementation of the scheme's algorithms

As already made for the Joye-Libert cryptosystem implementation, I will show which classes of the framework have been extended and which simply used as they are. For the implementation of the $\mathsf{Outsourcing}_{lin}$ (Sec. 3.3.3.2) the framework has been employed in all its totality.

Starting from the classes representing the keys, `ComputKey`, `DecodKey`, cmdEncodKey and `OnDecodKey` have been of course all extended; we can call the derived classes `FGP_14_lin_funct_EncodKey`, `FGP_14_lin_funct_ComputKey`, `FGP_14_lin_funct_DecodKey` and `FGP_14_lin_funct_OnDecodKey`. Each of them basically contains all the parameters required to execute the correspondent algorithms, so i.e. the `FGP_14_lin_funct_ComputKey` will contain just an NTL ZZ object representing the composite N, while the `FGP_14_lin_funct_ComputKey` will contain the k message bit-size, the N, y, α parameters (all NTL ZZ objects) and a `PRF_keys` parameter (recall that the `Encode` algorithm of the scheme needs to use the ACF-efficient PRF); all the above classes include as always an additional constructor to set all the necessary parameters and a typical `get` function for each variable to retrieve.

The other classes of the framework that have been inherited are `Encoding`, `Message` and `Program_function`. We can call the derived classes `FGP_14_lin_funct_Encoding`, `FGP_14_lin_funct_Message`, `FGP_14_lin_funct_Program_function`.

The `FGP_14_lin_funct_Encoding` represents and `encoding` element and, following the scheme in Sec. 3.3.3.2, it consists of a couple of element, a ciphertext and

a tag, that becomes a couple of NTL ZZ objects; the class provides the additional constructor to set these objects and the classic `get` function to recover them.

The `FGP_14_lin_funct_Message` is sostantially the same of the Joye-Libert cryptosystem implementation. It contains an NTL ZZ object where to store the value to encrypt, an additional constructor receiving as parameter directly a NTL ZZ object representing the value to encrypt, an additional constructor receiving as parameter a `long` representing the bit size of the integer to encrypt (in this case, inside the constructor it will be called an other function of the class to generate a random NTL ZZ integer of the right size representing the value to encrypt) and a classic `get` function to retrieve the NTL ZZ object value.

The `FGP_14_lin_funct_Program_funcion` contains the linear function in the form of a `vector` of coefficients (NTL ZZ objects). Actually the additional constructor takes as parameter also an `int` element representing the number of τ_i labels contained in the `Program`; the reason is that when we pass to the constructor the `vector` of coefficients of the function we perform a check to verify that the number of coefficients is equal to the number of τ_i. The class provides several utility functions to retrieve the entire function (`vector` of NTL ZZ objects), only a coefficient (NTL ZZ object) or just a vector coefficient index (`int`).

The $\mathsf{Outsourcing}_{lin}$ key generation

The first step is to implement the KeyGen algorithm of the $\mathsf{Outsourcing}_{lin}$ scheme. As already made for the Joye-Libert(JL) cryptosytem implementation, what we need is to implement the `KeyGen` virtual function of the `SchemeKeyGen` “interface” of the framework. For this reason, one can create a concrete class, i.e. called `FGP14_lin_funct_SchemeKeyGen`, in which define a body for the `KeyGen` virtual function.

Regarding this class, there is actually not much to say. Reviewing the KeyGen algorithm in Sec. 3.3.3.2, we can see that many of the parameters generate by this algorithm are essentially the same for the $\mathsf{Outsourcing}_{lin}$ scheme and for the ACF-efficient PRF (a part from the seeds of the ACF-efficient PRF); moreover the parameters of the $\mathsf{Outsourcing}_{lin}$ scheme are almost the same of the Joye-Libert keys generation. Actually, there are only two differences: in the $\mathsf{Outsourcing}_{lin}$ the two primes p and q are k-quasi-safe primes while in JL, $p,q \equiv 1 \pmod{2^k}$; this implies a little modification in the generation algorithms of p and q; the other difference is represented by the generator g not present in JL. Therefore, to obtain the code of the `KeyGen` inside the `FGP14_lin_funct_SchemeKeyGen` class, it is enough to consider the key generation code described for the Joye-Libert keys generation implementation and simply modify it (Fig. 5.31 and Fig. 5.32) and add the code showed in the ACF-efficient PRF key generation to build the seeds.

```
1         long bit_size_p_q_prime =(N_factoring_sec_par_bit_size/2) -
             k_message_bit_size;
2         ZZ p,p_prime;
3         GenPrime(p_prime, bit_size_p_q_prime, RSA_sec_param);
4         while(!ProbPrime(two_to_the_k*p_prime+1,miller_rabin_tests)){
5                 GenPrime(p_prime, bit_size_p_q_prime, RSA_sec_param);
6     };
7         p=two_to_the_k*p_prime+1;
8         ZZ q,q_prime;
9         GenPrime(q_prime, bit_size_p_q_prime, RSA_sec_param);
10    while(!ProbPrime(two_to_the_k*q_prime+1,miller_rabin_tests)){
11                GenPrime(q_prime, bit_size_p_q_prime, RSA_sec_param);
12    };
13        q=two_to_the_k*q_prime+1;
```

Figure 5.31: k-quasi-safe primes generation

.

How it is possible see from the Fig. 5.31, here, instead of using the `RandomLen` function of NTL ZZ, we use the `void GenPrime(ZZ& n, long l, long err);` NTL ZZ function because we need that p' and q' are also primes. Indeed, such function generates a random prime n of length l so that the probability that the resulting n is composite is bounded by 2^{-err}.

In the Fig. 5.32, the algorithm picks a random element $x \in Z_N^*$, then computes $y = x^{2^k}$ and evaluates $t_1 = y^{p'}$, $t_2 = y^{q'}$; if $t_1 \neq 1$ and $t_2 \neq 1$ we go out from the while and we set the generator $g = x^{2^k}$, otherwise we continue the research.

The procedure remains the same: the `KeyGen` virtual function has the task of generating all the keys for the encoding, computing and decoding phase, using them to build the `Keys` object that will be returned by the function. Fig. 5.33 shows just the last part of the code in which the keys are built and stored in the `Keys` object.

The $\mathsf{Outsourcing}_{lin}$ offline decoding

If we consider the description of the $\mathsf{Outsourcing}_{lin}$ scheme in presence of the ACF-efficiency property, the only different thing in the KeyGen algorithm consists of the use of the offline closed-form efficient algorithm of the PRF. Recalling the structure of the framework, the ACF-efficiency property has been modelled using the two ABCs `SchemeOfflineDecode` and `SchemeOnlineDecode`; so we need to implement the `OfflineDecode` virtual function of the `SchemeOfflineDecode` "interface" of the framework. For this reason, one can create a concrete class,

```
1          ZZ x,x_two_k,x_two_k_p_prime,x_two_k_q_prime,g;
2          do{
3              x_two_k_p_prime=0;
4              x_two_k_q_prime=0;
5          do{
6                  RandomBnd(x,N);
7                  gcd=GCD(x,N);
8          }
9          while(gcd!=1);
10             PowerMod(x_two_k,x,two_to_the_k,N);
11             PowerMod(x_two_k_p_prime,x_two_k%p_prime,p_prime,p_prime);
12             PowerMod(x_two_k_q_prime,x_two_k%q_prime,q_prime,q_prime);
13         }
14         while(x_two_k_p_prime==1 or x_two_k_q_prime==1);
15     PowerMod(g,x,two_to_the_k,N);
```

Figure 5.32: Generation of the generator g of the subgroup of $\mathbb{Z}_N^*$,
$\mathcal{R}_k = \{x^{2^k} : x \in Z_N^*\}$

.

i.e. called `FGP14_lin_funct_SchemeOfflineDeode`, in which define a body for the `OfflineDecode` virtual function. Inside such function it will be sufficient to create a `PRF` object passing it the `PRF_keys` (contained in the `FGP_14_lin_funct_DecodKey`) and then call the `CFEval_off()` of the `PRF` class (Fig.5.34)

The Outsourcing$_{lin}$ encoding

Following the description of the Outsourcing$_{lin}$ scheme, the next step is to implement the Encode algorithm. As already made for the Joye-Libert cryptosytem implementation, we have to implement the `Encode` virtual function of the `SchemeEncode` "interface" of the framework. For this reason, one can create a concrete class, i.e. called `FGP14_lin_funct_SchemeEncode`, in which define a body for the `Encode` virtual function.

Even in this case there is no much to say because the algorithm simply encrypts a value m (the code is the same described in the Joye-Libert encoding implementation) to produce a ciphertext for the `FGP_14_lin_funct_Encoding` and then it uses the `compute()` function of the `PRF` class to obtain the value to create the tag for the `FGP_14_lin_funct_Encoding` (Fig.5.35)

```
1  //generation of all the parameters
2  ...
3  //building the pseudorandom function parameters
4  PRF_parameters
       prf_params(RSA_sec_param,N,p_prime,q_prime,p,q,two_to_the_k,g,
       aes_key_byte_size,sha3_digest_byte_size);
5  //pseudorandom function seeds
6  pair<const byte*,const byte*> K1(k1,k1_prime);
7  pair<const byte*,const byte*> K2(k2,k2_prime);
8  //building the pseudorandom function keys (parameters + seeds)
9  PRF_seeds prf_seeds(K1,K2);
10 PRF_keys prf_keys(prf_params,prf_seeds);
11 //building all the keys (to encode, compute and decode)
12 FGP_14_lin_funct_EncodKey* encodKey = new
       FGP_14_lin_funct_EncodKey(k_message_bit_size,N,y,alpha,prf_keys);
13 FGP_14_lin_funct_ComputKey* computKey= new
       FGP_14_lin_funct_ComputKey(N);
14 FGP_14_lin_funct_DecodKey* decodKey= new
       FGP_14_lin_funct_DecodKey(p,p_prime,y,k_message_bit_size,N,alpha,prf_keys);
15 Keys* keys= new Keys(encodKey,computKey,decodKey);
16 return keys;
```

Figure 5.33: Final part of the `KeyGen` function code inside the `FGP14_lin_funct_SchemeKeyGen` class

.

The Outsourcing$_{lin}$ computation

The Compute algorithm of the Outsourcing$_{lin}$ scheme is not present in the Joye-Libert implementation. In this case we need to implement also the `Compute` virtual function of the `SchemeCompute` “interface” of the framework. Again, one can create a concrete class, i.e. called `FGP14_lin_funct_SchemeCompute`, in which define a body for the `Compute` virtual function.

The code is self-explanatory (Fig.5.36); it simply follows the description given in Sec. 3.3.3.2.

The Outsourcing$_{lin}$ decoding

Regarding the `Decode` algorithm of the Outsourcing$_{lin}$ scheme, we have to make a distinction between the case in presence of the ACF-efficiency property and that without such property.

```
1  OnlineDecodKey* FGP_14_lin_funct_User:: OfflineDecode(const DecodKey&
       decodKey, const Program& prog)
2  {
3          OnlineDecodKey* onlineDecodKey=NULL;
4          FGP_14_lin_funct_DecodKey FGP_decodKey(dynamic_cast<const
               FGP_14_lin_funct_DecodKey&>(decodKey));
5          PRF my_prf(FGP_decodKey.get_prf_keys());
6
7          ZZ w_f=my_prf.CFEval_off(prog);
8
9          FGP_14_lin_funct_OnDecodKey ondecodkey(w_f);
10
11         onlineDecodKey= new OnlineDecodKey(decodKey,ondecodkey);
12         return onlineDecodKey;
13 }
```

Figure 5.34: `OfflineDecode` function code inside the `FGP14_lin_funct_SchemeOfflineDecode` class

.

```
1  Encoding* FGP_14_lin_funct_User:: Encode(const EncodKey& encKey, const
       Label& delta, const Label& tau_i, const Message& message)
2  {
3          FGP_14_lin_funct_Encoding* encoding=NULL;
4          ZZ c_i;
5          ZZ prf_output=my_prf.compute(delta,tau_i);
6          ZZ c_alpha,t_i;
7          PowerMod(c_alpha,c_i,alpha,N);
8          t_i = MulMod(c_alpha,prf_output,N);
9          encoding = new FGP_14_lin_funct_Encoding(c_i,t_i);
10         return encoding;
11 }
```

Figure 5.35: `Encode` function code inside the `FGP14_lin_funct_SchemeEncode` class

.

As always, we need to implement the `Decode` virtual function of the `SchemeDecode` "interface" of the framework and the `OnlineDecode` virtual function of the `SchemeOnlineDecode` "interface" of the framework. Again, one can create two concrete classes, i.e. called `FGP14_lin_funct_SchemeDecode`, in which define a body for

```
1  Encoding* FGP_14_lin_funct_User:: Encode(const EncodKey& encKey, const
       Label& delta, const Label& tau_i, const Message& message){
2          FGP_14_lin_funct_Encoding* encoding=NULL;
3          FGP_14_lin_funct_Program_function function(dynamic_cast<const
               FGP_14_lin_funct_Program_function&>(funct));
4          ZZ N=(dynamic_cast<const
               FGP_14_lin_funct_ComputKey&>(compKey)).get_N();
5          int coefficients=function.get_coefficients_number();
6          if(coefficients!=(int)vec.get_vector_size()){
7                      cout << "\nError";
8                      exit(EXIT_FAILURE);
9          }
10         ZZ c(1),t(1),fi(0);
11         int n;
12         ZZ c_i,t_i;
13         FGP_14_lin_funct_Encoding* enc;
14         for(n=0;n<coefficients;n++) {
15                 fi=function.get_coefficient(n);
16                 enc=(dynamic_cast<FGP_14_lin_funct_Encoding*>(
                       vec.get_Encoding(n)) );
17                 c_i=enc->get_Ciphertext();
18                 t_i=enc->get_Tag();
19                 MulMod(c,c,PowerMod(c_i,fi,N),N);
20                 MulMod(t,t,PowerMod(t_i,fi,N),N);
21         }
22         encoding = new FGP_14_lin_funct_Encoding(c,t);
23         return encoding;
24 }
```

Figure 5.36: `Compute` function code inside the `FGP14_lin_funct_SchemeCompute` class

.

the `Decode` virtual function, and `FGP14_lin_funct_SchemeOnlineDecode`, in which define a body for the `SchemeOnlineDecode` virtual function.

The code for the `Decode` function implementation is showed in Fig.5.37. In the first stage, for each τ_i label inside the `Program` we call the `compute()` function of the `PRF` function. Than, all these values are used to compute the product W; once that the equation to set the acceptance bit is verified, if the acceptance bit is 1 we can decrypt the ciphertext contained in the encoding produced by the `Compute` function. The decryption algorithm code is the same seen for the Joye-Libert implementation.

The code for the `OnlineDecode` function implementation is showed in Fig.5.38. Here the code is even simpler; instead of compute a PRF for each τ_i inside the `Program`, it is simply call the `CFEval_on()` function of the `PRF` class using the `FGP_14_lin_funct_OnDecodKey` inside the `OnlineDecodKey` produced by the `OfflineDecode` function. Than, the value W outputted by the PRF is used to compute the equation that has to be verified to set the acceptance bit; if the acceptance bit is 1 we can decrypt the ciphertext contained in the encoding produced by the `Compute` function. The decryption algorithm code is the same seen for the Joye-Libert implementation.

Final remarks

We conclude this section with a last comment. The $\mathsf{Outsourcing}_{lin}$ scheme implementation is an implementation in which every algorithm of the $\mathsf{Outsourcing}_{lin}$ scheme has its correspondent class in the implementation ($\mathsf{KeyGen}() \leftrightarrow$ `FGP_14_lin_funct_SchemeKeyGen`, $\mathsf{Encode}() \leftrightarrow$ `FGP_14_lin_funct_SchemeEcode`, etc.) providing in this way a one-to-one correspondence. This is, clearly, not the only possible solution. Another approach could be that in which one creates a class `User` that implements more than one ABC of the framework, i.e. the `SchemeEncode` and the `SchemeDecode`, and a class `Server` that implements the `SchemeCompute` ABC. This observation suggests that there are many way to employ the framework, combining the interfaces in the way that better fits with the own scheme.

```
1  Decoding* FGP_14_lin_funct_User:: Decode(const DecodKey& decodKey,
       const Program& prog, const Label& delta, const Encoding& encod){
2          Decoding* decoding=NULL;
3          FGP_14_lin_funct_DecodKey FGP_decodKey(dynamic_cast<const
               FGP_14_lin_funct_DecodKey&>(decodKey));
4          FGP_14_lin_funct_Encoding FGP_encoding(dynamic_cast<const
               FGP_14_lin_funct_Encoding&>(encod));
5          FGP_14_lin_funct_Program_function function(dynamic_cast<const
               FGP_14_lin_funct_Program_function&>(prog.get_function()));
6          ZZ alpha,N,ct;
7          alpha = FGP_decodKey.get_alpha();
8          N = FGP_decodKey.get_N();
9          c=FGP_encoding.get_Ciphertext();
10         t=FGP_encoding.get_Tag();
11         PRF my_prf(FGP_decodKey.get_prf_keys());
12         ZZ W_i,W(1);
13         int coefficients=function.get_coefficients_number();
14         int n;
15         for(n=0;n<coefficients;n++) {
16             W_i=my_prf.compute(delta,prog.get_tau(n));
17             MulMod(W,W,PowerMod(W_i,function.get_coefficient(n),N),N);
18         }
19         ZZ c_to_the_alpha_times_W=MulMod(PowerMod(c,alpha,N), W, N);
20         if((operator==(t,c_to_the_alpha_times_W))==1) {
21                 //decryption algorithm
22                 ...
23                 ZZ m; //decrypted value
24                 FGP_14_lin_funct_Message* mess= new
                       FGP_14_lin_funct_Message(m);
25                 decoding= new Decoding(true,mess);
26         }
27         else {
28                 ZZ empty(0);
29                 FGP_14_lin_funct_Message* mess= new
                       FGP_14_lin_funct_Message(empty);
30                 decoding= new Decoding(false,mess);
31         }
32         return decoding;
33 }
```

Figure 5.37: Decode function code inside the FGP14_lin_funct_SchemeDecode class
.

```
1  Decoding* FGP_14_lin_funct_User:: OnlineDecode(const OnlineDecodKey&
       onlineDecodKey, const Label& delta, const Encoding& encod){
2          Decoding* decoding=NULL;
3          FGP_14_lin_funct_Encoding FGP_encoding(dynamic_cast<const
               FGP_14_lin_funct_Encoding&>(encod));
4          FGP_14_lin_funct_OnDecodKey FGP_on_decodKey(dynamic_cast<const
               FGP_14_lin_funct_OnDecodKey&>(onlineDecodKey.get_OnDecodKey()));
5          FGP_14_lin_funct_DecodKey FGP_decodKey(dynamic_cast<const
               FGP_14_lin_funct_DecodKey&>(onlineDecodKey.get_DecodKey()));
6          ZZ w_f= FGP_on_decodKey.get_w_f();
7          ZZ alpha = FGP_decodKey.get_alpha();
8          ZZ N = FGP_decodKey.get_N();
9          ZZ c=FGP_encoding.get_Ciphertext();
10         ZZ t=FGP_encoding.get_Tag();
11         PRF my_prf(FGP_decodKey.get_prf_keys());
12         ZZ W=my_prf.CFEval_on(delta,w_f);
13         ZZ c_to_the_alpha_times_W=MulMod(PowerMod(c,alpha,N), W, N);
14         if((operator==(t,c_to_the_alpha_times_W))==1){
15                 //decryption algorithm
16                 //...
17                 ZZ m; //decrypted value
18                 FGP_14_lin_funct_Message* mess= new
                       FGP_14_lin_funct_Message(m);
19                 decoding= new Decoding(true,mess);
20         }
21         else {
22                 ZZ empty(0);
23                 FGP_14_lin_funct_Message* mess= new
                       FGP_14_lin_funct_Message(empty);
24                 decoding= new Decoding(false,mess);
25         }
26         return decoding;
27 }
```

Figure 5.38: `OnlineDecode` function code inside the `FGP14_lin_funct_SchemeOnlineDecode` class

.

Chapter 6

Experiments

In this chapter we show some timing performances related to several experiments we performed using the implementations described in the previous chapter.

Following the content of the chapter, the Sec. 6.1 is dedicated to the experiments about the Joye-libert cryptosystem. We show the timing performances of all the algorithms, considering different configurations of parameters. In the Sec. 6.2 we present the experiments about the $\mathsf{Outsourcing}_{lin}$ scheme. Even in this case we show the timing performances of all the algorithms involved in the scheme.

Recalling the considerations made in the last part of Sec. 5.1.2 about the constraints related to the bit-size k of the message to encrypt, related to the security parameter and to the bit size of the composed number N, we decided to perform the following experiments:

- with security parameter $\lambda = 80$ and bit-size of the message $k = 128$;
- with security parameter $\lambda = 128$ and bit-size of the message $k = 128$;
- with security parameter $\lambda = 128$ and bit-size of the message $k = 128$;
- with security parameter $\lambda = 128$ and bit-size of the message $k = 128$

The experiments have been done on a Intel Core i7-3610QM (2.30 GHz, 6 MB L3 cache, 4 cores, 8GB RAM DDR3) machine, with Operating System Ubuntu 12.04 LTS 64 bit.

The resulting timings shown in the following sections represent the average values over multiple iterations (depending on the algorithm in exam) for each experiment. Actually, we will also show the standard deviations.

6.1 Experiments about the implementation of the Joye-Libert cryptosystem

This section is dedicated to the description of the experiments related to the implementation of the Joye-Libert linear homomorphic encryption scheme.

6.1.1 Key generation

From the table it is possible to see that most of the Key generation time is spent for the generation of the prime numbers.

We provide the results over sets of 100 iterations for each experiment.

Table 6.1 shows the experiments, describing the resulting timings in milliseconds.

	λ	k	*Average* (ms)	*Standard deviation* (ms)
Key Generation	80	128	9.317	5.079
Primes generation	80	128	9.035	5.089
Remaining part	80	128	0.281	0.272
Key Generation	128	128	284.090	195.570
Primes generation	128	128	284.059	195.525
Remaining part	128	128	0.031	0.313
Key Generation	128	256	258.916	173.334
Primes generation	128	256	258.853	173.377
Remaining part	128	256	0.063	0.442
Key Generation	128	512	272.941	165.233
Primes generation	128	512	271.941	165.197
Remaining part	128	512	0.999	1.106

Table 6.1: Key Generation algorithm and its sub-parts in milliseconds.

6.1.2 Encoding (encryption)

The second algorithm we analyze is the encoding algorithm.

We provide the results over sets of 100 iterations for each experiment. We also consider two cases:

(a) messages randomly picked in $\{0, ..., 2^k - 1\}$;

(b) random messages of 32 bits.

Table 6.2 shows the experiments for the case (a) while Table 6.3 shows the experiments for the case (b) . In both cases, the resulting timings are expressed in milliseconds.

λ	k	*Average*(ms)	*Standard deviation*(ms)
80	128	0.258	0.025
128	128	1.411	0.032
128	256	2.673	0.068
128	512	5.041	0.399

Table 6.2: Encoding (encryption) of messages randomly picked in $\{0, ..., 2^k - 1\}$. Results in milliseconds.

λ	k	*Average*(ms)	*Standard deviation*(ms)
80	128	0.150	0.030
128	128	0.615	0.383
128	256	1.447	0.024
128	512	2.604	0.052

Table 6.3: Encoding (encryption) of random messages of 32 bits. Results in milliseconds.

6.1.3 Computation

The third algorithm we analyze is the compute algorithm. For the experiments, we consider the composition of the two basic functions Add and CMult presented in the Eval algorithm of the Joye-libert encryption scheme. Namely, given a set $\mathbf{a}= (a_1, ..., a_n)$ of n random elements belonging to the message space, and considering n ciphertexts, $\mathbf{c}=(c_1, ..., c_n)$ the function f we chose outputs $c_1^{a_1} \cdot c_2^{a_2}, ..., c_n^{a_n}$.

We perform 10 iterations for each experiment, varying the number n of coefficients and ciphertexts, chosing n equal to 1000, 10000, 100000. The function coefficients and the ciphertexts are randomly picked, at each iteration, in $\{0, ..., 2^k - 1\}$.

Table 6.4 shows the experiments, describing the resulting timings in seconds.

n	λ	k	*Average*(s)	*Standard deviation*(s)
1000	80	128	0.105	0.003
10000	80	128	1.018	0.012
100000	80	128	10.403	0.241
1000	128	128	0.733	0.0062
10000	128	128	7.244	0.084
100000	128	128	107.798	0.767
1000	128	256	1.424	0.014
10000	128	256	19.670	2.706
100000	128	256	141.153	1.2
1000	128	512	2.766	0.021
10000	128	512	38.188	3.46
100000	128	512	0	0

Table 6.4: Compute algorithm in seconds.

6.1.4 Decoding (decryption)

The last algorithm we analyze is the decode (decryption) algorithm. Here we compare the resulting timings of the best decryption algorithm presented by Joye and Libert in [18] and the resulting timings of our new decryption algorithm. Even in this case, we provide the results over sets of 100 iterations for each experiment. We also consider the cases:

(a) messages randomly picked in $\{0, ..., 2^k - 1\}$;

(b) random messages of 32 bits.

Table 6.5 shows the experiments for the case (a) while Table 6.6 shows the experiments for the case (b) .

	λ	k	*Average* (ms)	*Standard deviation* (ms)
Joye-Libert (Alg. 3.5)	80	128	2.245	0.112
Our alg. (Alg. 3.7)	80	128	1.469	0.149
Joye-Libert (Alg. 3.5)	128	128	13.596	0.280
Our alg. (Alg. 3.7)	128	128	8.880	0.646
Joye-Libert (Alg. 3.5)	128	256	47.255	0.905
Our alg. (Alg. 3.7)	128	256	28.768	1.651
Joye-Libert (Alg. 3.5)	128	512	180.376	2.038
Our alg. (Alg. 3.7)	128	512	107.821	6.012

Table 6.5: Decoding (decryption) of messages randomly picked $\in \{0, \ldots, 2^k - 1\}$. Results in milliseconds.

	λ	k	*Average* (ms)	*Standard deviation* (ms)
Joye-Libert (Alg. 3.5)	80	128	1.623	0.870
Our alg. (Alg. 3.7)	80	128	0.649	0.366
Joye-Libert (Alg. 3.5)	128	128	13.192	0.279
Our alg. (Alg. 3.7)	128	128	5.359	0.522
Joye-Libert (Alg. 3.5)	128	256	43.087	0.637
Our alg. (Alg. 3.7)	128	256	8.83	1.076
Joye-Libert (Alg. 3.5)	128	512	161.095	2.006
Our alg. (Alg. 3.7)	128	512	15.637	2.389

Table 6.6: Decoding (decryption) of random messages of 32 bits. Results in milliseconds.

6.2 Experiments about the implementation of the $\mathsf{Outsourcing}_{lin}$ scheme

This section is dedicated to the description of the experiments related to the the implementation of the $\mathsf{Outsourcing}_{lin}$ scheme.

6.2.1 Key generation

The first algorithm we analyze is the key generation algorithm. In this case, we consider the resulting timings of the complete Key Generation algorithm and the it sub-parts:

- prime numbers generation;
- PRF key generation;
- remaining part.

Recall that the key generation algorithm of the $\mathsf{Outsourcing}_{lin}$ scheme has to produce two k-quasi-safe primes, so the time required is higher. we provide the results over sets of 10 iterations for each experiment. Table 6.7 shows the experiments in seconds.

	λ	k	*Average* (s)	*Standard deviation* (s)
Key Generation	80	128	1.723	0.555
Primes generation	80	128	1.722	0.555
PRF key generation	80	128	0.0005	0.00007
Remaining part	80	128	0.0004	0.0003
Key Generation	128	128	152.963	132.948
Primes generation	128	128	152.656	132.948
PRF key generation	128	128	0.005	0.0008
Remaining part	128	128	0.0013	0.0007
Key Generation	128	256	185.085	164.070
Primes generation	128	256	185.078	164.070
PRF key generation	128	256	0.0061	0.001
Remaining part	128	256	0.00126	0.00069
Key Generation	128	512	82.612	33.795
Primes generation	128	512	82.603	33.795
PRF key generation	128	512	0.0086	0.0018
Remaining part	128	512	0.00096	0.00065

Table 6.7: Key Generation algorithm in seconds.

We can see from the resulting timings that when the value of k increases, the average value can decrease. We can try to justify this fact remembering the algorithm of generation of the two k-quasi-safe primes p and q. At each iteration of such algorithm we have to generate a random prime p' whose dimension depends on k and we have to verify the primality of the entire prime p; recall that $p = 2^k p' + 1$, so if k increases, the bit-size of p' decreases. This leads to a reduction of the times, since we have to generate a prime p' smaller, while the bit-size of the probable prime p to check remains the same (it depends on the security parameter).

6.2.2 Encoding

The second algorithm we analyze is the encoding algorithm.

In this case, we consider the resulting timings of the complete Encode algorithm and the two sub-parts related to the generation of the ciphertext and of the tag.

Even in this case, we provide the results over sets of 100 iterations for each experiment. We also consider two cases:

(a) messages randomly picked in $\{0, ..., 2^k - 1\}$;

(b) random messages of 32 bits.

Table 6.8 shows the experiments for the case (a) while Table 6.9 shows the experiments for the case (b) . In both cases, the resulting timings are expressed in milliseconds.

6.2.3 Computation

The third algorithm we analyze is the compute algorithm. We perform 10 iterations for each experiment, varying the number n of messages over which we make the computation, choosing n equal to 1000, 10000, 100000.

We generate a random function of n random coefficients, in $\{0, ..., 2^k - 1\}$, at each iteration of every experiment. Therefore we consider n τ labels and only one Δ label at each iteration of every experiment.

Table 6.10 shows the experiments, describing the resulting timings in seconds.

In Table 6.11 and in Table 6.12 we provide the resulting timings of other experiments. We test two binary operations:

- $f(m_1, m_2) = m_1 + m_2$, with m_1, m_2 random messages of 32 bits;

	λ	k	*Average* (ms)	*Standard deviation* (ms)
Encoding	80	128	2.914	0.33
Cipher. generation	80	128	0.292	0.053
Tag generation	80	128	2.621	0.291
Encoding	128	128	67.178	0.655
Cipher. generation	128	128	1.919	1.936
Tag generation	128	128	65.247	0.612
Encoding	128	256	65.723	0.505
Cipher. generation	128	256	3.859	0.163
Tag generation	128	256	61.864	0.481
Encoding	128	512	60.42	0.144
Cipher. generation	128	512	7.20	0.03
Tag generation	128	512	52.831	0.128

Table 6.8: Encode algorithm and its sub-parts for messages randomly picked in $\{0, ..., 2^k - 1\}$. Results in milliseconds.

	λ	k	*Average* (ms)	*Standard deviation* (ms)
Encoding	80	128	3.186	0.274
Cipher. generation	80	128	0	0
Tag generation	80	128	3.186	0.274
Encoding	128	128	66.634	0.154
Cipher. generation	128	128	1.962	0.013
Tag generation	128	128	64.67	0.15
Encoding	128	256	46.992	0.55
Cipher. generation	128	256	2.69	0.06
Tag generation	128	256	44.293	0.536
Encoding	128	512	43.398	0.68
Cipher. generation	128	512	5.185	0.185
Tag generation	128	512	38.312	0.692

Table 6.9: Encode algorithm and its sub-parts for random messages of 32 bits. Results in milliseconds.

n	λ	k	*Average* (s)	*Standard deviation* (s)
1000	80	128	0.217	0.00665
10000	80	128	2.196	0.25
100000	80	128	29.627	0.514
1000	128	128	2	0.178
10000	128	128	20.674	0.0376
100000	128	128	173.812	19.747
1000	128	256	4.610	0.076
10000	128	256	41.489	0.061
100000	128	256	328.494	24.824
1000	128	512	7.535	0.64
10000	128	512	80.899	0.084
100000	128	512	707.738	153.271

Table 6.10: Compute algorithm in seconds.

- $g(m_1, m_2) = c_1 m_1 + c_2 m_2$, with m_1, m_2 random messages of 32 bits and c_1, c_2 coefficients of 32 bits.

We perform 100 iterations for each experiment.

λ	k	*Average* (ms)	*Standard deviation* (ms)
128	80	0.0017	0.0045
128	128	0.019	0.0042
128	256	0.022	0.0028
128	512	0.018	0.0041

Table 6.11: Compute algorithm for $f(m_1, m_2) = m_1 + m_2$. Results in milliseconds.

6.2.4 Decoding

The last algorithm we analyze is the Decode algorithm.

We decided to test the efficient version of the Decode algorithm, verifying the performances of the OfflineDecode algorithm and of the OnlineDecode algorithm.

λ	k	*Average* (ms)	*Standard deviation* (ms)
128	80	0.143	0.0077
128	128	0.824	0.02
128	256	0.801	0.116
128	512	0.689	0.301

Table 6.12: Compute algorithm for $g(m_1, m_2) = c_1 m_1 + c_2 m_2$. Results in milliseconds.

Offline decoding

The performances of the OfflineDecode algorithm depend on the number of messages to encode. As we have done for the Computation algorithm, we generate a random function of n random coefficients, in $\{0, ..., 2^k - 1\}$, at each iteration of every experiment. Therefore we consider n τ labels at each iteration of every experiment; we consider again results over a set of 10 iterations for each experiment. We vary the number n of *tau* labels, choosing n equal to 1000, 10000, 100000.

Table 6.13 shows the experiments, describing the resulting timings in seconds.

n	λ	k	*Average*(s)	*Standard deviation*(s)
1000	80	128	0.00633	0.000775
10000	80	128	0.0568	0.001
100000	80	128	0.796	0.0669
1000	128	128	0.0102	0.00217
10000	128	128	0.111	0.00238
100000	128	128	0.933	0.18
1000	128	256	0.0119	0.000123
10000	128	256	0.106	0.001
100000	128	256	0.911	0.175
1000	128	512	0.00947	0.000862
10000	128	512	0.109	0.0025
100000	128	512	0.913	0.211

Table 6.13: OfflineDecode algorithm in seconds.

In Table 6.14 and in Table 6.15 we provide the resulting timings of other experiments. We test the OfflineDecode algorithm of the two binary operations used to test the Compute algorithm:

- $f(m_1, m_2) = m_1 + m_2$, with m_1, m_2 random messages of 32 bits;
- $g(m_1, m_2) = c_1 m_1 + c_2 m_2$, with m_1, m_2 random messages of 32 bits and c_1, c_2 coefficients of 32 bits.

We perform 100 iterations for each experiment.

λ	k	*Average* (ms)	*Standard deviation* (ms)
80	128	0.024	0.017
128	128	0.03	0.007
128	256	0.035	0.005
128	512	0.031	0.007

Table 6.14: OfflineDecode algorithm for $f(m_1, m_2) = m_1 + m_2$. Results in milliseconds.

λ	k	*Average* (ms)	*Standard deviation* (ms)
80	128	0.024	0.006
128	128	0.033	0.006
128	256	0.031	0.007
128	512	0.026	0.012

Table 6.15: OfflineDecode algorithm for $g(m_1, m_2) = c_1 m_1 + c_2 m_2$. Results in milliseconds.

Online decoding

Regarding the OnlineDecode algorithm, we consider the resulting timings of the entire OnlineDecode algorithm and those of the verification step and of the decryption algorithm

Even in this case, we provide the results over sets of 100 iterations for each experiment. We consider two cases:

(a) online decoding of encodings of messages randomly picked in $\{0, ..., 2^k - 1\}$;

(b) online decoding of encodings of random messages of 32 bits.

Table 6.16 shows the experiments for the case (a) while Table 6.17 shows the experiments for the case (b) . In both cases, the resulting timings are expressed in milliseconds.

	λ	k	*Average* (ms)	*Standard deviation* (ms)
OnlineDecode	80	128	5.977	1.18
Verification	80	128	2.324	1.064
Decryption	80	128	3.653	0.501
OnlineDecode	128	128	68.686	1.609
Verification	128	128	46.256	1.269
Decryption	128	128	22.429	0.727
OnlineDecode	128	256	191.758	3.150
Verification	128	256	70.151	0.463
Decryption	128	256	121.605	3.111
OnlineDecode	128	128	475.714	7.772
Verification	128	256	55.217	0.281
Decryption	128	512	420.495	7.742

Table 6.16: OnlineDecode algorithm and its sub-parts for encodings of messages randomly picked in $\{0, ..., 2^k - 1\}$. Results in milliseconds.

	λ	k	*Average*	*Standard deviation*
OnlineDecode	80	128	4.679	0.409
Verification	80	128	1.911	0.213
Decryption	80	128	2.767	0.23
OnlineDecode	128	128	91.605	0.754
Verification	128	128	64.737	0.091
Decryption	128	128	26.866	0.76
OnlineDecode	128	256	162.190	4.653
Verification	128	256	71.573	2.699
Decryption	128	256	90.614	2.794
OnlineDecode	128	128	231.089	3.37
Verification	128	256	38.464	0.355
Decryption	128	512	192.624	3.297

Table 6.17: OnlineDecode algorithm and its sub-parts for encodings of random messages of 32 bits. Results in milliseconds.

Chapter 7

Conclusions

In this thesis we have tackled the problem of *secure* outsourcing of data and computation. We presented the security issues related to this problem, namely *integrity* and *privacy* and we analysed some possible solutions to these two issues, exploiting advanced cryptographic tools, such as Homomorphic Message Authenticators and Fully Homomorphic Encryption.

Our contribution is both theoretical and practical.

Considering our theoretical contribution, using as starting points the articles of [3] and [12], we have defined the new Outsourcing scheme with the aim of realizing a very *generic* and *flexible* model that might be employed to represent several *secure* outsourcing schemes. Such model can be used to represent secure outsourcing schemes that provide only integrity, only privacy or, interestingly, integrity with privacy. Using our new model we also re-defined an highly efficient scheme constructed in [12], that we called $\mathsf{Outsourcing}_{lin}$ and that is a scheme for computing multi-variate polynomials of degree 1 over the ring $\mathbb{Z}_{2^k}$.

Considering our practical contribution, we built a Framework to implement the Outsourcing scheme achieving, in this way, one of the main goals of this thesis. Then, such Framework has been tested to realize several implementations, specifically the implementation of the Joye-Libert cryptosystem and the implementation of our $\mathsf{Outsourcing}_{lin}$ scheme.

In the context of this practical work, the thesis also led to some novel contributions:

- the design and the implementation, in collaboration with Dario Fiore, of a new decryption algorithm for the Joye-Libert encryption scheme, that performs better than the algorithms proposed by the authors in [18];

- the implementation of the amortized-closed-form efficient pseudorandom function of [12]. There was no prior implementation of this function and it represented a non trivial work, which can become useful in other contexts.

The implementations have been used to execute several tests for measuring the timing performances of the main algorithms. Since there were no existing implementations of the $\mathsf{Outsourcing}_{lin}$ scheme, this allowed to analyse its behaviour in applications of practical interest.

The Framework has been created with the idea of helping in the implementation of theoretical secure outsourcing schemes. We hope that the framework might become a *concrete* point of reference for all those researchers, students or simply users that wish to implement and test own secure outsourcing schemes. While the work of the thesis accomplished the goals we had set (namely to build and use such framework), future work could be done.

Possible directions include: improving and optimizing our implementations; using the framework to implement more outsourcing schemes and to compare several of them; using the framework to realize secure outsourcing applications.

Appendix

Block cipher

[1] A block cipher is a deterministic algorithm operating on fixed groups of bits, called *blocks*, with an unvarying transformation that is specified by a symmetric key. The block size depends on the cipher being used, but it is usually 64 or 128 bits. A symmetric cipher uses linear and non-linear transformation to encrypt and decrypt messages; linear transformations are operations such as rotate and bit shifts, non-linear transformations include XOR, substitutions using a S-box, and permutations using a P-box. Linear and non-linear transformations are sometimes referred to as confusion and diffusion.

Among all the several existing block ciphers, probably one of the most used in practice is the AES. It specifies a cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. It has been adopted as a U.S. government Federal Information Processing Standard: FIPS PUB 197 Advanced Encryption Standard (AES). It has supersedes the *Data Encryption Standard* (DES) block cipher.

Modes of operation

There are several ways to use PRFs to encrypt arbitrarily long messages from *fixed length* PRFs. Historically, these methods are called *modes of operation*.

If we would like to encrypt data which is 64 bytes long, and we have chosen a cipher with a block size of 128 bits, the cipher will break the 64 bytes into four blocks, 128 bits each (Fig. 8.1). How the blocks are encrypted depends on the *mode of operation*.

[1] https://en.wikipedia.org/wiki/Block_cipher

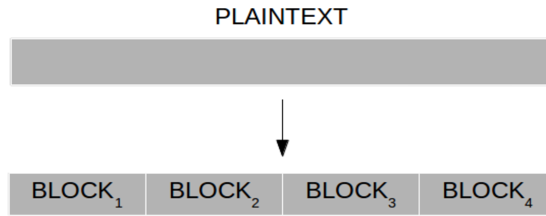

Figure 7.1: Blocking Plain Text.

In early cryptography (circa 1980), there were four approved modes from which to chose: ECB, CBC, OFB, and CFB. These modes were standardized (among others) in FIPS 81, ANSI X3.106, and ISO/IEC 10116. Later came modes such as CTR and CTS. CTR and CTS were standardized in NIST SP800-38A (SP800-38A recognized the four modes of FIPS 81, while ISO 10116 was updated). Modes of operation specify how the output of the previous rounds are used as input to the subsequent rounds. Depending on the desired properties of the cipher, we would select different modes.

Bibliography

[1] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, "Handbook of Applied Cryptography", CRC Press, 16 ott 1996 pp. 145-148

[2] Algorithms, key size and parameters report - 2014 November, 2014, European Union Agency for Network and Information Security

[3] M. Backes, D. Fiore, R. M. Reischuk, "Verifiable Delegation of Computation on Outsourced Data", ACM Connference on Computer and Communication Security, November 2013, ACM press

[4] S. Banabbas, R. Gennaro, Y. Vahlis, "Verifiable delegation of computation over large datasets", Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010, Proceedings, volume 6841 of Lecture Notes in Computer Science, Springer, 2011, pp. 111-131

[5] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", Retrieved April 13, 2013.

[6] D. Boneh, X. Boyen, H. Shacham, "Short group signatures", M. Franklin, editor, CRYPTO 2004, Vol. 3152 of LNCS, pp. 41-55, Santa Barbara, CA, USA, August 15-19, 2004, Springer, Berlin, German

[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully Homomorphic Encryption without Bootstrapping", ITCS 2012

[8] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP", CRYPTO 2012, Springer

[9] D. Catalano, D. Fiore, "Practical homomorphic MACs for arithmetic circuits, In Eurocrypt '13: Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2013.

[10] Y. Dodis, "Introduction to cryptography", Spring 2012, Lecture 8, page 5, http://www.cs.nyu.edu/courses/spring12/CSCI-GA.3210-001/index.html

[11] Y. Dodis, "Introduction to cryptography", Spring 2012, Lecture 11, pp 4-5, http://www.cs.nyu.edu/courses/spring12/CSCI-GA.3210-001/index.html

[12] D. Fiore, R. Gennaro, V. Pastro, “Efficiently verifiable computation on encrypted data”, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Pages 844-855, ACM New York, NY, USA ®2014 ISBN: 978-1-4503-2957-6, DOI 10.1145/2660267.2660366

[13] D.M. Freeman, “Improved security for linearly homomorphic signatures: A generic framework”, M.Fischlin, J. Buchmann, M. Manulis, editors, PKC 2012, Vol. 7293 of LNCS, pp. 697-714, Darmstadt, Germany, May 21-23, 2012, Springer, Berlin, Germany

[14] R. Gennaro, C. Gentry, B. Parno, “Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers”, 31 August 2010, CRYPTO 2010

[15] R. Gennaro, D. Wichs, “Fully homomorphic message authenticators”, Cryptology ePrint Archive, Report 2012/290, 2012, http://eprint.iacr.org/

[16] C. Gentry, “Fully homomorphic encryption using ideal lattices”, M. Mitzenmacher, editor, STOC, pp. 169-178, ACM, 2009

[17] Jonathan Katz and Yehuda Lindell, “Introduction to Modern Cryptography”, Chapman & Hall/CRC (Taylor & Francis Group), 2008 pp. 386-394, ISBN: 978-1-58488-551-1

[18] M. Joye, B. Libert, “Efficient cryptosystems from 2^k-th power residue symbols”, Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings, pp 76-92, ®2013, ISBN 978-3-642-38348-9, DOI 10.1007/978-3-642-38348-9_5

[19] Recommendation for key management - Part 1: General (Revision 3), NIST Special Publication 800-57, National Institute of Standards and Technology, 2012.