Non-linear Operations and Verifiable Decryption in BGV

Julen Bernabé Rodríguez

Information and Communication Technologies

Master's Degree on Cybersecurity and Privacy

Name of the tutor:
Katarzyna Kowalska

Name of the SRP:
Andreu Pere Isern Deyà

July 11, 2023

Universitat Oberta de Catalunya

Final Work Card

Title of the work:	Non-linear Operations and Verifiable Decryption in BGV
Name of the author:	Julen Bernabé Rodríguez
Name of the tutor:	Katarzyna Kowalska
Name of the SRP:	Andreu Pere Isern Deyà
Date of delivery:	July 11, 2023
Studies or Prgram: Privacy	Master's Degree on Cybersecurity and
Area or the Final Work: Technologies	Information and Communication
Language of the work:	English
Keywords:	Fully Hommomorphic Encryption, BGV, non-linear operations, verifiable decryption

Abstract

Fully Homomorphic Encryption (FHE) schemes are cryptosystems that allow to compute over encrypted messages. Due to the inherent difficulty of training models using private data, this special property is closely watched. One of the most celebrated FHE schemes is BGV, which allows to compute arithmetic operations over encrypted integers. Non-linear operations, though, need more work to be computed. In this work, several techniques to perform non-linear operations such as comparisons on BGV are introduced. An implementation of these techniques in the OpenFHE open-source library is given too.

Finally, an introduction to the verifiable decryption problem is introduced. Due to the very nature of FHE, the scientist only gets the encrypted results, and therefore has to ask the data owner to decrypt it. Verifiable decryption protocols are born precisely to ensure that this data decryption is done correctly. In this work, the verifiable decryption protocols for the BGV scheme in [4] and [50] are analyzed and the latter is implemented for OpenFHE.

Contents

Chapter 1

Introduction.

Since the beginning of times, people have wanted to hide information safely within messages. However, with the current importance of data analysis in any field, one no longer just wants to be able to send or store encrypted information; but wants to be able to operate on the data while it is encrypted. With this objective in mind, some Privacy Enhancement Technologies (PET) have been born that, through mathematical tools, provide this functionality. One of such technologies is Fully Homomorphic Encryption (FHE).

Basically, FHE schemes allow to operate over encrypted messages while preserving the result of the operations. This opens the possibility of exploiting previously unable data, due to its confidentiality. The private data exploitation process using homomorphic ciphers can be seen in Figure 1.1.

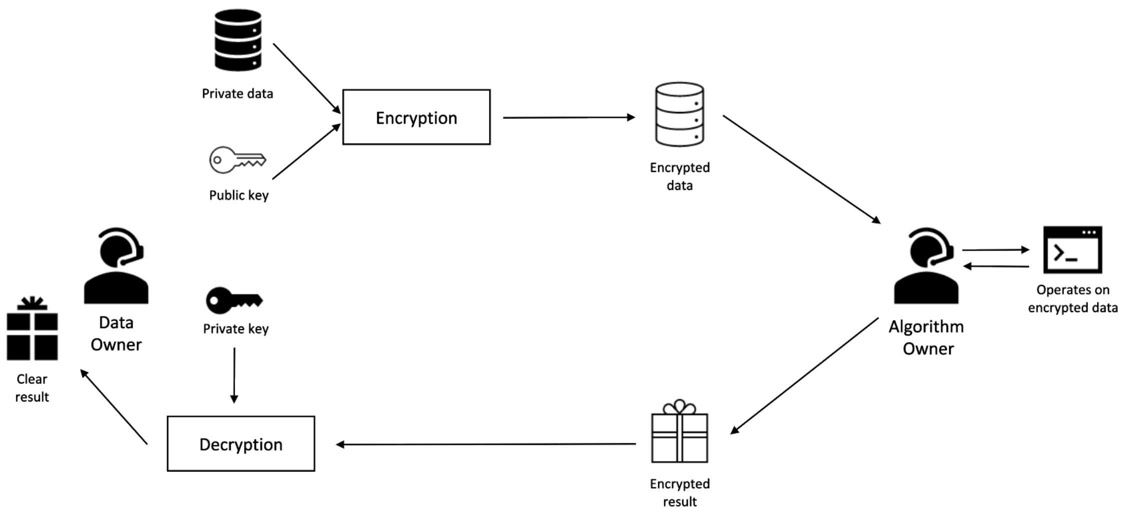


Figure 1.1: The process for computing over encrypted data using FHE.

The key property of these encryption systems is that the encryption function is an homomorphism. Being an homomorphism is a property of mappings between algebraic structures. Mappings with this property preserve some operations between the algebraic structures that define the function. Namely, a mapping:

$$f : A \longrightarrow B$$

is an homomorphism for operation $*$ if for every $a_1, a_2 \in A$:

$$f(a_1 * a_2) = f(a_1) \star f(a_2),$$

for some operation $\star$ defined in B.

A fully homomorphic encryption scheme, though, not only preserves one operation (addition or product) but both of them. This means that every combination of these operations can be computed homomorphically, opening a new paradigm in data exploitation.

There are several approaches to build FHE schemes. In this document, we focus on the BGV scheme, a simple but very efficient FHE scheme that allows to compute linear operations over encrypted integers. The main goal of this work is to extend this scheme to be able to compute non-linear operations, such as comparisons or integer divisions, homomorphically.

Despite letting us compute over encrypted data, there is an issue with data exploitations using FHE schemes. Indeed, the protocol from Figure 1.1 shows a straightforward way for computing over encrypted data using homomorphic encryption, denying the algorithm owner to learn nothing about the result. This protocol is useful when the result of the computation is only wanted by the data owner. However, in some cases, the algorithm owner also wants to know the result. For instance, in a Federated Learning setting, the owner of the model must know the results of the predictions of the model over the private datasets.

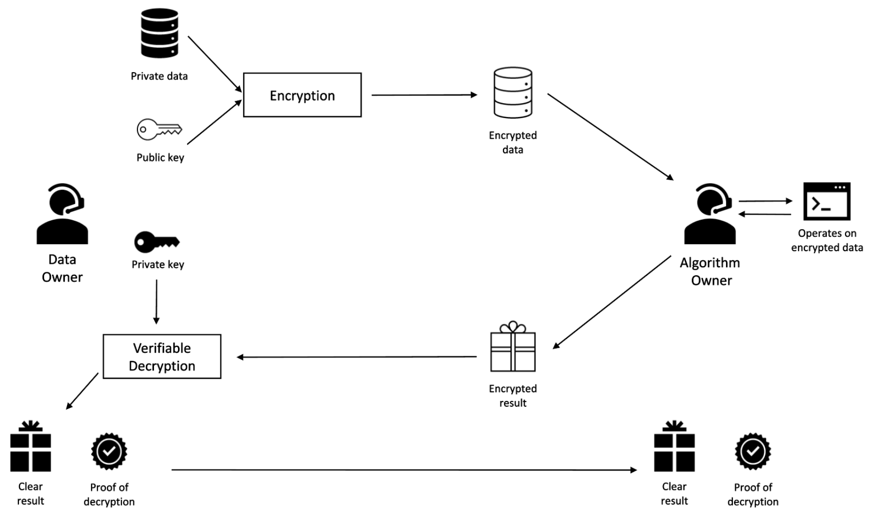


Figure 1.2: Process for computing over encrypted data using verifiable decryption.

In situations where the result must be given back to the algorithm owner, verifiable decryption (VD) techniques are needed. This way, the data exploitation process from Figure 1.1 is extended to the one shown in Figure 1.2, where the VD is considered as a functionality.

Verifiable decryption protocols for FHE are in an early life stage, but are considered a very useful tool to make possible to perform model trainings over private data. In this document, the VD problem in BGV is introduced, and two approaches to find a solution based on different technologies are given: lattice-based zero-knowledge proofs and threshold FHE.

In summary, this document is organized as follows. In Chapter 2 all the preliminaries and technical background are introduced. Chapter 3 describes the history of homomorphic encryption and gives a detailed explanation of the predominant FHE schemes presently available. Chapter 4 covers the techniques to perform non-linear operations over BGV ciphertexts, while Chapter 5 focuses on describing the verifiable decryption protocols for the BGV scheme. Finally, Chapters 6 and 7 give implementation details and conclusions, respectively.

1.1 Proposed methodology to achieve the previous objectives.

This section describes the methodology that is proposed to achieve the goals of the project. The project, in fact, will be split into five work packages (WP). WP1 is the most theoretical and introductory one, covering the general design and architecure of the project. Work packages 2 and 3 will focus on the implementations of the theoretical results obtained in WP1. WP0 and WP4 are transversal to the other three. WP4 covers all the writting that has to be done for the final memory and in WP0 all the coordinations between student and teacher occur. Figure 1.3 shows a Pert diagram for the project:

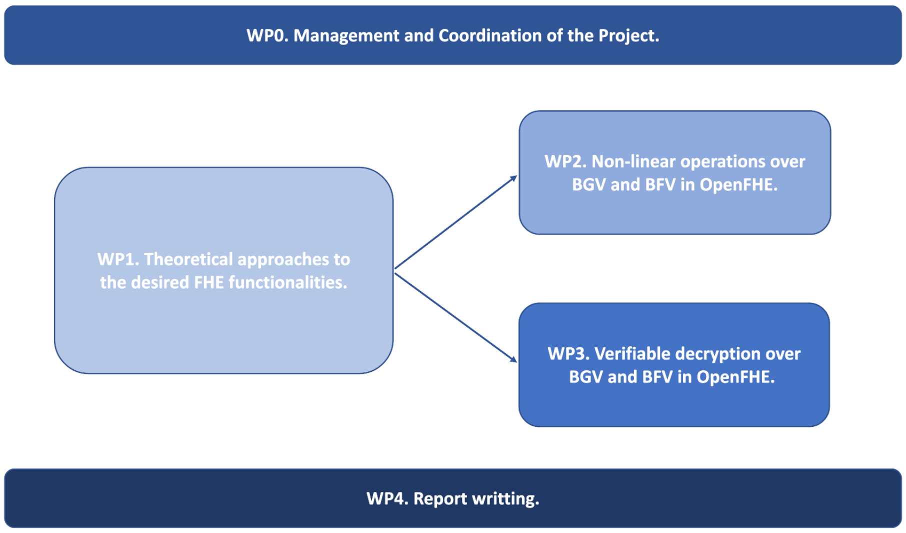

WP0. Management and Coordination of the Project.
WP1. Theoretical approaches to the desired FHE functionalities.
WP2. Non-linear operations over BGV and BFV in OpenFHE.
WP3. Verifiable decryption over BGV and BFV in OpenFHE.
WP4. Report writting.

Figure 1.3: Pert diagram for the Final Master's Thesis.

In the following, the work packages are described in detail, as well as the tasks that comprise them:

WP0. Management and Coordination of the Project.

Description. This work package encompasses all the others, and for this reason it will remain active throughout the entire project. Its main objective is that all the participants of the project can be coordinated to carry it out, and therefore manage it correctly.

Task 0.1. Management and Coordination of the Project.

This task covers all those activities that have to do with the project management and coordination: communication, meetings, minutes, coordination and follow-up of the tasks... For its correct execution, we will use different tools (Google Meet, Overleaf...) that facilitate teamwork and the exchange of documents for reading or editing.

Deliverables associated to this WP:
No deliverables are expected for this work package.

WP1. Theoretical Approaches to the desired FHE functionalities.

Description. This work package is intended to make all the advances that have to do with the theoretical part of the project. On the one hand, it will be necessary to analyze the existing State of the Art, in order to have a solid theoretical base that can be used in the following work packages. On the other hand, in this WP we will also work on the design of the algorithms that allow us to approximate the non-linear operations that we want to operate on BGV/BFV. Finally, the lattice-based ZKPs and Threshold FHE will be studied in detail, in order to apply this knowledge later in the project.

Task 1.1. Exhaustive analysis over the State of the Art.

In task 1.1, the main objective is to analyze the state of the art of FHE, understand in detail the BGV and BFV ciphers and all the necessary theoretical background so that it can be applied in the following tasks. Within this task, the following activities can be listed:

- Study of the theoretical basis necessary to understand the BGV/BFV ciphers. Within this theoretical base, it will be necessary to collect results of:
  - Commutative algebra
  - Group theory
  - The theory behind polynomial rings
  - The RLWE problem
  - etc.
- Analysis of homomorphic ciphers prior to BFV and BGV.
- Comprehensive study of BGV/BFV ciphers, their properties and homomorphic operations.

Task 1.2. Algorithm design for non-linear operations over BGV/BFV.

Task 1.2 has as main objective the design of the algorithms that will allow applying non-linear operations on BGV/BFV ciphertexts. To achieve this objective, this task is broken down into the following activities:

- Selection of the operations that we want to calculate.
- Study of the applicable techniques to be able to approximate the above operations. Among others, the following will be studied:
  - Fermat's Little Theorem.
  - Lagrange's Polynomial Interpolation Theorem.
- Design of the algorithms that will allow, through the previous techniques, to calculate the desired non-linear operations.

Task 1.3. Deep dive in lattice-based ZKPs and Threshold FHE.

Finally, in this last task, we will carry out an exhaustive study of the two approaches mentioned in this document to build VD protocols: lattice-based ZKPs and Threshold FHE. We will have the following activities:

- Preliminary study on the ZKPs, the most used ones and the problems they solve.
- Exhaustive analysis on lattice-based ZKPs, the theory behind them and the problems they solve.
- Analysis of the State of the Art in Threshold FHE, the available designs and existing implementations.

Deliverables associated to this WP:

- D1.1: Deliverable on the theoretical approaches to the desired FHE functionalities.

WP2. Non-linear operations over BGV and BFV in OpenFHE.

Description. In this work package, all the necessary advances will be made to be able to apply non-linear operations on ciphertexts in BGV/BFV. The main idea of this work package is to study the OpenFHE library in detail, see the features it has, and from these features and the work done in WP1, implement the algorithms designed to approximate non-linear operations. Finally, a study of the performance of the proposed solutions will be carried out, taking into account the BGV/BFV parameters, which define the security of the scheme.

Task 2.1. Study of the available functionalities in BGV/BFV inside OpenFHE.

In this task, we will take a closer look at the code available in the OpenFHE implementation of BGV and BFV. From this code, it will be easier to analyze what new operations need to be implemented to achieve the desired functionality (*i.e.,* the non-linear operations). The activities to be carried out in this task are:

- Analysis of the code available in OpenFHE for BGV/BFV and the list of operations that can be performed.
- Run code examples from the library.
- Preparation of the environment for the implementation of the algorithms.

Task 2.2. Implementation of the non-linear operations over BGV/BFV.

Task 2.2 focuses on the implementation of algorithms that approximate non-linear operations. During this task, the code analyzed in the above task will be extended with the necessary functionalities to be able to calculate the desired operations. These are the activities to be carried out:

- Creation of libraries that provide basic functionalities necessary during the algorithms. For instance, Fermat's Little Theorem or Polynomial Interpolation.
- Implementation of the approximation algorithms using the previous libraries.

Task 2.3. Performance and security issues related to the solution.

As the last task, the performance of the given solution for this problem will be analyzed. The main idea is to vary the cryptographic context defined for the BGV and BFV ciphers (which is the one that defines the security of the scheme), and observe how this affects the efficiency of the implemented algorithms. With these results, we will have a better idea of what it costs to solve this problem in real use cases.

Deliverables associated to this WP:

- D2.1: Implementation of the approximations for non-linear operations over BGV/BFV in OpenFHE.
- D2.2: Performance and security results of the given solution.

WP3. Verifiable decryption over BGV and BFV in OpenFHE.

Description. WP3 focuses on giving a solution for a verifiable decryption protocol over OpenFHE. Taking as a basis the work done in Task 1.3, this work package intends to give a theoretical design of a VD protocol, implement it and finally study its performance depending on the security parameters used inside BGV/BFV.

Task 3.1. Design of a verifiable decryption protocol for BGV and BFV.

The first task to consider is precisely the design of the verifiable decryption protocol for BGV/BFV. For this purpose, both Threshold FHE and lattice-based ZKP approaches will be considered, and finally only one solution will be given. The activities to be done in this task are:

- Propose a verifiable decryption protocol using Threshold FHE.
- Design a verifiable decryption protocol using lattice-based zero-knowledge proofs and commitments.
- Choose the design that is more suitable considering: simplicity, security, efficiency...

Task 3.2. Study of the implementation of the protocol in OpenFHE.

Task 3.2 focuses on the study of the implementation of the VD protocol designed in the previous task. During this task, the code analyzed in Task 2.1 will be extended with the necessary functionalities to be able to perform a verifiable decryption on the BGV/BFV ciphertexts. These are the activities to be carried out:

- Study of the creation of libraries that provide basic functionalities necessary during the algorithms. For instance, lattice-based zero-knowledge proofs or lattice-based commitments.
- Study of the implementation of the verifiable decryption protocol for BGV and BFV.

Task 3.3. Performance and security issues related to the implementation.

As the last task, the performance of the given solution for this problem will be analyzed. The main idea is to vary the cryptographic context defined for the BGV and BFV ciphers (which is the one that defines the security of the scheme), and observe how this affects the efficiency of the protocol. With these results, we will have a better idea of what it costs to have a VD protocol for BGV/BFV in real use cases.

Deliverables associated to this WP:

- D3.1: Design of the verifiable decryption protocol for BGV and BFV and the study of its implementation inside OpenFHE.
- D3.2: Performance and security results of the given solution.

WP4. Report writting.

Description. Lastly, WP4 is focused on writting the memory of the Final Master's Thesis. This memory covers all the previously proposed deliverables and is the central part of the Final Master's Thesis. This work package will be active throughout the entire project.

Task 4.1. Report writting.

This WP only has one task, which covers the writting of the report. The activities forming this task are:

- First version of the memory (after ending up WP1).
- Second version of the memory (after ending WP2).
- Third version of the memory (after ending WP3 and revising the whole document).

Deliverables associated to this task:

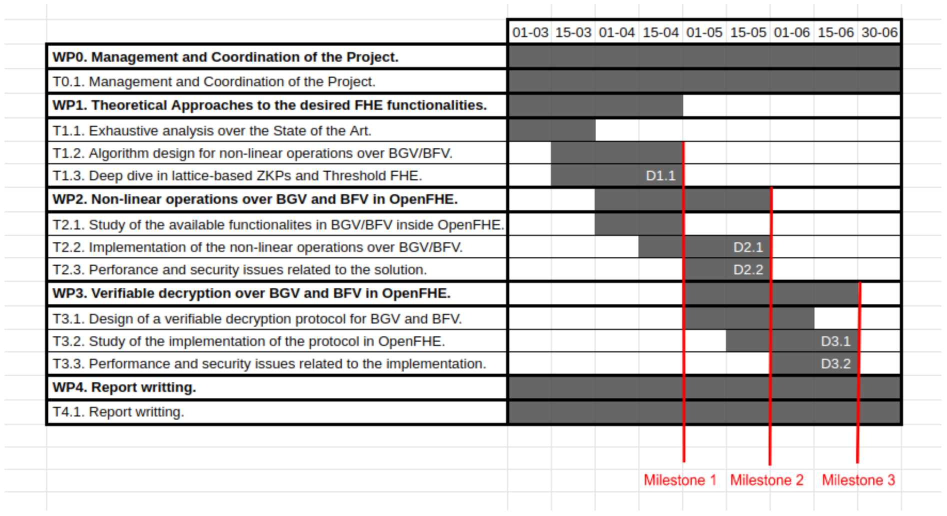
- D3.1: Memory of the Final Master's Thesis.

Project milestones and deadlines.

As can be deduced from the work packages, the project has the following milestones:

1. Theoretical approaches to the desired FHE functionalities.
2. Non-linear operations over BGV and BFV in OpenFHE.
3. Verifiable decryption over BGV and BFV in OpenFHE.

The deadlines for the above milestones and the deliverables associated to them can be seen in the Gantt diagram below:



	01-03	15-03	01-04	15-04	01-05	15-05	01-06	15-06	30-06
WP0. Management and Coordination of the Project.	▨	▨	▨	▨	▨	▨	▨	▨	▨
T0.1. Management and Coordination of the Project.	▨	▨	▨	▨	▨	▨	▨	▨	▨
WP1. Theoretical Approaches to the desired FHE functionalities.	▨	▨	▨	▨					
T1.1. Exhaustive analysis over the State of the Art.	▨	▨							
T1.2. Algorithm design for non-linear operations over BGV/BFV.		▨	▨	▨					
T1.3. Deep dive in lattice-based ZKPs and Threshold FHE.		▨	▨	D1.1					
WP2. Non-linear operations over BGV and BFV in OpenFHE.			▨	▨	▨	▨			
T2.1. Study of the available functionalites in BGV/BFV inside OpenFHE.			▨	▨					
T2.2. Implementation of the non-linear operations over BGV/BFV.				▨	▨	D2.1			
T2.3. Perforance and security issues related to the solution.					▨	D2.2			
WP3. Verifiable decryption over BGV and BFV in OpenFHE.					▨	▨	▨	▨	
T3.1. Design of a verifiable decryption protocol for BGV and BFV.					▨	▨	▨		
T3.2. Study of the implementation of the protocol in OpenFHE.						▨	▨	D3.1	
T3.3. Performance and security issues related to the implementation.							▨	D3.2	
WP4. Report writting.	▨	▨	▨	▨	▨	▨	▨	▨	▨
T4.1. Report writting.	▨	▨	▨	▨	▨	▨	▨	▨	▨

Figure 1.4: Gantt diagram for the project.

1.2 EU Sustainable Development Goals.

The ultimate objective of this work has a strong relation with goal 16 of the Sustainable Development Goals proposed by the United Nations (UN): *Peace, Justice and Strong Institutions.* Since the beginning of Big Data and data analysis, the big companies have benefited from the personal data of citizens. For this reason, as an example, the General Data Protection Regulation (GDPR) was published by the European Union in 2016. The main goal of this regulation was to protect the users' sensitive data from being used for the benefit of third parties. For that reason, the GDPR does not allow to exploit the private data of a EU citizen without the explicit consent of that citizen.

This regulation, however, does not consider the existing privacy enhancing technologies (PETs), such as homomorphic encryption. PETs are technologies that allow to compute over private data while protecting the privacy of that data. This way, in some sense, we return the ownership of the data to the data owner, and the big companies can continue benefiting from that data without the need of knowing it exactly.

In countries out of the EU, the situation is even worse. Companies can do whatever they want with users' data. There is no law protecting citizens' privacy, and besides, there is a lot of interest in exploting that data. The use of PETs in these countries means protecting the users and their right to have privacy, which is one of the Fundamental Human Rights.

What is more, these technologies open a new data economy paradigm: a marketplace where every user can rent their data to be used in an FHE algorithm, but with the data really being protected.

In conclusion, the advances in this Final Master's Thesis are proposed with the final vision of making the institutions and companies use the EU citizens' data in a cleaner way. Since FHE protects the data privacy, extending the functionality of this technology brings us closer to a more secure data exploitation.

Chapter 2

Preliminaries.

2.1 Basic algebra.

2.1.1 Group theory.

Definition 1 *A **group** is a set G together with an operation $\cdot$ that satisfies the properties:*

- *Associativity: For every $g_1, g_2, g_3 \in G$:*

$$g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3.$$

- *Identity: There exists an element e, called the identity, such that*

$$g \cdot e = e \cdot g = g \ \forall g \in G$$

- *Inverses: For every $g \in G$, there exists an element $g^{-1} \in G$ such that*

$$g \cdot g^{-1} = g^{-1} \cdot g = e$$

Besides, if $\cdot$ is also commutative, i.e., if:

$$g \cdot g' = g' \cdot g \ \forall g, g' \in G$$

*then the group is called **abelian**.*

Example The following classification can be done for the main number sets:

- $(\mathbb{Z}, +), (\mathbb{Q}, +)$ and $(\mathbb{R}, +)$ are abelian groups. Here $+$ denotes the usual addition.
- $(\mathbb{Q} - \{0\}, \cdot)$ and $(\mathbb{R} - \{0\}, \cdot)$ are abelian groups, but $\mathbb{Z}$ is not a group with the usual product operation.
- $(\mathbb{Z}_n, +)$, where $\mathbb{Z}_n$ is the cyclic set $\{0, 1 \ldots, n-1\}$ and

$$\forall a, b \in \mathbb{Z}_n\colon \ a + b = a + b \mod n$$

  is an abelian group, but $(\mathbb{Z}_n, \cdot)$ is not a group.

Definition 2 *Let $\mathbb{Z}_n$ be the set of integers modulo n. The set $\mathbb{Z}_n^*$, called the **set of units** of $\mathbb{Z}_n$, is the set of all the integers modulo n that have a multiplicative inverse.*

Lemma 1 *$(\mathbb{Z}_n^*, \cdot)$ is an abelian group.*

Theorem 1 ***(Fermat's Little Theorem)*** *Let p be a prime number. Then, for any $a \in \mathbb{Z}$ such that $p \nmid a$ it is true that*

$$a^{p-1} \equiv 1 \mod p.$$

Corolary 1 *Let p, q be prime numbers and $N = pq$. Let $g \in \mathbb{Z}_N^*$. Then*

$$g^{(p-1)(q-1)} \equiv 1 \mod N.$$

Proof From Fermat's Little Theorem we know that

$$(g^{p-1})^{q-1} \equiv 1 \mod q \text{ and } (g^{q-1})^{p-1} \equiv 1 \mod p.$$

Then trivially p divides $(g^{p-1})^{q-1} - 1$ and q divides $(g^{p-1})^{q-1} - 1$. Consequently pq must divide $(g^{p-1})^{q-1} - 1$, so

$$(g^{p-1})^{q-1} \equiv 1 \mod pq.$$

□

Definition 3 *Let $(A, \cdot)$ and $(B, \star)$ be two algebraic structures of the same type, a **homomorphism** is a map*

$$f : A \longrightarrow B$$

such that for $a_1, a_2 \in A$ it verifies $f(a_1 \cdot a_2) = f(a_1) \star f(a_2)$. In other words, homomorphisms are mappings that preserve certain operations between algebraic structures of the same type.

Example Consider the group of real numbers with the usual addition operation $(\mathbb{R}, +)$ and the group of positive real numbers with the product operation $(\mathbb{R}^+, \cdot)$. Then the mapping

$$\begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R}^+ \\ x & \longmapsto & e^x \end{array}$$

is a group homomorphism, since $e^{x+y} = e^x \cdot e^y$.

2.1.2 Ring theory.

Definition 4 *A **ring** is a set R together with two operations $+$ and $\cdot$ that satisfies the properties:*

- *For $+$:*
  - *Associativity: For every $r_1, r_2, r_3 \in R$:*

$$r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3.$$

  - *Commutativity:*
$$r + r' = r' + r \ \forall r, r' \in R$$
  - *Identity: There exists an element* 0*, called the identity, such that*
$$r + 0 = 0 + r = r \ \forall r \in R$$
  - *Inverses: For every* $r \in R$*, there exists an element* $-r \in R$ *such that*
$$r + -r = -r + r = e$$

- *For* $\cdot$*:*
  - *Associativity: For every* $r_1, r_2, r_3 \in R$*:*
$$r_1 \cdot (r_2 \cdot r_3) = (r_1 \cdot r_2) \cdot r_3.$$

- *For* $+$ *and* $\cdot$ *together:*
  - *Distributivity: For* $r_1, r_2, r_3 \in R$*:*
$$r_1 \cdot (r_2 + r_3) = r_1 \cdot r_2 + r_1 \cdot r_3$$
$$(r_1 + r_2) \cdot r_3 = r_1 \cdot r_3 + r_2 \cdot r_3$$

Besides:

- *If* $\cdot$ *is also commutative, i.e., if:*
$$r \cdot r' = r' \cdot r \ \forall r, r' \in R$$
then the ring is called ***commutative****.*

- *If* $\cdot$ *has an identity, i.e., if there exists an element* 1*, called the unity, such that*
$$r \cdot 1 = 1 \cdot r = r \ \forall r \in R$$
then the ring is called ***unitary****.*

Definition 5 *A unitary commutative ring* K *where for all* $0 \neq k \in K$ *there exists* $k^{-1} \in K$ *such that*
$$k \cdot k^{-1} = k^{-1} \cdot k = 1$$
is called a ***field****.*

Example The following classifications can be done for the main number sets:

- $(\mathbb{Z}, +, \cdot)$ is a commutative unitary ring. Here $+$ and $\cdot$ denote the usual addition and multiplication operations.
- $(\mathbb{Q}, +, \cdot)$ and $(\mathbb{R}, +, \cdot)$ are fields.
- $(\mathbb{Z}_n, +, \cdot)$ is a commutative unitary ring.
- $(\mathbb{Z}_p, +, \cdot)$, with p prime, is a field.

2.1.3 Polynomials.

Theorem 2 *Let K be a field. We denote by $K[x]$ the set of polynomials with coefficients in K. The following operations can be defined:*

$$\begin{array}{cccc} + : & K[x] \times K[x] & \longrightarrow & K[x] \\ & (a_0 + a_1x + \cdots + a_nx^n, b_0 + b_1x + \cdots + b_nx^n) & \longmapsto & (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_n + b_n)x^n \end{array}$$

and

$$\begin{array}{cccc} \cdot : & K[x] \times K[x] & \longrightarrow & K[x] \\ & (a_0 + a_1x + \cdots + a_nx^n, b_0 + b_1x + \cdots + b_nx^n) & \longmapsto & c_0 + c_1x + \cdots + c_nx^n \end{array}$$

where $c_\ell = \sum_{i+j=\ell} a_ib_j$. This set with the above operations is a commutative unitary ring.

Definition 6 *Let $p(x) \in K[x]$ be a polynomial. The* ***first norm*** *of $p(x)$ is the sum of its coefficients, i.e., if*

$$p(x) = a_0 + a_1x + \cdots + a_nx^n + \ldots$$

then

$$||p(x)||_1 = \sum_{i=0}^{\infty} a_i.$$

Definition 7 *Let $p(x) \in K[x]$ be a polynomial of the form*

$$p(x) = a_0 + a_1x + \cdots + a_nx^n + \ldots$$

Then the ***second norm*** *of $p(x)$ is defined as*

$$||p(x)||_2 = \sqrt{\sum_{i=0}^{\infty} (a_i)^2}.$$

Definition 8 *Let $p(x) \in K[x]$ be a polynomial. The* ***infinity norm*** *of $p(x)$ is the value of its largest coefficient, i.e., if*

$$p(x) = a_0 + a_1x + \cdots + a_nx^n + \ldots$$

then

$$||p(x)||_\infty = \max_{i \in \{0,1,\ldots,n,\ldots\}} (a_i).$$

Theorem 3 ***(Divisibility of polynomials)*** *Let $a(x), b(x) \in K[x]$ be two non-zero polynomials. There exist unique polynomials $q(x)$ and $r(x) \in K[x]$ such that*

$$a(x) = b(x) \cdot q(x) + r(x),$$

with the degree of $r(x)$ being lower than the degree of $b(x)$. The polynomials $q(x)$ and $r(x)$ are known as the quotient and remainder polynomials, respectively.

Definition 9 *Let $a(x)$ and $b(x)$ be two polynomials in $K[x]$ whose remainder from the division is $r(x) = 0$. Then $a(x)$ is said to be* ***a multiple of*** *$b(x)$ and $b(x)$ is said to* ***divide*** *$a(x)$. The set of multiples of $b(x)$ is denoted as*

$$(b(x)).$$

Definition 10 *A polynomial $a(x) \in K[x]$ is said to be* ***irreducible*** *if the only divisors of $a(x)$ in $K[x]$ are constant polynomials.*

Definition 11 *Let n be a positive integer. The* ***n-th cyclotomic polynomial*** *is the unique polynomial $\Phi(x)$ of degree n with integer coefficients that:*

- *Is irreducible.*
- *Divides $x^n - 1$.*
- *Does not divide $x^k - 1$ for any $k < n$.*

Example Some examples of cyclotomic polynomials, depending on the value of n:

- If n is prime, then:
$$\Phi_n(x) = 1 + x + \cdots + x^n - 1$$
- If n is a prime power, *i.e.*, if $n = p^k$:
$$\Phi_n(x) = \sum_{\ell=0}^{p-1} x^{\ell p^{k-1}}$$
- More concretely, if n is a power of 2, *i.e.*, if $n = 2^k$:
$$\Phi_n(x) = x^{2^{k-1}} + 1$$

Definition 12 *Let $m \in \mathbb{Z}$. A number $z \in \mathbb{C}$ is an m***-th root of unity*** *if*

$$z^m = 1.$$

These elements are of the form $z = e^{\frac{2\pi i}{m}k}$, and usually denoted as ζ_k for k in $\{1, \dots, m\}$. Moreover, an m-th root of unity is said to be ***primitive*** *if it is not a ℓ-th root of unity for any integer ℓ smaller than m. There are exactly $\phi(m)$ primitive roots of unity, where ϕ is Euler's totient function.*

There is an alternate definition for cyclotomic polynomials using the roots of unity.

Definition 13 *Let $\{\zeta_1, \dots, \zeta_{\phi(n)}\}$ be the n-th primitive roots of unity. The n-th* ***cyclotomic polynomial*** *is the polynomial:*

$$\Phi_n(x) = \prod_{k=1}^{\phi(n)} (x - \zeta_k).$$

Definition 14 *Let $K[x]$ be a ring of polynomials. An* ***ideal*** *is every subset $I \in K[x]$ such that:*

- *If $a(x), b(x) \in I$, then $a(x) + b(x) \in I$.*
- *If $a(x) \in I$, then for every $p(x) \in K[x]$: $a(x) \cdot p(x) \in I$.*

Lemma 2 *Let $K[x]$ be a ring of polynomials an $I \subset K[x]$ an ideal. Then, there is a polynomial $p(x) \in K[x]$ such that $I = (p(x))$.*

Definition 15 *Let $K[x]$ be a ring of polynomials and let $m(x) \in K[x]$. The set*

$$\frac{K[x]}{(m(x))}$$

is the set of the remainders after dividing polynomials by $m(x)$. This set with the following operations:

$$\begin{array}{rccc} + : & \frac{K[x]}{(m(x))} \times \frac{K[x]}{(m(x))} & \longrightarrow & \frac{K[x]}{(m(x))} \\ & (a(x), b(x)) & \longmapsto & a(x) + b(x) \mod (m(x)) \end{array}$$

and

$$\begin{array}{rccc} \cdot : & \frac{K[x]}{(m(x))} \times \frac{K[x]}{(m(x))} & \longrightarrow & \frac{K[x]}{(m(x))} \\ & (a(x), b(x)) & \longmapsto & a(x) \cdot b(x) \mod (m(x)) \end{array}$$

is a commutative unitary ring and is called the ***quotient ring*** *of $K[x]$ over $(m(x))$.*

Theorem 4 ***(Lagrange's Interpolation Theorem)*** *Let $\{x_i, y_i\}$ for $i \in \{0, 1, \ldots, n\}$ be pairs of points, where $x_i \neq x_j$ when $i \neq j$. Then, there exists a unique polynomial $L(x)$ of degree $deg(L) \leq n$ such that $L(x_i) = y_i$ for every $i \in \{0, 1, \ldots, n\}$. This polynomial can be computed with the following formula:*

$$L(x) = \sum_{i=0}^{n} \left(\prod_{0 \leq j \leq n, j \neq i} \frac{x - x_j}{x_i - x_j} \right) y_i.$$

2.2 The Ring-Learning with Errors problem.

The Ring-Learning with Errors (RLWE) problem is a ring adaptation of the Learning with Errors problem introduced by Oded Regev in [46]. This special form of Learning with Errors operates within the ring of polynomials over a finite field. In other words, the plaintexts are defined over

$$\mathcal{P} = \mathcal{R}_p = \frac{\mathbb{Z}_p[x]}{(\Phi(x))}.$$

where $\Phi(X)$ is a cyclotomic polynomial. Recall from Section 2.1.3 that a cyclotomic polynomial for n a power of two is of the form

$$\Phi(x) = x^n + 1.$$

In practice, usually these kind of cyclotomic polynomials are used for the RLWE implementations. Thus, assume cyclotomic polynomials of the form $x^n + 1$ for $n = 2^k$ from now on. In consequence, the plaintext ring can be rewritten as

$$\mathcal{P} = \mathcal{R}_p = \frac{\mathbb{Z}_p[x]}{(x^n + 1)}.$$

The above plaintext ring can be thought as the ring of polynomials with integer coefficients modulo p and of degree less than n.

The ciphertext ring, instead, is defined as

$$\mathcal{C} = \mathcal{R}_q \times \mathcal{R}_q,$$

where

$$\mathcal{R}_q = \frac{\mathbb{Z}_q[x]}{(x^n+1)}.$$

for q another prime number that typically is much greater than p.

RLWE's security strongly depends on the generation of polynomials with random coefficients that are sufficiently small. In order to measure if a polynomial has small coefficients, the infinity norm from Definition 8 is used. There are various approaches for generating small polynomials, but the mostly accepted one is to randomly choose the coefficients using a Discrete Gaussian Distribution χ with parameters

$$(\mu, \sigma, \beta)$$

where:

- μ is the mean of the distribution. The homomorphic encryption standard [2] fixes it to $\mu = 0$.
- σ is the standard deviation. The homomorphic encryption standard [2] fixes it to $\sigma = \frac{8}{\sqrt{2\pi}} \approx 3.2$.
- β is the integer limit for a coefficient value. In other words, the Gaussian Distribution sampling is limited to picking values up to β. The homomorphic encryption standard [2] fixes it to $\beta = 19$.

This way, a polynomial $e(x)$ with random small coefficients is generated, this is expressed as $e(x) \leftarrow \chi$.

Let us define the Ring-Learning with Errors problem. Let

- $c_i(x)$ with $i \in \{1, \dots, n\}$ a set of tuples of polynomials in $\mathcal{R}_q$.
- $e_i(x) \leftarrow \chi$ with $i \in \{1, \dots, n\}$.
- $sk(x) \leftarrow \chi$.
- $m_i(x) = c_i(x) \cdot sk(x) + e_i(x)$ for $i \in \{1, \dots, n\}$.

The RLWE problem consists of finding $sk(x)$ given the tuples $(m_i(x), c_i(x))$ for $i \in \{1, \dots, n\}$. If $\Phi(x)$ is a cyclotomic polynomial, the RLWE problem is considered equivalent to solving the Approximate Shortest Vector Problem (α-SVP) [37]. The α-SVP problem is generally considered hard to solve [38].

Chapter 3

Homomorphic Encryption.

Let $\mathcal{P}$ be the algebraic structure containing the plaintexts and $\mathcal{C}$ the algebraic structure containing the ciphertexts. An encryption system is homomorphic if there exists a homomorphism between $\mathcal{P}$ and $\mathcal{C}$. From Definition 3, the operations on the plaintexts preserved by the homomorphism may not correspond to the same operations over the ciphertexts. With this in mind, when writing *homomorphically calculate/compute* an operation, we are referring to the operation on the plaintexts that is preserved by the homomorphism.

Thus, homomorphic encryption (HE) allows certain types of operations to be performed on the ciphertexts obtaining the same result as if they were computed in clear. However, it is not possible to carry out operations that are not linear combinations of those preserved by the homomorphism. For this reason, when choosing which cipher to use for each use case, certain criteria must be considered:

- The operations to be homomorphically computed.
- The number of times these operations must be carried out to obtain the result (for example, n additions and 1 product).
- The algebraic structure of the plaintexts (integers, real numbers...).
- The security of the encryption system.

This chapter intends to give an approach to the main HE schemes that have been developed in the last years. With this purpose, Section 3.1 introduces the first proposed HE schemes chronologically until the first Fully Homomorphic Encryption (FHE) scheme was introduced by Craig Gentry [24]. Section 3.2 describes predominant FHE schemes over the integers: the one proposed by Brakerski, Gentry and Vaikuntanathan (BGV) [11] and the one from Brakerski, Fan and Vercauteren (BFV) [21]. In Section 3.3, the scheme proposed by Cheon, Kim, Kim and Song (CKKS) [15] is introduced. This scheme allows to perform operations over complex numbers, thus generalizing the BGV and BFV approach. Lastly, FHEW [19] and its improvement, the Fully Homomorphic Encryption over the Thorus (TFHE) approach [16] are described in Section 3.4. These schemes are specially developed to rapidly compute boolean gates instead of arithmetic operations.

3.1 The first HE schemes.

The first to refer to the homomorphic feature of some ciphers were Rivest, Adleman and Dertouzos in [47]. It was then that the study of homomorphic ciphers gained attention, since another potential began to be seen in this type of cryptographic systems. As a consequence, the first HE schemes were designed.

These first schemes were only homomorphic for one operation, and thus denoted as *partially homomorphic (PHE)*. The most famous PHE scheme is the RSA cryptosystem, which was introduced by Rivest, Shamir and Adleman in 1977 [48], so it is previous to the notion of homomorphic encryption. However, it naturally allows to homomorphically perform an arbitrary number of multiplications.

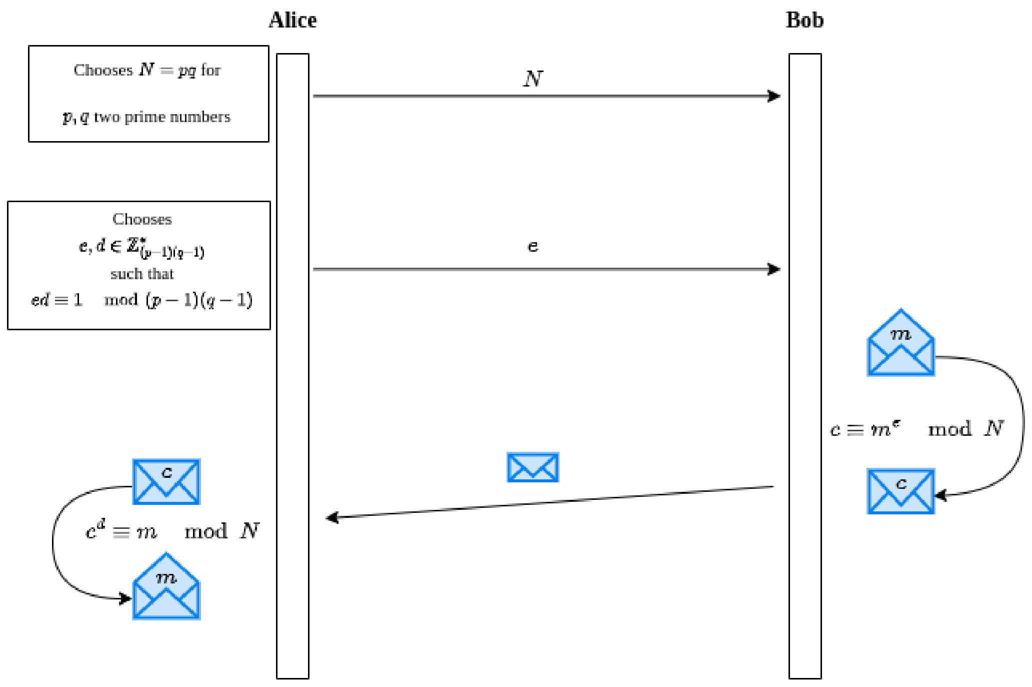

Alice
Bob
Chooses $N = pq$ for p, q two prime numbers
N
Chooses $e, d \in \mathbb{Z}^*_{(p-1)(q-1)}$ such that $ed \equiv 1 \mod (p-1)(q-1)$
e
m
$c \equiv m^e \mod N$
c
c
$c^d \equiv m \mod N$
m

Figure 3.1: RSA scheme sequence diagram.

3.1.1 The RSA scheme.

In Figure 3.1 a brief reminder is introduced on how the RSA encryption scheme works. The decryption follows because

$$c^d = m^{ed} = m^{1+\lambda(p-1)(q-1)} = m \cdot (m^{(p-1)(q-1)})^\lambda \equiv m \cdot 1^\lambda = m \mod N$$

and the congruence follows from Corollary 1. The security of this scheme is based on the fact that, given a product of two unknown primes N, a value e and an element $c \in \mathbb{Z}_N^*$, finding $m \in \mathbb{Z}_N^*$ such that $c \equiv m^e \mod N$ can be reduced to decomposing N in prime factors. The integer factorization problem is considered very dificult to solve in practice.

The homomorphism on the RSA scheme. Let us denote the encryption of a message m as $E(m)$. Then, for the RSA scheme:

$$E(m) = m^e \equiv c \mod N.$$

The above encryption function is a homomorphism for the product of two messages. Indeed, observe that

$$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e = E(m_1 \cdot m_2).$$

This means one can homomorphically compute an unlimited number of products using RSA. The classical RSA scheme (the one explained in this chapter), though, is not secure enough to be used in practice. To increase its security, several random digits are padded on the left of plaintexts before encrypting them:

$$m' = \{\text{random digits } |m\}.$$

This scheme is called the Padded RSA. The Padded RSA encryption function loses the homomorphic property.

Other PHE schemes. After the RSA scheme, other PHE systems were introduced. One of the most famous ones is the ElGamal cryptosystem [20]. The security of this scheme is based on the discrete logarithm problem, and the encryption function is homomorphic for the product, same as in RSA. Another known partially homomorphic scheme is the Paillier scheme, introduced by Pascal Paillier in [44]. This scheme is based on the complexity of computing n-th residue classes, and in this case it allows to homomorphically compute an arbitrary number of additions.

Somewhat Homomorphic Encryption systems. Somewhat Homomorphic Encryption (SHE) schemes are ciphers very similar to PHE. The former, however, are no longer homomorphic for a single operation, but allow both additions and multiplications to be performed homomorphically. The big drawback of these schemes is that one of these operations can only be performed a limited number of times. Indeed, to guarantee the security of the messages, it is necessary to introduce noise on the ciphertexts. Thus, despite one of the operations (usually addition) is not affected by the noise, the other (usually multiplication) is. As a result, if the limited operation is overused in an SHE scheme, the result of the homomorphic computation is incorrect.

3.1.2 Gentry's Fully Homomorphic Encryption scheme.

Craig Gentry was the first to build a fully homomorphic encryption (FHE) scheme in [24]. Gentry devised a new technique to transform a somewhat homomorphic scheme into an FHE one. Recall that, if a scheme is somewhat homomorphic, then it is possible to evaluate a function f homomorphically if it can be expressed as a combination of the operations that the scheme preserves. In that case, the homomorphic evaluation of f can be illustrated as shown in Figure 3.2.

$$\begin{tikzcd}
Enrcypt_{pk}(m) = c_m \arrow[r] & \boxed{f(c_m)} \arrow[r] & Enrcypt_{pk}(f(m))
\end{tikzcd}$$

Figure 3.2: Homomorphic evaluation of a function f.

Following the procedure of Figure 3.2, Gentry's idea was to reduce the noise present in a ciphertext by homomorphically evaluating the decryption function using an encryption of the secret key during the evaluation. This noise reduction technique is known today as bootstrapping.

Following the scheme of Figure 3.2, the bootstrapping operation works as follows:

$$\begin{tikzcd}
 & Enrcypt_{pk_1}(m) = c_m \arrow[d] & \\
Enrcypt_{pk_2}(sk_1) = c_{sk_1} \arrow[r] & \boxed{Decrypt(c_{sk_1})} \arrow[r] & \begin{array}{c} Enrcypt_{pk_2}(Decrypt(sk_1, c_m)) \\ = Encrypt_{pk_2}(m) \end{array}
\end{tikzcd}$$

Figure 3.3: The bootstrapping protocol proposed by Gentry.

In figure 3.3, sk_1 and sk_2 are two different secret keys and respectively pk_1 and pk_2 are their public keys. Note that the result of the bootstrapping of c_m, encrypted using pk_1, is a ciphertext corresponding to the encryption of m using pk_2.

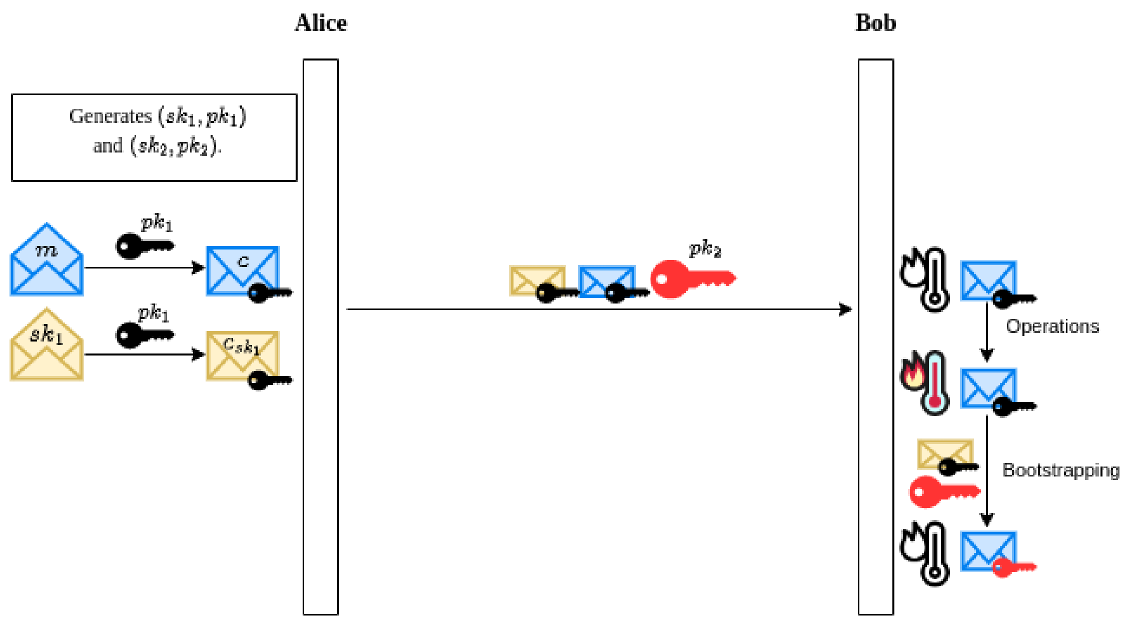


Figure 3.4: Gentry's noise reduction protocol.

Bootstrapping can be used whenever the decryption function can be expressed as a combination of the operations that are preserved due to the homomorphisms in the scheme. Gentry presented an SHE scheme with this special property. The idea of the bootstrapping is that decryption removes the noise on the ciphertext, so the noise present in a ciphertext is reset to the initial value when the bootstrapping is finished. Thus, Gentry used the bootstrapping to control the noise of the ciphertexts in his scheme, as can be seen in Figure 3.4.

At Bob's side, it can be seen how the noise (represented as termometers at the left of the messages) is reduced because of the bootstrapping. Figure 3.4 shows that the bootstrapping is in fact a key switching between sk_1 and sk_2.

In the years following this discovery, new FHE schemes began to appear. In fact, since the appearance of Gentry's work, FHE schemes have experienced an explosion of interest, as can be seen in Figure 3.5. This figure shows the evolution of FHE schemes over time.

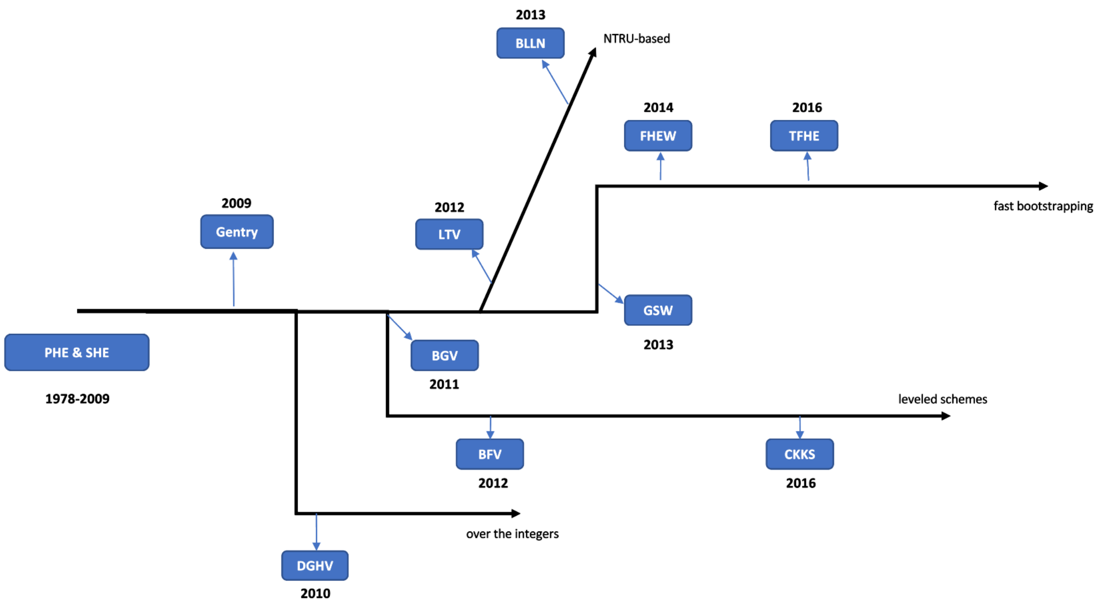


Figure 3.5: Timeline of the appearance of the main FHE schemes.

In Figure 3.5 it can be observed that several approaches have been followed in order to build new FHE schemes after Gentry's scheme was published. Nowadays, mainly two lines are being researched: the leveled homomorphic encryption schemes and the fast bootstrapping schemes.

Leveled FHE schemes are fully homomorphic schemes that can be configured beforehand to be able to homomorphically perform a concrete number of arithmetic operations without needing to bootstrap during the process. The configuration is set up via a cryptographic context, where the multiplicative depth (the number of multiplications that will be computed) is fixed. Thus, these schemes need to know beforehand what kind of computation is to be performed on the ciphertexts. There are two families of well-known leveled FHE schemes: the ones based on integer arithmetic, such as BGV and BFV, and the ones based on complex number approximations, such as CKKS.

The FHE schemes based on the fast bootstrapping approach are focused on reducing the cost of the bootstrapping operation. The principal schemes in this approach are based on bit-wise encryption, while leveled schemes tend to be block-ciphers. The first scheme of this kind, FHEW, was presented by Ducas and Micciancio in [19], but the TFHE approach [16] improved its performance by several orders of magnitude.

3.2 BGV and BFV.

The BGV and BFV schemes are based on the Ring-Learning with Errors (RLWE) problem. Recall from the RLWE problem (see chapter 2.2) that the plaintext space and ciphertext space were:

$$\mathcal{P} = \mathcal{R}_p = \frac{\mathbb{Z}_p[x]}{x^n+1}, \mathcal{C} = \mathcal{R}_q \times \mathcal{R}_q = \frac{\mathbb{Z}_q[x]}{x^n+1} \times \frac{\mathbb{Z}_q[x]}{x^n+1}.$$

BGV and BFV allow to perform additions and products over encrypted integers. Both schemes are very similar, so, in this chapter, BFV is firstly introduced and then the differences between both schemes are described. The BFV scheme is formed by the following subprotocols:

- Encoding and decoding of plaintexts.
- Key generation.
- Encryption and decryption.
- Homomorphic operations.

In the following chapters, the processes above are described. It is important to note that all the polynomial operations below are assumed to be done modulo $x^n + 1$.

3.2.1 Encoding and decoding of plaintexts.

Messages in BFV are assumed to be integer vectors whose components are bounded modulo p, the same prime number defining the plaintext space $\mathcal{P} = \mathcal{R}_p$. In order to encode messages, the following encoding function is used:

$$\begin{array}{rccc} \texttt{Encode}: & \mathbb{Z}_p^n & \longrightarrow & \mathcal{P} \\ & (m_0, \dots, m_{n-1}) & \longmapsto & m_0 + m_1x + \dots + m_{n-1}x^{n-1} \end{array}$$

Following this encoding function, up to n elements can be encoded simultaneously, where n is the degree of the cyclotomic polynomial. By convention, if $k < n$ elements want to be encoded, then the last $n - k$ elements are set to 0.

The decoding function works similarly:

$$\begin{array}{rccc} \texttt{Decode}: & \mathcal{P} & \longrightarrow & \mathbb{Z}_p^n \\ & m_0 + m_1x + \dots + m_{n-1}x^{n-1} & \longmapsto & (m_0, \dots, m_{n-1}) \end{array}$$

3.2.2 Key generation.

The key generation consists of two processes. The first one is the generation of the secret key $sk(x)$, and the second is the derivation of the public key $pk(x)$ from $sk(x)$:

- `SecretKeyGen`: sample $sk(x) \leftarrow \chi$.

- `PublicKeyGen`: generate a random $a(x)$ from $\mathcal{R}_q$ and $e(x) \leftarrow \chi$. Then

$$\mathbf{pk} = (pk_1(x), pk_2(x))$$

  where

  - $pk_1(x) = -(a(x) \cdot sk(x) + e(x)) \mod q$
  - $pk_2(x) = a(x)$

3.2.3 Encryption and decryption.

Let us assume a message $m \in \mathbb{Z}_p^n$ has been encoded as $m(x) \in \mathcal{P}$, and let $\Delta = \lfloor \frac{q}{p} \rfloor$. The first thing to do is sampling $u(x), e_1(x), e_2(x) \leftarrow \chi$. With these new polynomials and **pk**, the encryption of $m(x)$ works as follows:

1. Compute $c_1(x) = pk_1(x) \cdot u(x) + e_1(x) + \Delta \cdot m(x) \mod q$.
2. Compute $c_2(x) = pk_2(x) \cdot u(x) + e_2(x) \mod q$.

Then, the final ciphertext is $\mathbf{c} = (c_1(x), c_2(x))$. The decryption method consists of the next steps:

1. Compute $r_1(x) = c_1(x) + c_2(x) \cdot sk(x) \mod q$.
2. Compute $r_2(x) = \frac{p}{q} \cdot r_1(x)$
3. Round the coefficients to their nearest integer: $r_3(x) = \lfloor r_2(x) \rceil$.
4. Then, $m(x) = r_3(x) \mod p$.

Let us see why this protocol results in $m(x)$. The key part is to see what happens in Step 1:

$$\begin{aligned} c_1(x) + c_2(x) \cdot sk(x) &= [pk_1(x) \cdot u(x) + e_1(x) + \Delta \cdot m(x)] + [pk_2(x) \cdot u(x) + e_2(x)] \cdot sk(x) \\ &= \begin{array}{l} -(\cancel{a(x) \cdot sk(x)} + e(x)) \cdot u(x) + e_1(x) + \Delta \cdot m(x) + \\ \qquad + \cancel{a(x) \cdot u(x) \cdot sk(x)} + e_2(x) \cdot sk(x) \end{array} \\ &= -e(x) \cdot u(x) + e_1(x) + \Delta \cdot m(x) + e_2(x) \cdot sk(x) \\ &\equiv \Delta \cdot m(x) + \aleph(x) \mod q \end{aligned}$$

Note that in the last congruence this notation was introduced:

$$\aleph(x) \equiv -e(x) \cdot u(x) + e_1(x) + e_2(x) \cdot sk(x) \mod q.$$

Therefore, $\aleph(x)$ denotes the overall noise that is present after performing Step 1 of the decryption protocol. From Section 2.2, the infinity norm of $e(x), u(x), e_1(x), e_2(x)$ and $sk(x)$ is bounded by β. With this in mind and using Definition 8, it is direct to show that $||\aleph(x)||_\infty \leq n\beta^2 + \beta + n\beta^2 = 2n\beta^2 + \beta$.

Proof The proof for $||e_1(x)||_\infty \leq \beta$ is direct from the definition of sampling $e_1(x) \leftarrow \chi$, and the proofs for $-e(x) \cdot u(x)$ and $e_2(x) \cdot sk(x)$ are equivalent. Let us see that $||e_2(x) \cdot sk(x)||_\infty \leq n\beta^2$. For such a proof, consider the worst case of sampling both polynomials, *i.e.*:

- $e_1(x) = \beta + \beta x + \cdots + \beta x^n$.
- $sk(x) = \beta + \beta x + \cdots + \beta x^n$.

In this situation, the product of both polynomials results in:

$$\begin{array}{rcl} e_1(x) \cdot sk(x) & = & (\beta + \beta x + \cdots + \beta x^n) \cdot (\beta + \beta x + \cdots + \beta x^n) \\ & = & \beta(\beta + \beta x + \cdots + \beta x^n) \\ & + & \beta x(\beta + \beta x + \cdots + \beta x^n) \\ & + & \cdots \\ & \vdots & \ddots \\ & + & \beta x^n(\beta + \beta x + \cdots + \beta x^n) \end{array}$$

Observe from the last equality above that there is at least one monomial with degree n in each of the products: $\beta \cdot \beta x^n$, $\beta x \cdot \beta x^{n-1}$, ..., $\beta x^n \cdot \beta$. Since this monomial appears in all of the products, its resulting coefficient will be a maximum coefficient of the polynomial. This coefficient is clearly:

$$\beta^2 + \overset{n}{\cdots} + \beta^2 = n\beta^2.$$

□

Write $r_1(x) = \Delta m(x) + \frac{p}{q}\aleph(x) + p \cdot r(x)$. Continuing now with Step 2, then $r_2(x) = \frac{p}{q} r_1(x) = m(x) + \frac{p}{q}[\aleph(x) - \epsilon \cdot m(x)] + p \cdot r(x)$. The rounding operation in Step 3 removes $\frac{p}{q}[\aleph(x) - \epsilon \cdot m(x)]$, and the last modular operation by p in Step 4 removes $p \cdot r(x)$, obtaining as the result of the decryption $m(x)$.

As can be seen from the process, the key fact for a correct decryption is the rounding in Step 3. For the rounding to be correct, $\frac{t}{q}||\aleph(x) - \epsilon \cdot m(x)||_\infty < \frac{1}{2}$ must be ensured.

3.2.4 Homomorphic operations.

Let $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$ be two different ciphertexts. Two homomorphic operations are considered in the BFV scheme: the addition and the product. The addition of $\mathbf{c}$ and $\mathbf{c}'$ is straightforward:

$$\texttt{Add}(\mathbf{c}, \mathbf{c}') = [c_1(x) + c_1'(x), c_2(x) + c_2'(x)] = [c_1''(x), c_2''(x)].$$

Indeed, expanding the expresion on the right for $c_1''(x)$:

$$\begin{array}{rcl} c_1''(x) & = & pk_1(x) \cdot [u(x) + u'(x)] + [e_1(x) + e_1'(x)] + \Delta \cdot [m(x) + m'(x)] \\ & = & pk_1(x) \cdot u''(x) + e_1''(x) + \Delta \cdot (m(x) + m'(x)). \end{array}$$

As can be seen in the above equation, $c_1''(x)$ results in a ciphertext encrypting $m(x) + m'(x)$. For $c_2''(x)$ something similar happens:

$$\begin{array}{rcl} c_2''(x) & = & pk_2(x) \cdot [u(x) + u'(x)] + [e_2(x) + e_2'(x)] \\ & = & pk_2(x) \cdot u''(x) + e_2''(x). \end{array}$$

Lastly, the homomorphic product is shown. For this product to preserve the integrity of the multiplication on the plaintexts, it must be scaled by $\frac{t}{q}$:

$$\texttt{Mult}(\mathbf{c}, \mathbf{c}) = \frac{t}{q}[c_1(x)c_1'(x), c_2(x)c_2'(x)] = [\frac{t}{q}c_1''(x), \frac{t}{q}c_2''(x)].$$

Let us see that the operation is preserved in $c''_1(x)$:

$$\begin{aligned}
c''_1(x) &= \tfrac{t}{q}[(pk_1(x))^2(u(x)u'(x))] + \tfrac{t}{q}[e_1(x)e'_1(x)] + \tfrac{t}{q}[\Delta^2(m(x)m'(x))] \\
&= pk_1(x) \cdot [\tfrac{t}{q}pk_1(x)u(x)u'(x)] + [\tfrac{t}{q}e_1(x)e'_1(x)] + \Delta \cdot [m(x)m'(x)] \\
&= pk_1(x) \cdot u''(x) + e''_1(x) + \Delta \cdot [m(x)m'(x)]
\end{aligned}$$

Thus, $c''_1(x)$ stores a ciphertext encrypting message $m(x) \cdot m'(x)$. Observe that the scaling by $\frac{t}{q}$ is necessary to preserve the encryption of $m(x) \cdot m'(x)$ and not $\Delta \cdot m(x) \cdot m'(x)$. For $c''_2(x)$, analogously:

$$\begin{aligned}
c''_2(x) &= \tfrac{t}{q}[(pk_2(x))^2u(x)u'(x)] + \tfrac{t}{q}[e_2(x)e'_2(x)] \\
&= pk_2(x)[\tfrac{t}{q}pk_2(x)u(x)u'(x)] + [\tfrac{t}{q}e_2(x)e'_2(x)] \\
&= pk_2(x) \cdot u''(x) + e''_2(x).
\end{aligned}$$

3.2.5 Differences between BFV and BGV.

BGV is very similar to BFV, but there are some slight differences between both schemes. Let us introduce them following the order of presentation of BFV:

- Key generation.
- Encryption and decryption.
- Homomorphic operations.

Observe that the encoding and decoding of plaintexts is not included in the above list. This is because both schemes use the same encoding and decoding techniques.

Key generation.

Whereas the intrinsic idea is the same, some slight details make the key generation of BGV to be different from the one in BFV:

- `SecretKeyGen`: sample $sk(x) \leftarrow \chi$.
- `PublicKeyGen`: generate a random $a(x)$ from $\mathcal{R}_q$ and $e(x) \leftarrow \chi$. Then

  $$\mathbf{pk} = (pk_1(x), pk_2(x))$$

  where

  - $pk_1(x) = -(a(x) \cdot sk(x) + p \cdot e(x)) \mod q$.
  - $pk_2(x) = a(x)$.

  Recall that p is the prime number defining $\mathcal{P} = \mathcal{R}_p$.

As can be seen above, BGV scales the error by p. In Figure 3.6 the meaning of this scaling is studied.

Encryption and decryption.

Consider a message $m \in \mathbb{Z}_p^n$ that has been encoded as $m(x) \in \mathcal{P}$. Let **pk** be the public key from the key generation process. For the encryption in BGV, again sample $u(x), e_1(x), e_2(x) \leftarrow \chi$. Then $\mathbf{c} = (c_1(x), c_2(x))$ where

- $c_1(x) = pk_1(x) \cdot u(x) + p \cdot e_1(x) + m(x) \mod q$
- $c_2(x) = pk_2(x) \cdot u(x) + p \cdot e_2(x) \mod q$

At this point, two differences have arised. Firstly, $m(x)$ is not scaled by Δ in BGV. Secondly, the error ($e_1(x)$ and $e_2(x)$) is scaled by p.

In order to understand the reason for the differences during key generation and encryption of BGV and BFV, it is necessary to take a look at how these differences modify the resulting ciphertext. Figure 3.6 shows how the different elements are located in the ciphertexts of both schemes.

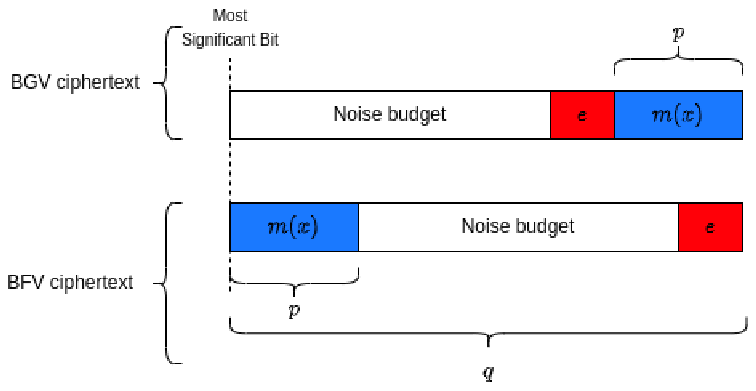


Figure 3.6: Shape of the BGV and BFV ciphertexts (adapted from Figure 1 in [15]).

As can be seen in Figure 3.6, in BGV, not scaling $m(x)$ by Δ means having $m(x)$ in the lowest significant bits of the ciphertext. Besides, since the error is scaled by p in this scheme, the final error is located just in the bits at the left of $m(x)$, and the most significant bits are reserved for this error (the noise) to grow. This is different in BFV, since $m(x)$ is located at the most significant bits. The error occupies the least significant bits (it is not scaled in this scheme) and the bits in the middle are reserved for this error to grow.

The decryption method in BGV is actually very similar to the one in BFV. However, in this case it is more direct:

1. Compute $r_1(x) = c_1(x) + c_2(x) \cdot sk(x) \mod q$.
2. Then, $m(x) = r_1(x) \mod p$.

In more detail, from Step 1 we have:

$$\begin{aligned}
c_1(x) + c_2(x)\cdot sk(x) &= [pk_1(x)\cdot u(x) + p\cdot e_1(x) + m(x)] + [pk_2(x)\cdot u(x) + p\cdot e_2(x)]\cdot sk(x)\\
&= -(\cancel{a(x)\cdot sk(x)} + p\cdot e(x))\cdot u(x) + p\cdot e_1(x) + m(x) + \\
&\qquad\qquad \cancel{+a(x)\cdot u(x)\cdot sk(x)} + p\cdot e_2(x)\cdot sk(x)\\
&= -p\cdot e(x)\cdot u(x) + p\cdot e_1(x) + m(x) + p\cdot e_2(x)\cdot sk(x)\\
&\equiv m(x) + p\cdot\aleph(x) \mod q
\end{aligned}$$

where $\aleph(x)$ is the same as in BFV. It is now easy to see that, reducing $m(x) + p\cdot\aleph(x)$ modulo p gives $m(x)$ as wanted. However, for this to happen, it is necessary that $||\aleph||_\infty < \frac{q}{2p}$. Indeed, if the infinity norm of $\aleph(x)$ surpasses that value, then:

$$||p\cdot\aleph||_\infty \geq \frac{q}{2} \Rightarrow m(x) + p\cdot\aleph(x) \geq m(x) + \frac{q}{2} \equiv m(x) + 1 \mod q.$$

Homomorphic operations.

Let $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$ be two different BGV ciphertexts. Same as in BFV, in BGV the addition and the product are the preserved operations. The addition is analogous to the one in BFV:

$$\texttt{Add}(\mathbf{c}, \mathbf{c}') = [c_1(x) + c_1'(x), c_2(x) + c_2'(x)] = [c_1''(x), c_2''(x)].$$

Expanding the expresion for $c_1''(x)$, we have:

$$\begin{aligned}
c_1''(x) &= pk_1(x)\cdot[u(x) + u'(x)] + p\cdot[e_1(x) + e_1'(x)] + [m(x) + m'(x)]\\
&= pk_1(x)\cdot u''(x) + p\cdot e_1''(x) + (m(x) + m'(x)).
\end{aligned}$$

Similarly, for $c_2''(x)$:

$$\begin{aligned}
c_2''(x) &= pk_2(x)\cdot[u(x) + u'(x)] + p\cdot[e_2(x) + e_2'(x)]\\
&= pk_2(x)\cdot u''(x) + p\cdot e_2''(x).
\end{aligned}$$

In the case of multiplication, recall that in BFV it had to be scaled by $\frac{t}{q}$. In BGV, this is not needed. The product can be computed directly:

$$\texttt{Mult}(\mathbf{c}, \mathbf{c}') = [c_1(x)c_1'(x), c_2(x)c_2'(x)] = [c_1''(x), c_2''(x)].$$

In fact, expanding $c_1''(x)$:

$$\begin{aligned}
c_1''(x) &= [(pk_1(x))^2(u(x)u'(x))] + [p^2 e_1(x)e_1'(x)] + [(m(x)m'(x))]\\
&= pk_1(x)\cdot[pk_1(x)u(x)u'(x)] + p[pe_1(x)e_1'(x)] + [m(x)m'(x)]\\
&= pk_1(x)\cdot u''(x) + pe_1''(x) + [m(x)m'(x)]
\end{aligned}$$

For $c_2''(x)$ we have something similar:

$$\begin{aligned}
c_2''(x) &= [(pk_2(x))^2 u(x)u'(x)] + [p^2 e_2(x)e_2'(x)]\\
&= pk_2(x)[pk_2(x)u(x)u'(x)] + p[pe_2(x)e_2'(x)]\\
&= pk_2(x)\cdot u''(x) + e_2''(x).
\end{aligned}$$

3.3 CKKS.

The Cheon-Kim-Kim-Song (CKKS) scheme [15] is an extension of the BGV scheme that allows to compute homomorphically over real numbers based on the RLWE problem. The key contribution of this scheme are the encoding and decoding functions, whiche make it possible to compute over the real numbers while using integer-coefficient polynomials.

The $Q(m, f)$ representation. In order to understand how the encoding and decoding functions work, it is a must to introduce the $Q(m, f)$ representation of fixed-point or floating point numbers. This representation is standardized [28] and typically used in different systems. The idea behind this representation is that in $Q(m, f)$ the fixed-point numbers are approximated to representatives that have the integral part between -2^m and 2^m, while their decimal (or fractional) part is defined by a step of 2^{-f}. This way, in a $Q(m, f)$ representation, we have representatives for the following numbers:

$$\{-2^m, -2^m + 2^{-f}, -2^m + 2 \cdot 2^{-f}, \ldots, 2^m - 2 \cdot 2^{-f}, 2^m - 2^{-f}\}.$$

Following the above formula, as an example, a $Q(4, 7)$ representation allows to define numbers between $[-16, 16)$ with a step of $2^{-7} = 0.0078125$. Observe that, in this case, there are representatives for real numbers between $[-16, 16)$ with a precision of two decimal digits:

$$\{-16, -15.9921875, -15.984375, \ldots, 15.984375, 15.9921875\}.$$

As a result, the conversion from a real number to a $Q(m, f)$ fixed-point number is easy:

$$\texttt{ToQmf}(r) = \lfloor r \cdot 2^f \rceil,$$

and the conversion back to a real number can be done by simply dividing by 2^f:

$$\texttt{ToReal}(r') = \frac{r'}{2^f}.$$

The CKKS encoding and decoding functions. The CKKS encoding and decoding functions follow the ideas from the $Q(m, f)$ conversion functions, but are intrinsically different to those ones, mainly because the plaintext and ciphertext spaces in CKKS are the same ones as in BGV and BFV:

$$\mathcal{P} = \mathcal{R}_p = \frac{\mathbb{Z}_p[x]}{x^n + 1}, \mathcal{C} = \mathcal{R}_q \times \mathcal{R}_q = \frac{\mathbb{Z}_q[x]}{x^n + 1} \times \frac{\mathbb{Z}_q[x]}{x^n + 1}.$$

As a consequence, a function that turns real numbers into integer coefficient polynomials is needed. The function used for this purpose in the CKKS scheme is the *complex canonical embedding.* The complex canonical embedding maps integer coefficient polynomials into vectors of complex numbers. The definition of the complex canonical embedding is the following:

$$\begin{array}{rccc} \sigma : & \mathcal{R}_p & \longrightarrow & \mathbb{C}^{\frac{n}{2}} \\ & f(x) & \longmapsto & (f(\zeta_1), f(\zeta_2), \ldots, f(\zeta_{\frac{n}{2}})) \end{array}$$

where $\{\zeta_1, \ldots, \zeta_{\frac{n}{2}}\}$ are the n-th primitive roots of unity defining $x^n + 1$ [1]. It is important to note that, since n is a power of two, then $\phi(n) = \frac{n}{2}$ and thus the number of primitive roots of unity is $\frac{n}{2}$. The complex canonical embedding is used for decoding in CKKS as follows:

$$\begin{array}{rccc} \texttt{Decode}: & \mathcal{R}_p & \longrightarrow & \mathbb{C}^{\frac{n}{2}} \\ & f(x) & \longmapsto & \sigma(\frac{1}{2^f} \cdot f(x)) \end{array}$$

Note that the scaling from the $Q(m, f)$ representation is done before applying the canonical embedding to the plaintexts. The idea for the encoding is similar, but in this case using the inverse of the complex canonical embedding:

$$\begin{array}{rccc} \texttt{Encode}: & \mathbb{C}^{\frac{n}{2}} & \longrightarrow & \mathcal{R}_p \\ & (z_1, \ldots, z_{\frac{n}{2}}) & \longmapsto & \lfloor 2^f \cdot \sigma^{-1}(z_1, \ldots, z_{\frac{n}{2}}) \rceil \end{array}$$

Such an encoding can be computed using polynomial interpolation (see Theorem 4).

Final notes on the CKKS scheme. Once the encoding and decoding functions are introduced, the CKKS scheme is very similar to the BGV scheme. The encryption and decryption functions are analogous to the ones of that scheme, and the same happens with the homomorphic additions and products. We refer the reader to [15] to see the details for a deeper understanding, but do not describe the scheme because it will not be used in this work.

It is important to note, though, that the BGV and BFV schemes can also be used for representing fixed point numbers by simply scaling by a factor $\alpha = 2^f$ when encoding the messages. The only thing to consider is that p must be large enough to hold the encoded messages with the wanted precision (defined by f). However, for this purpose it is better to use CKKS, since the encoding and decoding functions for fixed-point numbers in BGV and BFV result in less efficient operations and schemes [17].

3.4 TFHE.

FHEW [19] and TFHE [16] were born during the research of homomorphic schemes with efficient bootstrapping operations. Recall from Section 3.1.2 that the bootstrapping operation is intended to refresh the noise of the ciphertexts. This way, if a ciphertext has too much noise, after bootstrapping this noise is reduced and one is able to continue computing over that ciphertext without losing the encrypted value inside it. The first homomorphic schemes pursuing this idea were Gentry-Sahai-Waters (GSW) [25] and FHEW [19], but in this document only TFHE [16] is introduced, due to its better performance and current usage.

The TFHE cipher is focused on the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ and its variations. The torus actually is the set $\mathbb{T} = [0, 1)$, *i.e.*, the real numbers modulo 1. This set has not a ring structure, since the product is not well-defined. It is possible, though, to define an external product between torus elements and elements in $\mathbb{Z}$, giving the torus a $\mathbb{Z}$-module structure.

[1] See Definitions 12 and 13 for more details on the roots of unity and their relation with cyclotomic polynomials

Variations of the torus. TFHE is not exactly applying the torus in the encryption, but *the discretized torus polynomials*. For the reader to understand them, it is necessary to firstly introduce the discretized torus and then its generalization to polynomials.

Let $2 \leq z \in \mathbb{Z}$ and let $t \in \mathbb{T}$. Then there exist $\{t_1, t_2, \dots\}$ such that:

$$t = \sum_{i=1}^{\infty} \frac{t_i}{z^i}.$$

In practice, these expansions can be useful up to some fixed precision. Given $2 \leq z \in \mathbb{Z}$ and $\omega \geq 1$, then any $t \in \mathbb{T}$ can be approximated as follows:

$$t \approx \sum_{i=1}^{\omega} \frac{t_i}{z^i} \text{ with } t_i \in \{0, \dots, z-1\}.$$

The above representation limits the torus to the subset $\{0, \frac{1}{z^\omega}, \dots, \frac{z^\omega - 1}{z^\omega}\}$. It is usual to fix $z = 2$ and then call ω the bit precision, which is fixed to $\omega = 32$ or $\omega = 64$ bits too. In this setting, the set $\mathbb{T}_b = \{0, \frac{1}{b}, \dots, \frac{b-1}{b}\}$ where $b = 2^\omega$, is a submodule of $\mathbb{T}$ and is called the discretized torus.

The discretized torus (as well as the torus) can be generalized to polynomials whose coefficients belong to $\mathbb{T}_b$ (similarly to $\mathbb{T}$). The discretized torus polynomials are defined as:

$$\mathbb{T}_{n,b}[x] = \frac{\mathbb{T}_b[x]}{x^n + 1},$$

where $x^n + 1$ is cyclotomic. Let $\mathbb{Z}_{n,b}[x] = \frac{\mathbb{Z}_b[x]}{x^n+1}$ with $\mathbb{Z}_b = \mathbb{Z}/b\mathbb{Z}$. Observe that any polynomial $f(x) \in \mathbb{T}_{n,b}[x]$ can be expressed as $f(x) = g(x) \cdot \frac{1}{b}$ for some $g(x) \in \mathbb{Z}_{n,b}[x]$. This set of polynomials, analogously to $\mathbb{T}_b$, has two operations: the natural polynomial addition and the external product by a polynomial in $\mathbb{Z}_{n,b}[x]$.

The TFHE scheme. In TFHE, several schemes can be defined, depending on the plaintext and ciphertext spaces chosen. However, the scheme that actually allows to create a fully homomorphic encryption scheme is the one where the plaintext and ciphertext spaces are

$$\mathcal{P} = \mathbb{T}_{n,a}[x] \text{ and } \mathcal{C} = \mathbb{T}_{n,b}[x]^{k+1},$$

where $a|b$ and $\mathbb{T}_{n,b}[x]^{k+1} = \mathbb{T}_{n,b}[x] \times \overset{k+1}{\cdots} \times \mathbb{T}_{n,b}[x]$. The secret key that is used for encryption is $sk = (s_1, \dots, s_k)$ where $s_i \in \{0, 1\}\ \forall i \in \{1, \dots, k\}$. Encoding and decoding of messages is very similar to the encoding in BGV and BFV:

$$\begin{array}{rccc} \texttt{Encode}: & \mathbb{Z}_a^n & \longrightarrow & \mathcal{P} \\ & (m_0, \dots, m_{n-1}) & \longmapsto & \frac{m_0}{a} + \frac{m_1}{a}x + \cdots + \frac{m_{n-1}}{a}x^{n-1} \end{array}$$

As in BGV and BFV, by convention, if $\ell < n$ elements want to be encoded, then the last $n - \ell$ elements are set to 0. Decoding is direct from the encoding function.

For the encryption, we have the following function:

$$\begin{array}{rccc} \texttt{Encrypt}: & \mathbb{T}_{n,a}[x] & \longrightarrow & \mathbb{T}_{n,b}[x]^{k+1} \\ & m(x) & \longmapsto & (a_1(x), \dots, a_k(x), b(x)) \end{array}$$

where:

- The elements $(a_1(x), \ldots, a_k(x))$ are randomly chosen from $\mathbb{T}_{n,b}[x]$ every time a message $m(x)$ is to be encrypted.
- The polynomial $b(x)$ is of the form:

$$b(x) = \sum_{i=1}^{k} s_i \cdot a_i(x) + e(x) + m(x),$$

  where:

  - It is important to recall that the $s_i \in \{0,1\}$ and, as such, they establish whether $a_i(x)$ is to be added to $b(x)$ or not.
  - $e(x)$ is an error polynomial similar to the ones in BGV or BFV. In this case, its generation is threefold:
    1. Take $e_0(x) \leftarrow \chi$.
    2. Compute $e_1(x) = \lfloor b \cdot e_0(x) \rceil$, with $\lfloor \cdot \rceil$ being the rounding function. The product and rounding are done per each coefficient of the polynomial.
    3. Finally $e(x) = \frac{e_1(x)}{b}$. The division is done per each coefficient of the polynomial, as before.

The security of the above encryption is based on the GLWE assumption over the discretized torus [31]. This assumption states that an element $r(x) \in \mathbb{T}_{n,b}[x]$ such that $r = \sum_{i=1}^{k} s_i \cdot a_i(x) + e(x)$ is indistinguisable from a random torus element $r(x) \in \mathbb{T}_{n,b}[x]$.

Decryption, instead, is done in two steps:

1. Compute $m'(x) = b(x) - \sum_{i=1}^{k} s_i \cdot a_i(x)$ (inside $\mathbb{T}_{n,b}[x]$)
2. Then $m(x) = \frac{\lfloor a \cdot m'(x) \rceil \mod a}{a}$.

Bootstrapping in TFHE. Before introducing bootstrapping in TFHE, the homomorphic operations inside this scheme must be introduced. As seen above, in TFHE one can naturally compute additions and external products. However, the external product can be extended to an inner product using the GSW approach [25]. The addition is straightforward. Let $c^1 = (a_1^1(x), \ldots, a_k^1(x), b^1(x))$ and $c^2 = (a_1^2(x), \ldots, a_k^2(x), b^2(x))$ then:

$$c^1 + c^2 = (a_1^1(x) + a_1^2(x), \ldots, a_k^1(x) + a_2^2(x), b^1(x) + b^2(x)).$$

Observe that

$$\begin{aligned} b^1(x) + b^2(x) &= \textstyle\sum_{i=1}^{k} s_i \cdot a_i^1(x) + e^1(x) + m^1(x) + \sum_{i=1}^{k} s_i \cdot a_i^2(x) + e^2(x) + m^2(x) \\ &= \textstyle\sum_{i=1}^{k} s_i \cdot a_i^1(x) + \sum_{i=1}^{k} s_i \cdot a_i^2(x) + [e^1(x) + e^2(x)] + [m^1(x) + m^2(x)] \\ &= \textstyle\sum_{i=1}^{k} s_i \cdot [a_i^1(x) + a_i^2(x)] + [e^1(x) + e^2(x)] + [m^1(x) + m^2(x)] \end{aligned}$$

The other two operations are more challenging and involve Gadget matrixes and introducing the GSW encryption scheme [25], which is out of the scope of this document.

Assuming the above operations can be performed, the idea of the bootstrapping technique in TFHE is the same as in Gentry's one: evaluating the decryption function homomorphically, *i.e.*:

1. Compute $\texttt{Encrypt}_{sk_2}(m'(x)) = b(x) - \sum_{i=1}^{k} \texttt{Encrypt}_{sk_2}(s_i) \cdot a_i(x)$
2. Then $\texttt{Encrypt}_{sk_2}(m(x)) = \frac{\lfloor a \cdot \texttt{Encrypt}_{sk_2}(m'(x)) \rceil \mod a}{a}$.

The first step is easy being a combination of operations that are preserved by the homomorphisms in TFHE. The second step, instead, needs to apply torus polynomials' properties to be done. The whole process can be seen in [31].

Moreover, the TFHE scheme not only can perform bootstrapping, but an generalized version of it called *programmable bootstrapping* too. In programmable bootstrapping, one can evaluate another univariate function f over the ciphertext simultaneously to its recryption with a new key sk_2. Following the diagram of Figures 3.2 and 3.3, the programmable bootstrapping procedure is shown in Figure 3.7.

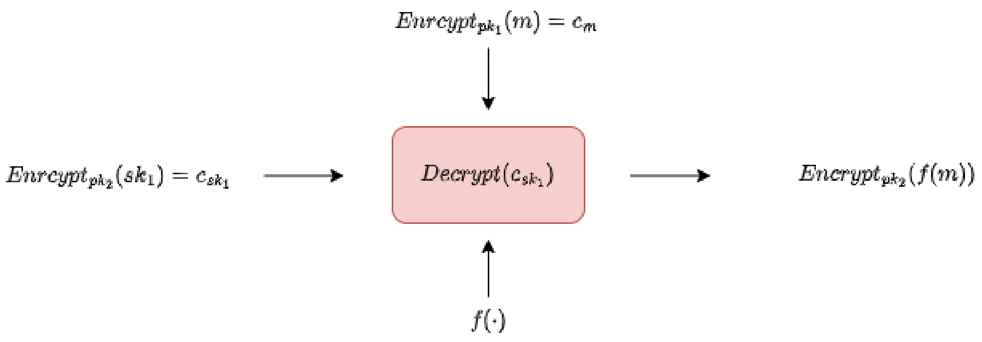


Figure 3.7: Programmable bootstrapping in TFHE.

Final notes on the TFHE scheme. As seen previously, TFHE is essentially different to the other FHE approaches. In fact, the first TFHE implementations were used to encrypt bits, using a simplified version of $\mathbb{T}_{n,b}[x]$ with $n = 1$. The operations allowed to homomorphically operate the basic logical gates. This, combined with programmable bootstrapping, allowed to evaluate homomorphically logical circuits of arbitrary length.

Nowadays, there are some implementations[2] trying to generalize this construction. These schemes are continuously gaining more attention and surely will be an interesting FHE choice in the future.

[2]Mainly the `tfhe-rs` implementation developed by Zama `https://github.com/zama-ai/tfhe-rs`.

Chapter 4

Allowing non-linear operations on BGV.

BGV and BFV turn out to be two interesting FHE schemes to perform privacy-preserving computations over the integers. Nevertheless, their main problem is well-known: only operations involving additions and products can be performed over their ciphertexts. Regarding the computational capabilities of the current programming languages, this is an issue to be considered.

From now on, this document will focus on the BGV scheme. The BFV scheme is usually considered an evolution of BGV [21], so the advances presented in this chapter can be applied to this scheme too. This chapter introduces the design of an algorithm to approximate non-linear operations for BGV ciphertexts.

4.1 Performing non-linear operations over the integers.

Being able to perform non-linear operations would greatly increase the utility of both BGV and BFV ciphers. The objective of this section is to give a possible design of an algorithm capable of approximating the following operations only using additions and products:

- Equality and inequality.
- Greater-than and less-than operations.
- Maximum and minimum.
- Integer division by known and encrypted divisor.

With the aim of extending BGV to have more computing capabilities, the FHE research community came up with several algorithms to compute comparisons and integer divisions [29, 40, 42, 43]. The algorithms to enable comparisons are usually based on Fermat's Little Theorem [40, 42], while integer division tends to use polynomial interpolation [40, 43].

4.1.1 Fermat's Little Theorem.

Theorem 1 (Fermat's Little Theorem) is a very interesting result because it turns an element in $\mathbb{Z}_p$ into a 0 or a 1 (*i.e.*, a boolean). Indeed, given an element $a \in \mathbb{Z}_p$, if $a \neq 0$, then by Theorem 1 we have that:

$$a^{p-1} \equiv 1 \mod p$$

and if $a = 0$ then $a^{p-1} = 0^{p-1} \equiv 0 \mod p$. With this result, an inequality test is straighforward to be built using only linear operations. Let $m_1, m_2 \in \mathbb{Z}_p$ be two messages:

1. Compute $d = m_1 - m_2$. If both messages are equal, then $d = 0$, and $d \neq 0$ otherwise.
2. Compute $r = d^{p-1} \mod p$. If $d = 0$, then $r = 0$, but if $d \neq 0$ then $r = 1$.

Expressing 0 as `False` and 1 as `True`, the inequality test for m_1 and m_2 is working. Observe that this test only involves a substraction (step 1) and several products (step 2). The equality test can be obtained by adding a third step to the previous ones:

3. Compute $r' = -r + 1$. If $r = 0$, $r' = -0 + 1 = 1$ and if $r = 1$ then $r' = -1 + 1 = 0$.

Again, this last step involves only linear operations. In conclusion, Fermat's Little Theorem allows to perform equality and inequality tests using only additions and products in a few steps. Since addition and product are homomorphic operations in BGV, this test can be performed homomorphically. The main challenge is the noise growth due to the amount of products of d^{p-1}, but applying the modular exponentiation algorithm can be significantly reduced, decreasing the noise growth too.

4.1.2 Polynomial interpolation.

Fermat's Little Theorem is a very interesting approach to perform equality/inequality tests, but the algorithm above cannot be extended to solve other non-linear operations, such as greater-than comparisons or similar operations. The reason for this is simple: Fermat's Little Theorem only makes a distinction between 0 and the other elements in $\mathbb{Z}_p$. For a greater-than comparison, once $d = m_1 - m_2$ is computed, one not only needs to know if the result is 0 or not, but the sign of d too.

In consequence, a solution based on Lagrange's polynomial interpolation formula is introduced in the following lines. Polynomial interpolation intends to approximate a function f using polynomials. Let $f(x)$ be the function to be approximated using Lagrange's interpolation and let $\{x_0, \cdots, x_n\}$ be $n+1$ points for whom their images $f(x_i)$ are known. Lagrange's interpolation polynomial is defined as follows:

$$L(x) = \sum_{i=0}^{n} \left(\prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \right) \cdot f(x_i). \tag{4.1}$$

Recall that in BGV the plaintext space is $\mathcal{P} = \frac{\mathbb{Z}_p[x]}{x^n+1}$. Hence, the above interpolation polynomial must be computed over $\mathbb{Z}_p$. In $\mathbb{Z}_p$ there is no division; the inverse of $\prod_{j=0,j\neq i}^{n}(x_i - x_j)$ must be computed. The key point for $L(x)$ to be an exact approximation of $f(x)$ is to build $L(x)$ with every point in $\mathbb{Z}_p$, since $\mathbb{Z}_p$ is a finite set.

Let us see the process to evaluate an arbitrary univariate function $f(x)$ homomorphically using polynomial interpolation. Let c be a BGV ciphertext encrypting a message $m \in \mathbb{Z}_p$ and $f(x)$ the function to be evaluated. Knowing p, Lagrange's interpolation polynomial can be computed in the clear using the formula in (4.1) using every element in $\mathbb{Z}_p$. The result is a polynomial of the form

$$L(x) = l_0 + l_1 x + \cdots + l_{p-1} x^{p-1}$$

approximating $f(x) \in \mathbb{Z}_p$. The idea is to evaluate $L(x)$ using c to obtain:

$$L(c) = l_0 + l_1 c + \cdots + l_{p-1} c^{p-1}.$$

For this evaluation, one needs to previously compute the set $\{c, c^2, \cdots, c^{p-1}\}$. Building this set

Algorithm 1: Homomorphic approximation of $f(x)$ using Lagrange's interpolation over BGV.

Input: A ciphertext $\mathbf{c} = \texttt{Encrypt}(m)$, the plaintext space $\mathbb{Z}_p$, the function to be approximated $f(x)$ and the BGV public key $\mathbf{pk}$.

Output: A ciphertext $\mathbf{c}' = \texttt{Encrypt}(f(m))$.

Initialization:

1. For every $x \in \mathbb{Z}_p$, compute $f(x)$ and store the sets:
   - $X = \mathbb{Z}_p = \{0, 1, \cdots, p-1\}$
   - $Y = \{f(0), f(1), \cdots, f(p-1)\}$
2. Compute $L(x)$ using Lagrange's interpolation formula over $\mathbb{Z}_p$ and the sets X and Y:
$$L(x) = \sum_{i=0}^{p-1} \left(\prod_{j=0, j\neq i}^{p-1} (x - x_j)(x_i - x_j)^{-1} \right) \cdot f(x_i).$$
   As a result, we obtain a polynomial:
$$L(x) = l_0 + l_1 x + \cdots + l_{p-1} x^{p-1}.$$
3. Encrypt every coefficient of $L(x)$. In other words, compute $\mathbf{s}_i = \texttt{Encrypt}(l_i)$, for $i = 0, \ldots, p-1$, using $\mathbf{pk}$. Store the encrypted values $\{\mathbf{s}_0, \ldots, \mathbf{s}_{p-1}\}$.
4. Compute homomorphically the set of powers $\{\mathbf{c}, \mathbf{c}^2, \ldots, \mathbf{c}^{p-1}\}$ using the Fast Modular Exponentiation algorithm over $\mathbf{c}$.
5. Compute homomorphically the value:
$$\mathbf{c}' = \mathbf{s}_0 + \mathbf{s}_1 \mathbf{c} + \cdots + \mathbf{s}_{p-1} \mathbf{c}^{p-1}.$$
6. Return $\mathbf{c}'$.

of elements and evaluating the polynomial only involves linear operations, so the evaluation of $L(x)$ can be done homomorphically in BGV. Besides, since the interpolation polynomial was computed over $\mathbb{Z}_p$, when decrypting $L(c)$ the Lagrange polynomial interpolation holds and $\texttt{Decrypt}(L(c)) = f(m)$.

The advantage of using this approach is that f can be any univariate function. Hence, any kind of nonlinear function can be approximated using this approach. As a summary, the algorithm for approximating any univariate function f is shown in Algorithm 1.

Chapter 5

State of the Art in Verifiable Decryption.

Homomorphic encryption is a very interesting technique in case one wants to operate on private data. However, let us note that, due to the very nature of FHE, this data exploitation can only be used by the data owner.

Indeed, it is the data owner who encrypts their private data and sends them to the data scientist. The latter will apply the necessary operations on them, but what they will finally obtain is an encrypted result, which can only be decrypted by the data owner. Therefore, the clear result can only be obtained by the data owner. This is described graphically in Figure 5.1, where Alice is the data owner and Bob is the data scientist.

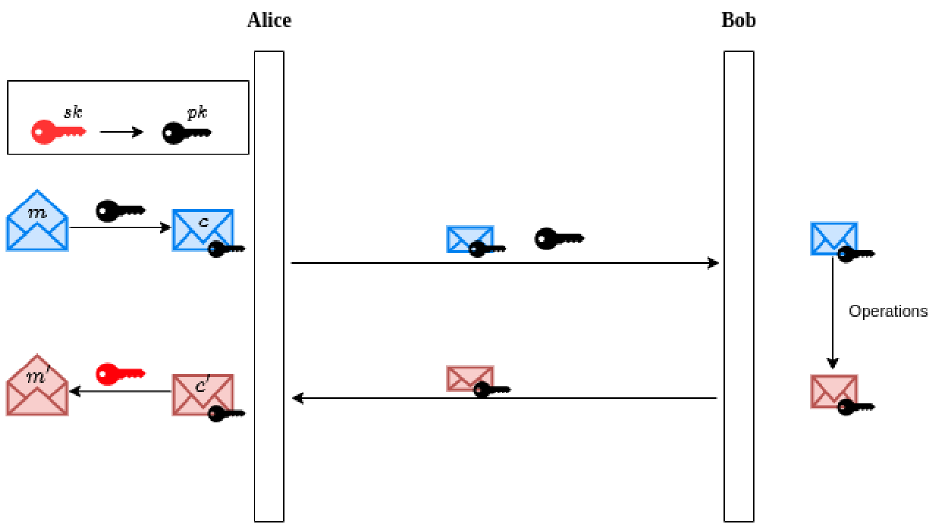


Figure 5.1: Data exploitation process in homomorphic encryption.

However, there are many occasions when the result of homomorphic operations is of interest to Bob, and not to Alice. As an example, suppose the Bob is training a pancreatic cancer precision model using medical data from different patients. In this case, the result is interesting for him, while Alice could have other incentives (e.g. economic ones).

In this situation, a new interaction should be added to the data exploitation process represented in Figure 5.1: the one in which Alice sends the decrypted result back to Bob. Figure 5.2 shows this new exploitation process.

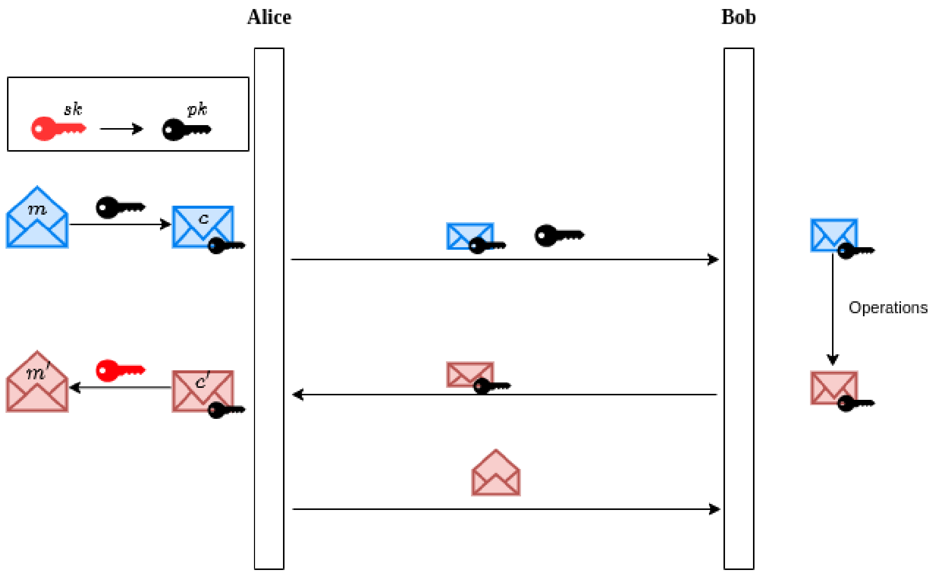


Figure 5.2: Data exploitation process with result coming back to the data scientist.

This paradigm opens a new issue to study: how can Bob ensure that the received result corresponds to m'? Alice could very well decrypt c' getting m', but send a different result m'' to Bob. Thus, Alice's economic incentive would be fulfilled, while Bob would receive an erroneous result that would not help him improve his model.

The purpose of this chapter is to find techniques that allow to verify that, indeed, the result received by Bob corresponds to the message obtained when decrypting c'. This property is known as *verifiable decryption* (VD).

The first proposed protocols tried to solve a similar problem called *verifiable encryption* (VE) [18, 27, 34]. In VE, Alice wants to prove to Bob that she knows the message m encapsulated inside c, without showing m. VE protocols also give integrity to some protocols, and can be very useful in certain situations, such as for setting up multi-party protocols [18]. Unfortunately, VE protocols cannot be extended to VD protocols, since the problem trying to be solved is not the same. They however use a technology that can be very useful for designing verifiable decryption protocols: lattice based zero-knowledge proofs.

Another approach to provide integrity to the FHE computations consists of extending the FHE schemes to a multi-party setting. The idea is not to use a pure multi-party computation protocol, but to create a distributed key that can then be used in a usual FHE setting. There are two types of multi-party FHE key generations, which lead to two different multi-party FHE protocols:

- Multi-key FHE: the final public and secret keys are generated using the personal public and secret keys of all the parties. For encryption and decryption, all the personal secret and public keys are needed.

- Threshold FHE: the final public and secret keys are generated as in the previous one, but for encryption and decryption it suffices to use a subset of the personal secret and public keys.

There is a lot of literature for both approaches, but multi-key FHE mainly focuses on bit-wise FHE [12, 14, 41, 45], while there are various threshold FHE designs for FHE over the integers [4, 8, 9, 30].

5.1 Lattice-based Zero-Knowledge Proofs.

A zero-knowledge proof is a solution to the problem where Alice wants to show Bob that she knows some secret without revealing it. Zero-knowledge proofs are usually based on elliptic curves or RSA [39] with two main purposes: privacy and scalability (mainly for blockchain networks).

Nevertheless, due to the importance given to lattice-based cryptography (considered quantum resistant), research on lattice-based zero-knowledge proofs has grown a lot in the last years. As a consequence, several works for constructing these arguments have been published so far [5, 10, 23, 32, 35]. Mainly, two kinds of proofs want to be constructed: proofs of linear relations [32, 35] and proofs of arithmetic circuits [5, 10, 23].

5.1.1 Proofs of linear relations.

A zero-knowledge proof of linear relations is a cryptographic protocol that allows one party, the prover, to convince another party, the verifier, that a specific mathematical relationship exists between a set of elements $\{x_1, \dots, x_n\}$, without revealing any information about the elements themselves.

Specifically, a zero-knowledge proof of linear relations can be used to prove that $\{x_1, \dots, x_n\}$ satisfy a linear relationship of the form:

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b$$

where $a_1, a_2, ..., a_n, b$ are known.

The zero-knowledge property ensures that the verifier can be convinced of the truth of the relation without learning any information about the individual values of $\{x_1, x_2, ..., x_n\}$. The prover can generate a proof that convinces the verifier without revealing the values themselves, thereby preserving privacy.

5.1.2 Proofs of arithmetic circuits.

A zero-knowledge proof of arithmetic circuits is a cryptographic protocol that allows the prover to demonstrate the verifier that they possess knowledge of inputs to an arithmetic circuit such that the circuit correctly evaluates to a specific output, without revealing any information about the inputs themselves.

Arithmetic circuits.

An arithmetic circuit is a computational model used to perform arithmetic operations on input values. It is composed of a collection of gates, each representing an arithmetic operation, and wires connecting the gates to transmit data. In an arithmetic circuit, the input values are typically considered as constants or variables, and the circuit performs computations by combining these inputs using basic arithmetic operations, such as addition and multiplication. The outputs of the circuit are the results of these computations.

The gates in an arithmetic circuit can be categorized into two main types:

- Addition Gates: An addition gate takes two input values and produces their sum. It is represented by a plus sign (+) or a similar symbol.
- Multiplication Gates: A multiplication gate takes two input values and produces their product. It is represented by a multiplication symbol ($\times$) or a similar symbol.

By combining these gates and arranging them in a specific manner, complex computations can be performed, including polynomial evaluations, matrix operations, and more.

Zero-knowledge proofs.

The goal of the zero-knowledge proof is to convince the verifier that the prover possesses the necessary inputs to make the circuit evaluate to a desired output value, while keeping the inputs confidential.

The zero-knowledge property ensures that the verifier gains confidence in the prover's claim without learning any additional information about the inputs. The prover constructs a proof that convinces the verifier of the correctness of the computation, without disclosing the actual input values used to generate the desired output.

Zero-knowledge proofs of arithmetic circuits have significant applications in various cryptographic protocols, such as secure multiparty computation [33], where multiple parties wish to compute a joint function over their private inputs without revealing those inputs. By using zero-knowledge proofs, the parties can interactively verify the correctness of their computations without disclosing sensitive information, ensuring privacy and security.

These proofs play a crucial role in the design of privacy-preserving technologies, enabling the verification of computations without revealing the underlying data or intermediate steps involved in the computation.

5.1.3 Commitment schemes.

Commitment schemes are a fundamental part on every protocol involving zero-knowledge proofs. A commitment scheme is a protocol that allows the prover to commit to a verifier on a secret value s without revealing it. The commitment scheme outputs a value $[[s]]$, usually called commitment of s, that can later be opened by the verifier to obtain s.

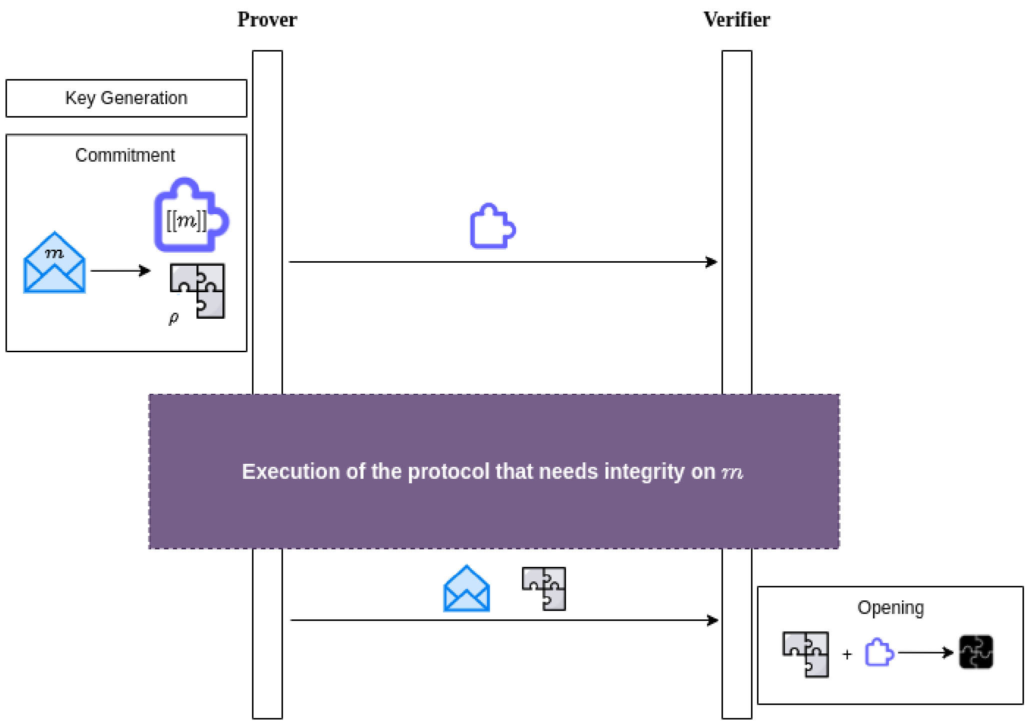


Figure 5.3: The usage of a commitment scheme to give integrity to larger protocols.

A commitment scheme is composed by three algorithms:

- *Key Generation:* Creates all the cryptographic material that is needed for the next algorithms.
- *Commitment:* On input a message m, creates a commitment $[[m]]$ and a proof ρ for the verifier to check the commitment in the future. This proof is called opening in the literature.
- *Opening:* On input m, $[[m]]$ and ρ, outputs 1 if the opening ρ applied to $[[m]]$ results in m. The output is 0 otherwise.

In conclusion, commitment schemes are used to give integrity to more complex protocols, as it is depicted in Figure 5.3.

As it can be deduced from Figure 5.3, at the end of the opening, the verifier can decide if the protocol was correctly executed by checking if m is the expected message.

5.2 Zero-Knowledge Proofs of Decryption in BGV.

During the past few years, several protocols for verifiable decryption using lattice-based zero-knowledge proofs have been proposed [13, 26, 36, 50]. All of them pursue the same functionality: a proof that Alice can send together with the decrypted message m', allowing Bob to verify that m' corresponds with the decryption of c' (see Figure 5.2).

The above protocols can be classified into two main categories: the ones proposed for the BGV cryptosystem [26, 50] and the protocols built for other FHE schemes [13, 36].

Starting with the latter, in [13] a zero-knowledge proof of correct decryption on Gentry's original FHE ciphertexts is proposed. For the proof to be built, they use an adaptation of the Schnorr zero-knowledge proof protocol introduced in [49]. The original proof in [49] is based on the discrete logarithm problem: Alice wants to convince Bob that she knows a secret value x such that $y = g^x$, where g is a generator of a cyclic group G, and both g and y are public. However, the verifiable decryption protocol in [49] is only able to prove the correct decryption of ciphertexts encrypting $m = 0$. This is a limitation that seriously affects the usability of the protocol.

The work of Lyubashevsky et al. [36] proposes a verifiable decryption protocol for the Kyber scheme. This scheme is not homomorphic, but it is one of the NIST finalists for the secure post-quantum cryptosystems. The Kyber scheme is based on lattices, but the plaintext space is smaller than the standard in BGV [2]. As a consequence, it may happen that some of the messages encrypted using BGV cannot be encrypted in the Kyber cyptosystem of [36]. The proposed protocol is very secure and complex; in fact, it involves using Gaussian sampling, partially spliting rings, rejection sampling and automorphisms. However, this can be seen as a limitation: the correct and secure implementation of the protocol results to be very complicated in practice.

5.2.1 Zero-Knowledge Proofs of Decryption for BGV.

This section is intended to introduce the verifiable decryption protocol proposed in [50]. Despite the protocol in [26] gives a complete verifiable decryption protocol for the BGV scheme, when comparing both protocols, the protocol in [50] results to be more direct and efficient.

Namely, there are two crucial differences between both protocols: the zero-knowledge proofs' sizes and the efficiency of the implementation. Firstly, the zero-knowledge proofs built in [26] are dependent on a soundness[1] parameter λ. As a result, the proofs built in [26] are 3.2 times larger than the ones in [50] for an interactive protocol with $\lambda = 10$. Analogously, for a non interactive protocol with $\lambda = 128$, the proof size in [26] is 41 times larger.

Secondly, the efficiency of the VD protocol proposed in [26] also depends on λ. More precisely, for $\lambda \leq 19$ both protocols have similar timings, but for greater soundness values the protocol in [50] is faster. For $\lambda = 128$ this difference increases up to 7 times faster.

5.2.2 The VD protocol in [50].

The verifiable decryption protocol proposed in [50] is the result of several algorithms introduced in other works:

- The commitment scheme proposed in [6].
- The zero-knowledge proof of linear relations from [3].
- The zero-knowledge proof of arithmetic circuits built in [5].

Consequently, for the correct description of the VD protocol in [50], firstly the above processes are introduced in the proposed order, and finally the VD protocol is described.

[1]Soundness is a crucial property in zero-knowledge proofs because it ensures that a dishonest prover cannot deceive the verifier into accepting an invalid or false statement.

Previous notation.

For the protocols to be clear, some notation needs to be established. Let N be the degree of the cyclotomic polynomial defining $\mathcal{R}_q$. Let χ and $\mathcal{N}$ be two Discrete Gaussian Distributions, the former over $\mathcal{R}_q$ and the latter over $\mathbb{Z}$. Both distributions are centered to $\mu = 0$, and with standard deviations σ_1, σ_2, respectively. Moreover, χ has an upper limit β and, as such, is analogous to the Discrete Gaussian Distribution defined in Section 2.2[2]. We define the sets

$$\mathcal{L} = \{x \in \mathcal{R}_q \mid ||x||_\infty = 1,\ ||x||_1 = \kappa\},\ \overline{\mathcal{L}} = \{x - x' \mid x, x' \in \mathcal{L};\ x \neq x'\}$$

$$\text{and } \mathcal{B} = \{0, 1\}.$$

Note that $||x||_1$ and $||x||_\infty$ are the first norm and infinity norm from Definitions 6 and 8, respectively. The value κ is fixed beforehand. Given the ring $\mathcal{R}_q$ for a prime q, we extend $\mathcal{R}_q$ using the common cartesian product, and define:

$$\mathcal{R}_q^k = \mathcal{R}_q \times \overset{k}{\cdots} \times \mathcal{R}_q.$$

The elements in $\mathcal{R}_q^k$ can be seen as (column) vectors and will be represented with bold lowercase letters. For instance, an element $\mathbf{a} \in \mathcal{R}_q^k$ is represented as:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix},$$

where every $a_i \in \mathcal{R}_q$ for $i = 1, \ldots, k$. Similarly, $\mathcal{R}_q^{k\times l}$ contains matrixes, which will be denoted with bold, uppercase letters. Again, if for example $\mathbf{A} \in \mathcal{R}_q^{k\times l}$ then:

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_l] = \begin{bmatrix} a_{11} & \cdots & a_{1l} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kl} \end{bmatrix}.$$

In this last case, the $\mathbf{a}_i$ are elements in $\mathcal{R}_q^k$ that, when extended as vector columns, result in the matrix $\{a_{ij}\}_{1\le i\le k, 1\le j\le l}$ where every $a_{ij} \in \mathcal{R}_q$. In situations where such vectors or matrices are randomly generated following χ or $\mathcal{N}$, the notation is $\mathbf{a} \leftarrow \chi^k$ or $\mathbf{A} \leftarrow \mathcal{N}^{k\times l}$.

The operations on these new sets are understood as in the usual vector and matrix fashion. Let $a \in \mathcal{R}_q$, $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q^k$ and $\mathbf{A}, \mathbf{B} \in \mathcal{R}_q^{n\times k}$:

$$a \cdot \mathbf{b} = \begin{bmatrix} a \cdot b_1 \\ \vdots \\ a \cdot b_k \end{bmatrix},\ a \cdot \mathbf{B} = \begin{bmatrix} a \cdot b_{11} & \cdots & a \cdot b_{1k} \\ \vdots & \ddots & \vdots \\ a \cdot b_{n1} & \cdots & a \cdot b_{nk} \end{bmatrix},\ \langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\top \cdot \mathbf{b} = \sum_{i=1}^{k} a_i b_i$$

and the operations $\mathbf{A} \cdot \mathbf{b}$ and $\mathbf{A} \cdot \mathbf{B}$ follow the normal matrix product procedure.

[2]It is important to note that the parameters defining both distributions are different.

The commitment scheme in [6].

The commitment scheme constructed by Baum et al. in [6] allows to commit to messages in $\mathcal{R}_q$ and can be used in more general protocols as shown in Figure 5.3. Recall from Section 5.1.3 that commitment schemes usually are composed by three algorithms: key generation, commitment and opening.

Algorithm 2: Key Generation.

Input: None.
Output: A public key $\texttt{pk}$ to commit to messages in $\mathcal{R}_q$.
Initialization:

1. Generate a random matrix $\mathbf{A}_1' \in \mathcal{R}_q^{n\times(k-n)}$ and a random vector $\mathbf{a}_2' \in \mathcal{R}_q^{(k-n-1)}$.
2. Build
$$\mathbf{A}_1 = \begin{bmatrix} \mathbf{I}_n & \mathbf{A}_1' \end{bmatrix},$$
where $\mathbf{I}_n$ is the identity matrix of rank n.
3. Build
$$\mathbf{a}_2 = \begin{bmatrix} \mathbf{0}^n & 1 & (\mathbf{a}_2')^\top \end{bmatrix},$$
where $\mathbf{0}^n$ is a vector of n zeroes and the term $(\mathbf{a}_2')^\top$ refers to the transpose of the column vector $\mathbf{a}_2'$.
4. Output
$$\texttt{pk} = \mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{a}_2 \end{bmatrix} \in \mathcal{R}_q^{(n+1)\times k}.$$

Algorithm 3 describes the commitment algorithm.

Algorithm 3: Commitment.

Input: The message $m \in \mathcal{R}_q$.
Output: The commitment $[[m]]$ and opening ρ.
Initialization:

1. Generate a random vector $\mathbf{r} \leftarrow \chi^k$.
2. Compute
$$[[m]] = \mathbf{A}\cdot\mathbf{r} + \begin{bmatrix} (\mathbf{0}^n)^\top \\ m \end{bmatrix} \in \mathcal{R}_q^{n+1}.$$
3. Build opening as the triple
$$\rho = (m, \mathbf{r}, 1).$$

Lastly, the opening algorithm is shown in Algorithm 4. Recall that the opening algorithm differs from the previous two algorithms in two ways. Firstly, this algorithm is executed in the last step of the more general protocol that needs commitments for integrity. This means that its execution is not directly after the other have been run.

The second difference comes from the fact that this algorithm is not executed by the prover. It is the verifiers that execute this algorithm, since they need to check the correctness of the received commitment $[[m]]$. For this verification, before executing the algorithm, the prover must send the opening ρ generated in the commitment algorithm. Recall from Algorithm 2 that $\rho = (m, \mathbf{r}, 1)$.

Algorithm 4: Opening.

```
Input: An opening $\rho$, a commitment $[[m]]$ and an element $\ell \in \overline{\mathcal{L}}$.
Output: 1 if $m \in \rho$ corresponds to the committed value $[[m]]$, 0 otherwise.
Initialization:
    1. Check that every component $r_i$ of $\mathbf{r}$ verifies
           $||r_i||_2 \leq 4\sigma_1\sqrt{N}.$
    2. Verify that the following equation holds:
           $\ell \cdot [[m]] = \mathbf{A} \cdot \mathbf{r} + \ell \cdot \begin{bmatrix} (\mathbf{0}^n)^\top \\ m \end{bmatrix}$
    3. If the previous checkings were succesful return 1, 0 otherwise.
```

The zero-knowledge proof of linear relations in [3].

Given two commited values $[[x]]$ and $[[x']]$ and two public values $\delta, \gamma \in \mathcal{R}_q$, Aranha et al. presented in [3] a protocol to prove in zero-knowledge that

$$x' = \delta \cdot x + \gamma.$$

The proposed zero-knowledge proof makes use of Algorithms 2 and 3 for the generation of $[[x]]$ and $[[x']]$. The complete scheme is diagrammed in Figure 5.4. For this protocol, $[[x]]$ and $[[x']]$ are expressed as

$$[[x]] = \begin{bmatrix} \mathbf{x}_1 \\ x_2 \end{bmatrix}, [[x']] = \begin{bmatrix} \mathbf{x}'_1 \\ x'_2 \end{bmatrix}.$$

Also, the openings ρ and ρ' obtained from Algorithm 2 are defined as:

$$\rho = (x, \mathbf{r}_x, 1) \text{ and } \rho = (x, \mathbf{r}_{x'}, 1).$$

The zero-knowledge proof of arithmetic circuits in [5].

Let $\mathbf{M} \in \mathcal{R}_q^{n\times k}$ be a public matrix, let $\mathbf{s} \in \mathcal{R}_q^k$ be a secret vector known by the prover such that every component s_i satisfies $||s_i||_2 \leq \epsilon$ for some ϵ and let

$$\mathbf{M} \cdot \mathbf{s} = \mathbf{t}.$$

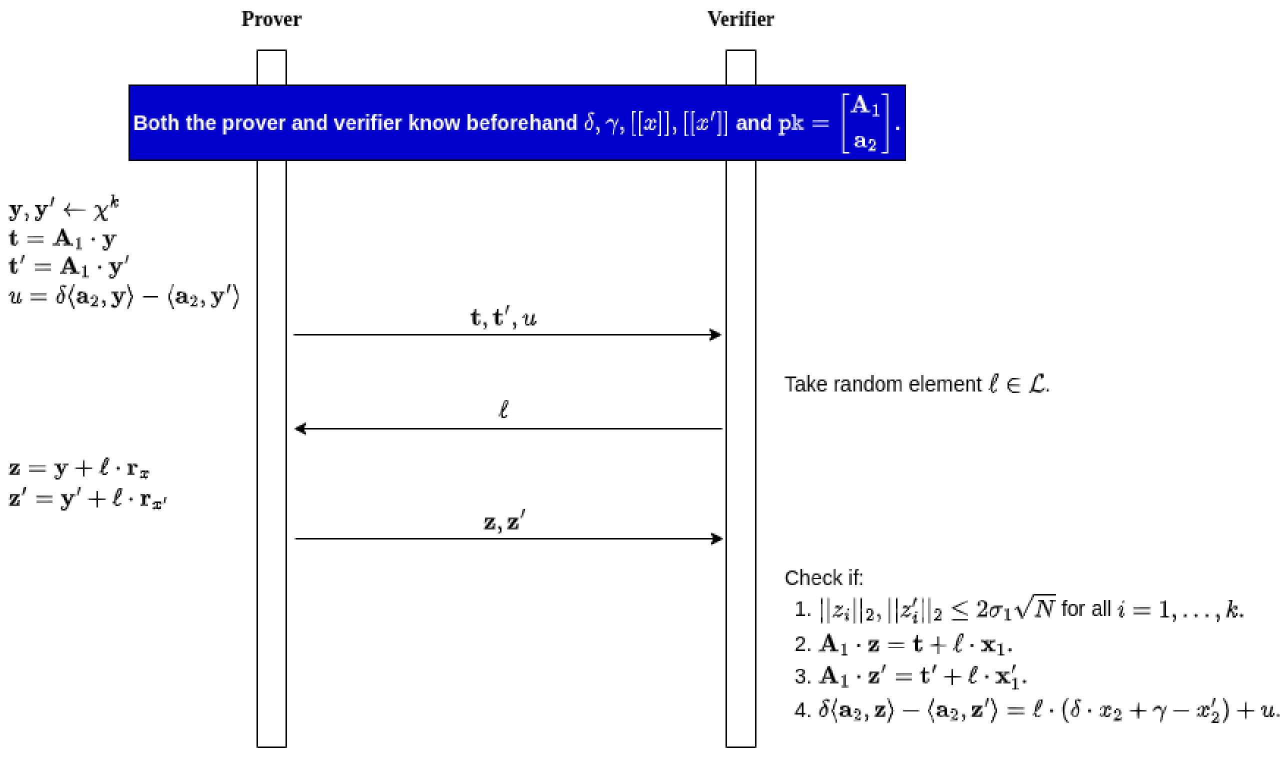

Prover
Verifier
Both the prover and verifier know beforehand $\delta, \gamma, [[x]], [[x']]$ and $\mathrm{pk} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{a}_2 \end{bmatrix}$.
$\mathbf{y}, \mathbf{y}' \leftarrow \chi^k$
$\mathbf{t} = \mathbf{A}_1 \cdot \mathbf{y}$
$\mathbf{t}' = \mathbf{A}_1 \cdot \mathbf{y}'$
$u = \delta\langle \mathbf{a}_2, \mathbf{y}\rangle - \langle \mathbf{a}_2, \mathbf{y}'\rangle$
$\mathbf{t}, \mathbf{t}', u$
Take random element $\ell \in \mathcal{L}$.
ℓ
$\mathbf{z} = \mathbf{y} + \ell \cdot \mathbf{r}_x$
$\mathbf{z}' = \mathbf{y}' + \ell \cdot \mathbf{r}_{x'}$
$\mathbf{z}, \mathbf{z}'$
Check if:
1. $||z_i||_2, ||z_i'||_2 \leq 2\sigma_1\sqrt{N}$ for all $i = 1, \dots, k$.
2. $\mathbf{A}_1 \cdot \mathbf{z} = \mathbf{t} + \ell \cdot \mathbf{x}_1$.
3. $\mathbf{A}_1 \cdot \mathbf{z}' = \mathbf{t}' + \ell \cdot \mathbf{x}_1'$.
4. $\delta\langle \mathbf{a}_2, \mathbf{z}\rangle - \langle \mathbf{a}_2, \mathbf{z}'\rangle = \ell \cdot (\delta \cdot x_2 + \gamma - x_2') + u$.

Figure 5.4: Protocol to prove the relation $x' = \delta \cdot x + \gamma$ in zero-knowledge given $\delta, \gamma, [[x]], [[x']]$ and `pk`.

The protocol constructed by Baum et al. in [5] allows for the prover to prove the verifier in zero-knowledge that, indeed, the above evaluation holds and all the s_i are bounded.

The protocol is depicted in Figure 5.5. Observe in Figure 5.5 that the protocol makes use of the Discrete Gaussian Distribution over the integers $\mathcal{N}$. Therefore, since $\mathbf{Y} \leftarrow \mathcal{N}^{k\times n}$, we have that $\mathbf{Y} \in \mathbb{Z}^{k\times n}$. Also note that $\mathbf{b}$ is a vector composed of zeroes and ones since $\mathcal{B} = \{0, 1\}$. In [50], ϵ is chosen to be $\epsilon = 2\sqrt{2N}\sigma_2$.

Non-interactiveness via the Fiat-Shamir transform.

The protocols described in Figures 5.4 and 5.5 give zero-knowledge proofs for the relations between commited or secret values. They therefore are key in the VD protocol introduced in Figure 5.7.

However, both zero-knowledge protocols above need an interaction with the verifier before building the final proof. In fact, in Figure 5.4, the verifier is asked to generate a random element $\ell \in \mathcal{L}$, whereas in Figure 5.5 is asked to generate $\mathbf{b} \in \mathcal{B}^n$. This is not desirable, as it would require interactions between prover and verifier during the execution of each protocol.

The work by Amos Fiat and Adi Shamir in [22] introduced a solution to turn interactive protocols into non-interactive ones. This solution is usually called the *Fiat-Shamir transform*, and is exhaustively used in cryptography and, especially, in zero-knowledge proof protocols.

An interactive protocol can be split into four phases: the commitment phase, the challenge

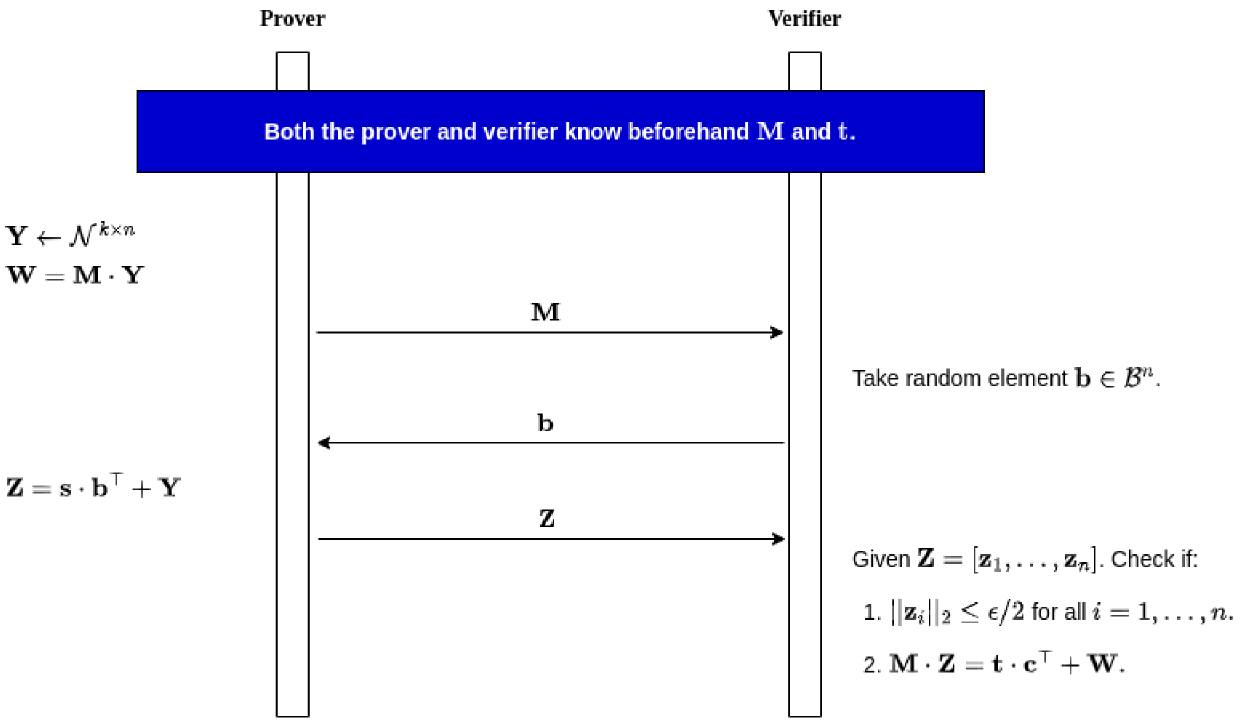

Prover
Verifier
Both the prover and verifier know beforehand $\mathbf{M}$ and $\mathbf{t}$.
$\mathbf{Y} \leftarrow \mathcal{N}^{k\times n}$
$\mathbf{W} = \mathbf{M} \cdot \mathbf{Y}$
$\mathbf{M}$
Take random element $\mathbf{b} \in \mathcal{B}^n$.
$\mathbf{b}$
$\mathbf{Z} = \mathbf{s} \cdot \mathbf{b}^\top + \mathbf{Y}$
$\mathbf{Z}$
Given $\mathbf{Z} = [\mathbf{z}_1, \dots, \mathbf{z}_n]$. Check if:
1. $||\mathbf{z}_i||_2 \leq \epsilon/2$ for all $i = 1, \dots, n$.
2. $\mathbf{M} \cdot \mathbf{Z} = \mathbf{t} \cdot \mathbf{c}^\top + \mathbf{W}$.

Figure 5.5: Protocol to prove the relation $\mathbf{M} \cdot \mathbf{s} = \mathbf{t}$ with bounded $\mathbf{s}$ in zero-knowledge.

phase, the proof generation phase and the verification phase. The commitment and proof generation phases are run by the prover, while the challenge and verification phases are executed by the verifier (see Figure 5.6, left). In order to avoid interactions, the objective is to remove the need of the verifier running the challenge phase. In this phase, the verifier generates a random suitable element that the prover uses to build the final proof in the proof generation phase. This way, since the verifier knows about the randomness of the generated element, there is no doubt during verification that the proof is correct.

In consequence, we need a technique able to produce enough randomness on the prover's side to construct a correct proof. Simultaneously, this technique must allow the verifier to check what is this random value that has been created. There is a unique cryptographical element that can do this: *hash functions*.

Following the Fiat-Shamir transform, one can avoid the challenge phase by hashing the elements of the commitment phase, which are known by both parties. This way, the prover generates the necessary randomness to generate a correct zero-knowledge proof, and the verifier can generate the same randomness using the same hash function over the commitment phase elements. In consequence, the challenge phase can be avoided and the interactive protocol be transformed to a non-interactive one, as shown at the right of Figure 5.6.

In conclusion, the Fiat-Shamir transform can be used to build non-interactive proofs from the interactive protocols in [3] and [5]. In fact, when applying the Fiat-Shamir transform to them, the results of the challenge phases of each of the protocols will be 'the output of a hash function applied to the full transcript' [50]. From now on, Π_{ZKPLR} is the zero-knowledge proof obtained after applying the Fiat-Shamir transform to the protocol described in Figure

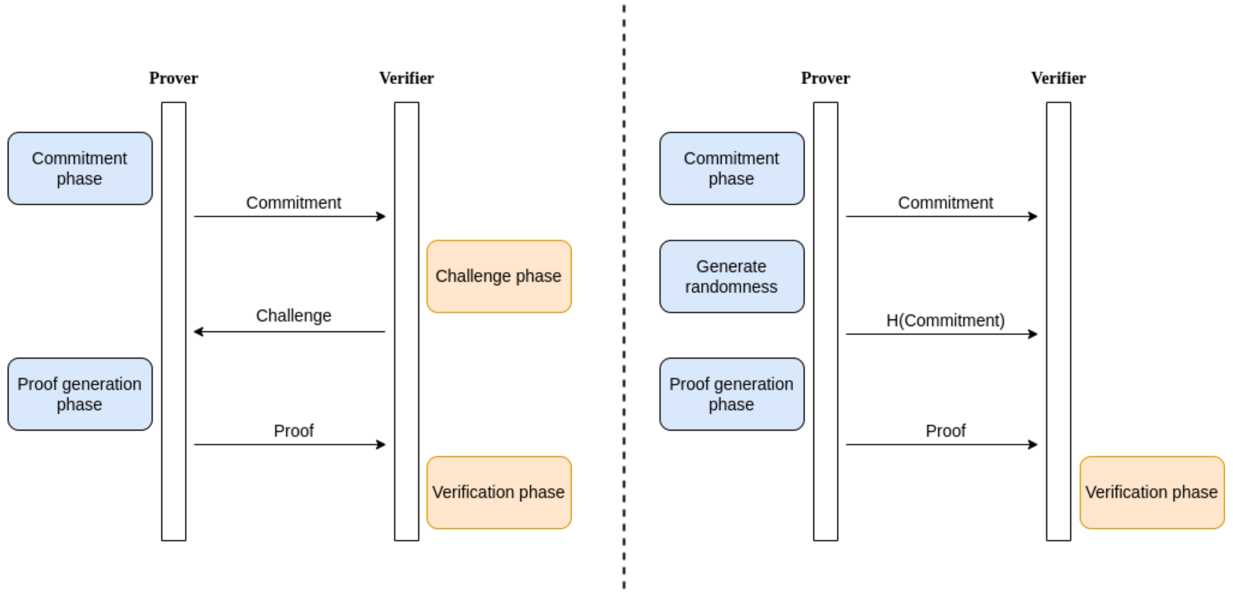


Figure 5.6: Application of the Fiat-Shamir transform to convert an interactive proof system to a non-interactive one.

5.4, whereas the zero-knowledge proof resulting from the application of the transform to the protocol in Figure 5.5 will be denoted as Π_{ZKPAC}.

The Verifiable Decryption protocol in [50].

Tjerand Silde gathers all the previous pieces to build a very direct and efficient VD protocol in [50]. He proposes a protocol that can generate zero-knowledge proofs of correct decryption for an arbitrary number of ciphertexts $\mathbf{c}_1, \ldots, \mathbf{c}_n$. However, the simplified version of the protocol for the decryption of only one ciphertext $\mathbf{c}$ is shown here. Figure 5.7 contains the simplified VD protocol for a BGV ciphertext proposed in [50].

It is important to note that, in Figure 5.7, when writing $\eta_1 \leftarrow \Pi_{ZKPLR}$, η_1 is assumed to be the output of the execution of the protocol Π_{ZKPLR}, *i.e.*, η is the zero-knowledge proof that proves linear relations between committed elements.

This protocol gives a full proof of decryption since the verifier can check at the end that:

- The noise $\aleph$ committed by the prover satisfies to be the noise obtained during the decryption because it satisfies that:

$$p\aleph \equiv c_1 + c_2 \cdot sk - m.$$

- This noise is bounded by $||\aleph||_2 \leq \epsilon$. For it to be correctly bounded, ϵ must be $\epsilon < \frac{q}{2p}$. This way, from Definitions 6 and 8, we have that:

$$||\aleph||_\infty \leq ||\aleph||_2 < \frac{q}{2p}.$$

In [50], ϵ is fixed to $\epsilon = 2\sqrt{2N}\sigma_2$.

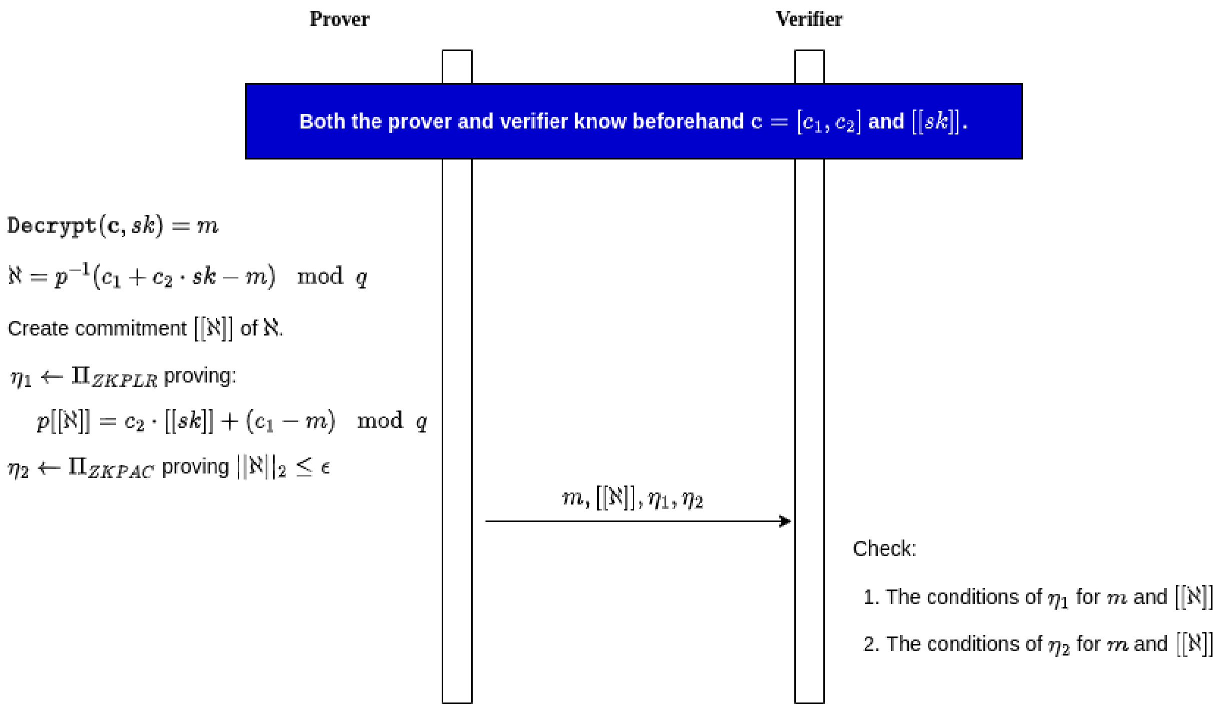

Prover
Verifier
Both the prover and verifier know beforehand $\mathbf{c} = [c_1, c_2]$ and $[[sk]]$.
$\mathtt{Decrypt}(\mathbf{c}, sk) = m$
$\aleph = p^{-1}(c_1 + c_2 \cdot sk - m) \mod q$
Create commitment $[[\aleph]]$ of $\aleph$.
$\eta_1 \leftarrow \Pi_{ZKPLR}$ proving:
$p[[\aleph]] = c_2 \cdot [[sk]] + (c_1 - m) \mod q$
$\eta_2 \leftarrow \Pi_{ZKPAC}$ proving $||\aleph||_2 \leq \epsilon$
$m, [[\aleph]], \eta_1, \eta_2$
Check:
1. The conditions of η_1 for m and $[[\aleph]]$
2. The conditions of η_2 for m and $[[\aleph]]$

Figure 5.7: Verifiable decryption protocol for the BGV scheme.

5.3 Threshold Fully Homomorphic Encryption.

The aim of this section is to build a VD protocol adapting the threshold FHE approach introduced in [4] to a two-party setting. The reference threshold FHE protocol is the one in [4], since its solution is the one implemented in OpenFHE [1]. The latter is the library used for the implementation. See Chapter 6 for the argumentation of choosing this library.

Thus, this section is organized as follows. In Section 5.3.1 the polynomial representations of ciphertexts and the use of *evaluation keys* to compute the products of ciphertexts are introduced. Section 5.3.2 describes the threshold FHE protocol proposed in [4]. Finally, in Section 5.3.3 it is shown how to use the threshold FHE protocol to build a VD protocol and the security of the solution is described.

5.3.1 Introducing the Evaluation Key.

Let $\mathbf{c} \in \mathcal{C}$ be a BGV ciphertext. In Section 3.2, $\mathbf{c}$ was expressed as $\mathbf{c} = (c_1(x), c_2(x))$. It is common, however, to represent ciphertexts with their *symbolic polynomials*. Indeed, for a ciphertext $\mathbf{c}$ as above, its symbolic polynomial is:

$$\phi_{\mathbf{c}}(X(x)) = c_1(x) + c_2(x) \cdot X(x)$$

With this representation, observe that $\phi_{\mathbf{c}}(sk(x))$ leads to the first step of the BGV decryption:

$$\phi_{\mathbf{c}}(sk(x)) = c_1(x) + c_2(x)sk(x) = m(x) + p\aleph(x) + q\aleph'(x).$$

Homomorphic operations in symbolic polynomial representation. With this new representation, the homomorphic operations can be redefined. Let $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$ be two ciphertexts encapsulating $m(x)$ and $m'(x)$, respectively. For the addition:

$$\begin{aligned}\phi_{\mathbf{c}}(sk(x)) + \phi_{\mathbf{c}'}(sk(x)) &= [m(x) + p\aleph_1(x) + q\aleph_2(x)] + [m'(x) + p\aleph_1'(x) + q\aleph_2'(x)] \\ &= m(x) + m'(x) + p[\aleph_1(x) + \aleph_1'(x)] + q[\aleph_2(x) + \aleph_2'(x)] \\ &= \phi_{\mathbf{c}+\mathbf{c}'}(sk(x))\end{aligned}$$

Equivalently, for the product the following holds:

$$\begin{aligned}\phi_{\mathbf{c}}(sk(x)) \cdot \phi_{\mathbf{c}'}(sk(x)) &= [m(x) + p\aleph_1(x) + q\aleph_2(x)] \cdot [m'(x) + p\aleph_1'(x) + q\aleph_2'(x)] \\ &= \begin{array}{l} m(x) \cdot m'(x) + p[m(x)\aleph_1'(x) + m'(x)\aleph_1(x) + p\aleph_1(x)\aleph_1'(x)] + \\ +q[m(x)\aleph_2'(x) + p\aleph_1(x)\aleph_2'(x) + m'(x)\aleph_2(x) + p\aleph_2(x)\aleph_1'(x) + q\aleph_2(x)\aleph_2'(x)] \end{array} \\ &= \phi_{\mathbf{c}\cdot\mathbf{c}'}(sk(x))\end{aligned}$$

In consequence, one can compute the needed operations using symbolic polynomials. With this representation and abusing notation (writing polynomials as coefficients), the operations can be computed as follows:

- Addition:
  $$\phi_{\mathbf{c}+\mathbf{c}'}(X) = [c_1 + c_2X] + [c_1' + c_2'X] = c_1 + c_1' + (c_2 + c_2')X.$$
  The degree of $\phi_{\mathbf{c}+\mathbf{c}'}(X)$ is the same as the symbolic polynomials of the ciphertexts, so the addition can be computed as follows: $\mathbf{c} + \mathbf{c}' = (c_1 + c_1', c_2 + c_2')$

- Product:
  $$\phi_{\mathbf{c}\cdot\mathbf{c}'}(X) = [c_1 + c_2X] \cdot [c_1' + c_2'X] = c_1c_1' + (c_1c_2' + c_2c_1')X + c_2c_2'X^2.$$
  The polynomial can be rewritten as $\phi_{\mathbf{c}\cdot\mathbf{c}'}(X) = C_1 + C_2X + C_3X^2$ where:
  - $C_1 = c_1c_1'$.
  - $C_2 = c_1c_2' + c_2c_1'$.
  - $C_3 = c_2c_2'$.

  Note that the degree of the resulting polynomial is 2. This is a problem, since it must be expressed as a polynomial of degree 1 (equivalently, as an element of the form $\mathbf{c}^* = (C_1^*, C_2^*)$).

The need of an evaluation key. Let $\phi_{\mathbf{c}\cdot\mathbf{c}'}(X) = C_1 + C_2X + C_3X^2$ be the result of a product between two ciphertexts $\mathbf{c}$ and $\mathbf{c}'$. We want to find $\phi_{\mathbf{c}^*}(X) = C_1^* + C_2^*X$ such that:

$$C_1 + C_2sk + C_3sk^2 \equiv C_1^* + C_2^*sk + \aleph \mod q.$$

In consequence, we need an encapsulation of $sk^2(x)$ for us to be able to decrease the degree of $\phi_{\mathbf{c}\cdot\mathbf{c}'}(X)$. The encapsulation of $sk^2(x)$ is given inside what is called the *evaluation key*, which is an additional public key 'masking' $sk^2(x)$. The evaluation key $\mathbf{ek} = (ek_1(x), ek_2(x))$ is computed analogously to the public key:

- $ek_1(x) = -(\alpha(x)sk(x) + pe(x)) + sk^2(x) \mod q$
- $ek_2(x) = \alpha(x)$

With **ek** one can now compute the following (abusing notation again):

- $C_1^* = C_1 + ek_1 C_3 \mod q$
- $C_2^* = C_2 + ek_2 C_3 \mod q$

Observe now that:

$$\begin{array}{rcl} \phi_{\mathbf{c}^*}(sk) & = & C_1^* + C_2^* sk \\ & = & [C_1 + ek_1 C_3] + [C_2 + ek_2 C_3] sk \\ & = & C_1 + [-(\alpha \cdot sk + p \cdot e) + sk^2] C_3 + [C_2 + \alpha C_3] sk \\ & = & C_1 - \alpha C_3 sk - pC_3 e + C_3 sk^2 + C_2 sk + \alpha C_3 sk \\ & \equiv & C_1 + C_2 sk + C_3 sk^2 - pC_3 e \mod q. \end{array}$$

Thus, the evaluation key makes it possible to reduce the degree of the symbolic polynomials resulting from products of ciphertexts. This way, the product of two ciphertexts $\mathbf{c} = (c_1, c_2)$ and $\mathbf{c}' = (c_1', c_2')$ can be expressed as follows:

$$\mathbf{c} \cdot \mathbf{c}' = (c_1 c_1' + ek_1 c_2 c_2', c_1 c_2' + c_2 c_1' + ek_2 c_2 c_2')$$

In conclusion, in practice, the homomorphic operations in BGV are computed using the symbolic polynomial representations. For the product, though, an evaluation key is needed to preserve the degree of the polynomial. This evaluation key can be created at the same time as the public key and is public too.

5.3.2 The threshold FHE protocol by Asharov et al.

This section describes the threshold FHE protocol presented by Asharov et al. in [4]. The OpenFHE community implemented this protocol in their GitHub library and an example showing how to use it for BGV, BFV and CKKS can be seen in [7]https://github.com/openfheorg/openfhe-development/blob/main/src/pke/examples/threshold-fhe.cpp.

The protocol in [4], though, is described only for BGV, so the following lines are intended to describe a simplified version of that protocol. As it is known from Section 3.2, the classical BGV scheme consists of the following phases:

- Key generation
- Encryption
- Homomorphic operations (additions and products)
- Decryption

The authors observed in [4] that BGV is homomorphic with respect to the keys. In other words, if one adds several secret and public key pairs, the resulting pair is a valid key pair:

$$(sk^*, \mathbf{pk}^*) = \sum_i (sk_i, \mathbf{pk}_i)$$

The idea then is to adapt only the key generation and decryption phases to a multi-party setting, while the encryptions and homomorphic evaluations are the classical ones using $(sk^*, \mathbf{pk}^*)$. This way, only one party (or even a trusted third party) must compute the homomorphic operations, which are usually the heavy ones.

Threshold parameters. The classical BGV key generation needs some parameters to be fixed beforehand:

$$(\chi_{\mu,\sigma,\beta}, p, q, n).$$

Recall that χ is a Discrete Gaussian Distribution with mean μ, standard-deviation σ and bounded by β. From now on, this distribution is denoted as χ_β. The integer n is the degree of the cyclotomic polynomial used for both the plaintext and ciphertext spaces. For the plaintext space, p defines the limits for the coefficients of the polynomials, and similarly does q for the ciphertext space.

The threshold key generation needs to set up several parameters more:

- $\beta_{\texttt{Enc}}, \beta_{\texttt{Dec}}$ and $\beta_{\texttt{Eval}}$: Define bounds for the Discrete Uniform Distributions $\chi_\beta^{\texttt{Enc}}$, $\chi_\beta^{\texttt{Dec}}$ and $\chi_\beta^{\texttt{Eval}}$, respectively. These distributions will be used to add some so-called *smudging noise* in key parts of the threshold BGV scheme. This is necessary from the fact that, when decrypting, there is the need to add some extra noise for the decrypted parts to reveal no more than the plaintexts.

- $a(x)$: This polynomial will be used during the generation of the public key $\mathbf{pk}_k$ for every party P_k. Recall from the BGV key generation that $pk_2(x) = a(x)$.

- $\alpha(x)$: The polynomial used for the generation of $\mathbf{ek}_k$ for all P_k must be fixed too. Recall from Section 5.3.1 that $ek_2(x) = \alpha(x)$.

- N: is the number of parties executing the protocol.

Threshold key generation.

On input the above public parameters, during the threshold key generation the aim is to output the following values:

- A share of the threshold secret key sk^*.

- The threshold public key $\mathbf{pk}^*$.

- The threshold evaluation key $\mathbf{ek}^*$.

```
Algorithm 5: Threshold public key generation.
  Input: A secret key $sk_k(x)$ and the threshold parameters.
  Output: The threshold public key $\mathbf{pk}^*$.
  Initialization:
    1. Each party $P_k$ derives her public key from the secret key:
         $\mathbf{pk}_k = [pk_{k,1}(x), pk_{k,2}(x)] = [-(a(x)sk_k(x) + pe_k(x)), a(x)]$
       Then broadcasts $\mathbf{pk}_k$ to the other parties.
    2. Once all the $\mathbf{pk}_k$ are received for $k \in \{1, \dots, N\}$ compute:
         $pk_1^* = \sum_{k=1}^{N} pk_{k,1}(x).$
    3. Return $\mathbf{pk}^* = [pk_1^*, a(x)]$.
```

In order to get the share of the threshold secret key, every party P_k only has to randomly choose $sk_k(x) \leftarrow \chi_\beta$. Recall that, at the end of the three protocols above, the threshold secret key will be $sk^* = \sum_{k=1}^{N} sk_k(x)$.

For the generation of the threshold public key, the protocol is described in Algorithm 5.

Let us see that $\mathbf{pk}^*$ is the public key of sk^*. In fact, if one adds all the $pk_{k,1}(x)$ for all $k \in \{1, \dots, N\}$, obtains

$$\sum_{k=1}^{N} pk_{k,1}(x) = \sum_{k=1}^{N} -(a(x)sk_k(x) + pe_k(x)) = -[a(x)\sum_{k=1}^{N} sk_k(x) + p\sum_{k=1}^{N} e_k(x)].$$

The result is trivial from the last expression.

The protocol described in Algorithm 5 is simple and only requires one round of interaction among the parties. The protocol for generating the evaluation key, though, is quite the opposite: it is complex and, indeed, requires more rounds of interaction. In this case, the evaluation key is aimed to be of the form

$$\mathbf{ek}^* = [ek_1^*, ek_2^*] = \left[-(A(x)\sum_{k=1}^{N} sk_k(x) + pE(x)) + \left(\sum_{k=1}^{N} sk_k(x)\right)^2, A(x)\right],$$

where $A(x)$ and $E(x)$ are random polynomials. Observe that the definition of $\mathbf{ek}$ is exactly the same as in classical BGV but changing $sk(x)$ to $\sum_{k=1}^{N} sk_k(x)$. The protocol for the evaluation key generation is shown in Algorithm 6.

There are several points to expand from Algorithm 6:

- Observe that

$$\left(\sum_{k=1}^{N} sk_k(x)\right)^2 = \sum_{k=1}^{N} (sk_k(x))^2 + 2\sum_{\substack{i,j=1\\ i\neq j}}^{N} sk_i(x)sk_j(x).$$

Algorithm 6: Threshold evaluation key generation.

Input: A secret key $sk_k(x)$, the threshold public key $\mathbf{pk}^*$ and the threshold parameters.

Output: The threshold evaluation key $\mathbf{ek}^*$.

Initialization:

1. Each party P_k encapsulates her secret key $sk_k(x)$ by computing:

$$\mathbf{c}_k^k = [c_{k,1}^k, c_{k,2}^k] = [-(\alpha(x)sk_k(x) + pe_k^k(x)) + sk_k(x), \alpha(x)]$$

2. Then, for every $\ell \in \{1, \dots, N\}$ such that $\ell \neq k$, P_k encapsulates the value 0 by computing:

$$\mathbf{c}_k^\ell = [c_{k,1}^\ell, c_{k,2}^\ell] = [-(\alpha(x)sk_k(x) + pe_k^\ell(x)) + 0, \alpha(x)].$$

3. Each party P_k broadcasts to the other parties the values $\{c_k^i\}_{1\leq i\leq N}$. Note that all are encapsulations of 0 but c_k^k that encapsulates $sk_k(x)$.

4. On input $\{\mathbf{c}_1^i\}_{1\leq i\leq N}, \dots, \{\mathbf{c}_N^i\}_{1\leq i\leq N}$, compute:

$$C^1 = \left[\sum_{k=0}^{N} c_{k,1}^1, \alpha(x)\right], \ \dots, \ C^N = \left[\sum_{k=0}^{N} c_{k,1}^N, \alpha(x)\right].$$

At this point, every C^ℓ is encrypting $sk_\ell(x)$ with the correct threshold sk^* for every $\ell \in \{1, \dots, N\}$.

5. For $j \in \{1, \dots, N\}$, P_k samples $e_{\texttt{Eval},j}(x) \leftarrow \chi_{\beta_{\texttt{Eval}}}$. Then P_k encrypts N times the message $m(x) = 0$ using $\mathbf{pk}^*$ and the $e_{\texttt{Eval},j}(x)$'s:

$$\text{Get } \bar{\mathbf{z}}_k^1 \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \text{ and then } \mathbf{z}_k^1 = \left[z_{k,1}^1(x) + pe_{\texttt{Eval},1}(x), z_{k,2}^1(x)\right]$$
$$\vdots$$
$$\text{Get } \bar{\mathbf{z}}_k^N \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \text{ and then } \mathbf{z}_k^N = \left[z_{k,1}^N(x) + pe_{\texttt{Eval},N}(x), z_{k,2}^N(x)\right]$$

6. Each party P_k now computes, for every $\ell \in \{1, \dots, N\}$:

$$\hat{C}_k^\ell = sk_k(x) \cdot C^\ell + \mathbf{z}_k^\ell = \left[sk_k(x)\sum_{k=0}^{N} c_{k,1}^1, sk_k(x)\alpha(x)\right] + \left[z_{k,1}^\ell(x) + pe_{\texttt{Eval},\ell}(x), z_{k,2}^\ell(x)\right].$$

The last addition is performed component-wise.

7. Each party P_k broadcasts $\{\hat{C}_k^\ell\}_{1\leq \ell\leq N}$.

8. On input $\{\hat{C}_1^\ell\}_{1\leq \ell\leq N}, \dots, \{\hat{C}_N^\ell\}_{1\leq \ell\leq N}$, compute $\mathbf{ek}^* = \sum_{k=1}^N \sum_{\ell=1}^N \hat{C}_k^\ell$.

9. Return $\mathbf{ek}^*$.

Thus, part of $\mathbf{ek}^*$ has to be similar to the above formula.

- After performing step 4, if one expands any of the C^ℓ, obtains that its first component C_1^ℓ is:

$$\begin{aligned} C_1^\ell &= \sum_{k=0}^N c_{k,1}^\ell \\ &= \sum_{k=0,k\neq\ell}^N[-(\alpha(x)sk_k(x)+pe_k^\ell(x))] - (\alpha(x)sk_\ell(x)+pe_\ell^\ell(x)) + sk_\ell(x) \\ &= sk_\ell(x) - \sum_{k=0}^N \alpha(x)sk_k(x) + pe_k^\ell(x) \\ &= sk_\ell(x) - \alpha(x)\sum_{k=0}^N sk_k(x) - p\sum_{k=0}^N e_k^\ell(x) \\ &= -\left(\alpha(x)\sum_{k=0}^N sk_k(x) + p\sum_{k=0}^N e_k^\ell(x)\right) + sk_\ell(x) \end{aligned}$$

In consequence, every C^ℓ is encrypting $sk_\ell(x)$ with the correct threshold sk^*.

- Inside every $\mathbf{z}_k^\ell$ for every $k,\ell \in \{1,\dots,N\}$ there is the following:

$$\begin{aligned} \mathbf{z}_k^\ell &= \left[\left(pk_1^*(x)u_k^\ell(x)+pe_{k,1}^\ell(x)\right), \left(pk_2^*(x)u_k^\ell(x)+pe_{k,2}^\ell(x)\right)\right] \\ &= \left[\left(-(a(x)\sum_{i=1}^N sk_i(x)+pE(x))u_k^\ell(x)+pe_{k,1}^\ell(x)\right), \left(a(x)u_k^\ell(x)+pe_{k,2}^\ell(x)\right)\right] \end{aligned}$$

- After step 6, observe that $\hat{C}_k^\ell$ encapsulates $m(x) = sk_k(x)sk_\ell(x)$. Let us abuse on the notation of the polynomials to simplify the formulae below. Indeed, expanding $\hat{C}_k^\ell$:

$$\begin{aligned} \hat{C}_k^\ell &= \left[\left(sk_k\sum_{k=0}^N c_{k,1}^1 + z_{k,1}^\ell + pe_{\texttt{Eval},\ell}\right), \left(sk_k\alpha + z_{k,2}^\ell\right)\right] \\ &= \left[sk_k\left(-(a\sum_{i=1}^N sk_i + pE'\right) + sk_\ell - \left(a\sum_{i=1}^N sk_i + pE\right)u_k^\ell + pe_{\texttt{Eval},\ell}, \left(sk_k a + au_k^\ell + pe_{k,2}^\ell\right)\right] \\ &= \left[-a(sk_k+u_k^\ell)\left(\sum_{i=1}^N sk_i\right) + sk_k sk_\ell + p\left(e_{\texttt{Eval},\ell} - sk_k E' - Eu_k^\ell\right), \left(a(sk_k+u_k^\ell)+pe_{k,2}^\ell\right)\right] \\ &= \left[-A_k^\ell\left(\sum_{i=1}^N sk_i\right) + pE_k^\ell + sk_k sk_\ell, \left(A_k^\ell + pe_{k,2}^\ell\right)\right]. \end{aligned}$$

In the last equality, observe that $A_k^\ell = a(sk_k+u_k^\ell)$ and $E_k^\ell = e_{\texttt{Eval},\ell} - sk_k E' - Eu_k^\ell$ for every $k,\ell \in \{1,\dots,N\}$. Observe that, if one now computes

$$\mathbf{ek}^* = \sum_{k=1}^N\sum_{\ell=1}^N \hat{C}_k^\ell$$

then obtains:

- $A = \sum_{k=1}^N\sum_{\ell=1}^N A_k^\ell = a\left(N\cdot\sum_{k=1}^N sk_k + \sum_{k,\ell=0}^N u_k^\ell\right).$
- $\sum_{k=1}^N\sum_{\ell=1}^N sk_k sk_\ell = \sum_{k=1}^N sk_k sk_k + 2\sum_{\substack{k,\ell=1\\k\neq\ell}}^N sk_k sk_\ell = \left(\sum_{i=1}^N sk_i\right)^2.$

Defining $\mathcal{E}_1 = \sum_{k=1}^N\sum_{\ell=1}^N E_k^\ell$ and $\mathcal{E}_2 = \sum_{k=1}^N\sum_{\ell=1}^N e_{k,2}^\ell$, then $\mathbf{ek}^*$ is of the form:

$$\mathbf{ek}^* = \left[-A\left(\sum_{i=1}^N sk_i\right) + p\mathcal{E}_1 + \left(\sum_{i=1}^N sk_i\right)^2, A + p\mathcal{E}_2\right]$$

As a result, $\mathbf{ek}^*$ is of desired form.

Threshold decryption.

Given a ciphertext $\mathbf{c}$ encrypted with $\mathbf{pk}^*$, parties only can make a partial decryption of $\mathbf{c}$ because they own $sk_k(x)$ but not the other secret keys.

Thus, a threshold decryption protocol is needed to correctly decrypt $\mathbf{c}$. This decryption protocol is described in Algorithm 7.

Algorithm 7: Threshold decryption.

Input: A ciphertext $\mathbf{c} = (c_1(x), c_2(x))$, the secret key $sk_k(x)$ and the threshold parameters.

Output: The message $m(x)$ encapsulated inside $\mathbf{c}$.

Initialization:

1. Each party P_k performs her partial decryption using $sk_k(s)$:
$$\begin{aligned}\omega_k &= c_2(x)sk_k(x) + pe_{\texttt{Dec},k}(x) = [a(x)u(x) + pe_2(x)]sk_k(x) + pe_{\texttt{Dec},k}(x)\\ &= a(x)u(x)sk_k(x) + pe_2(x)sk_k(x) + pe_{\texttt{Dec},k}(x),\end{aligned}$$
for some $e_{\texttt{Dec},k}(x) \leftarrow \chi_{\beta_{\texttt{Dec}}}$. Then broadcasts ω_k to the other parties.
2. Once all the ω_i are received for $i \in \{1, \dots, N\}$ compute:
$$m(x) = \left(c_1(x) + \sum_{i=1}^{N} \omega_i \mod q\right) \mod p$$
3. Return $m(x)$.

Let us observe that:

$$\begin{aligned}c_1(x) + \textstyle\sum_{i=1}^{N} \omega_i &= -\left(a(x)\textstyle\sum_{i=1}^{N} sk_i(x) + pe(x)\right)u(x) + pe_1(x) + m(x) + \textstyle\sum_{i=1}^{N} \omega_i\\ &= -a(x)u(x)\textstyle\sum_{i=1}^{N} sk_i(x) - pe(x)u(x) + pe_1(x) + m(x)\\ &\quad +a(x)u(x)\textstyle\sum_{i=1}^{N} sk_i(x) + pe_2(x)\textstyle\sum_{i=1}^{N} sk_i(x) + p\textstyle\sum_{i=1}^{N} e_{\texttt{Dec},i}(x)\\ &= m(x) + p\left(e(x)u(x) + e_1(x) + e_2\textstyle\sum_{i=1}^{N} sk_i(x) + \textstyle\sum_{i=1}^{N} e_{\texttt{Dec},i}(x)\right)\\ &= m(x) + p\aleph(x)\end{aligned}$$

And thus the noise can be cancelled obtaining $m(x)$ as desired.

5.3.3 The VD protocol using threshold FHE.

In this section, a VD protocol using threshold FHE by restricting the threshold FHE protocol described in Section 5.3.2 to a two-party setting is given. The complete VD protocol can be seen in Figure 5.8.

Observe in Figure 5.8 that the interactions for constructing the threshold public and evaluation keys have been merged. This is proposed in [4] for an N party setting. Also note that, during the homomorphic evaluation, the operation to be computed is a product, where Bob uses $\mathbf{ek}^*$ to obtain the result.

It is important to note, though, that the threshold protocol built in [4] is secure for only *honest-but-curious* adversaries. An honest-but-curious adversary is a party who will try to learn as much as possible from the execution of the protocol because there is interest in it. However, this adversary will allways be honest and follow the protocol, never trying to deviate from it. In multi-party computation, honest-but-curious adversaries are considered the weakest ones.

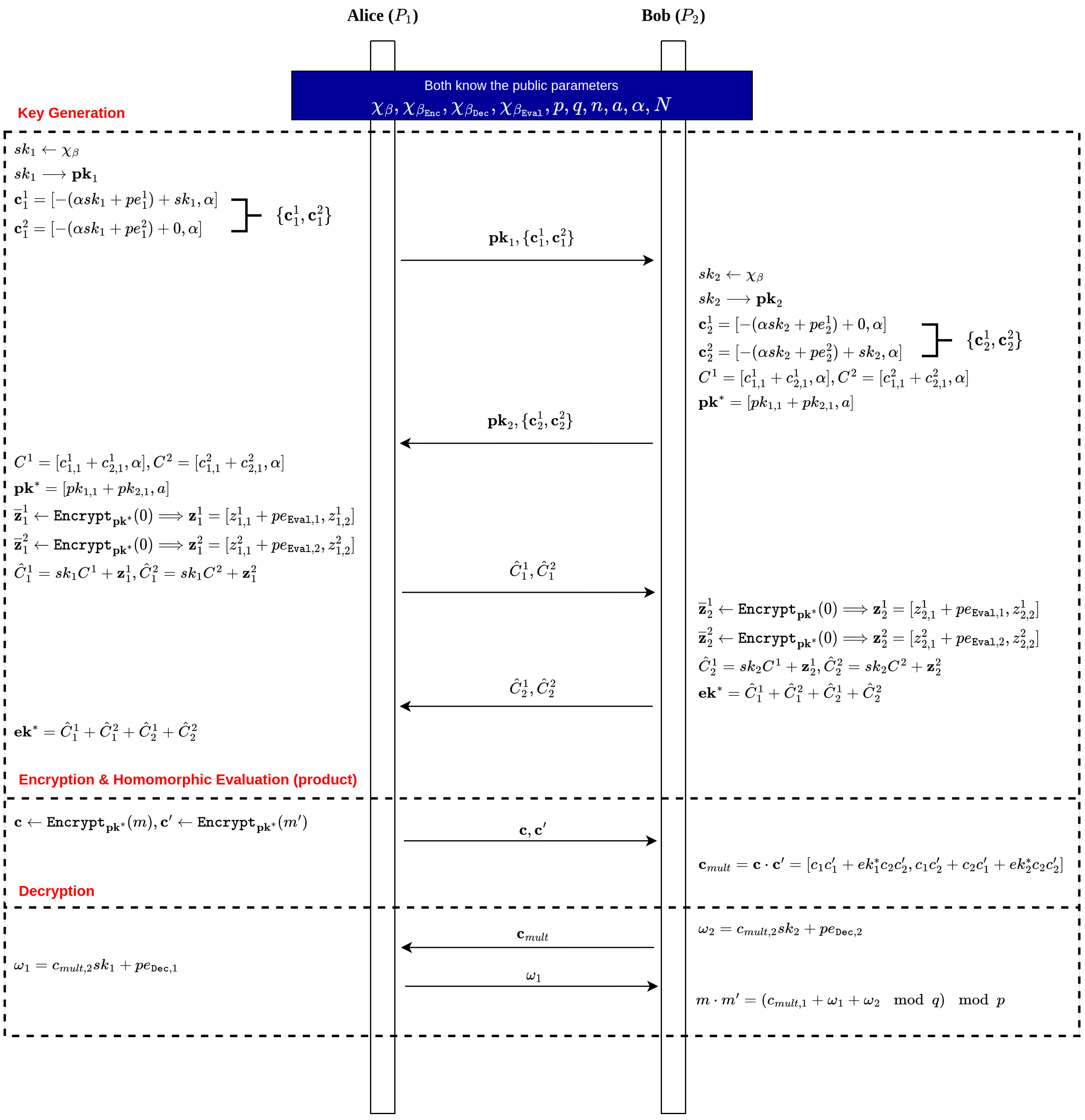

Alice (P_1)
Bob (P_2)
Both know the public parameters
$\chi_\beta, \chi_{\beta_{\texttt{Enc}}}, \chi_{\beta_{\texttt{Dec}}}, \chi_{\beta_{\texttt{Eval}}}, p, q, n, a, \alpha, N$
Key Generation
$sk_1 \leftarrow \chi_\beta$
$sk_1 \longrightarrow \mathbf{pk}_1$
$\mathbf{c}_1^1 = [-(\alpha sk_1 + pe_1^1) + sk_1, \alpha]$
$\mathbf{c}_1^2 = [-(\alpha sk_1 + pe_1^2) + 0, \alpha]$
$\{\mathbf{c}_1^1, \mathbf{c}_1^2\}$
$\mathbf{pk}_1, \{\mathbf{c}_1^1, \mathbf{c}_1^2\}$
$sk_2 \leftarrow \chi_\beta$
$sk_2 \longrightarrow \mathbf{pk}_2$
$\mathbf{c}_2^1 = [-(\alpha sk_2 + pe_2^1) + 0, \alpha]$
$\mathbf{c}_2^2 = [-(\alpha sk_2 + pe_2^2) + sk_2, \alpha]$
$\{\mathbf{c}_2^1, \mathbf{c}_2^2\}$
$C^1 = [c_{1,1}^1 + c_{2,1}^1, \alpha], C^2 = [c_{1,1}^2 + c_{2,1}^2, \alpha]$
$\mathbf{pk}^* = [pk_{1,1} + pk_{2,1}, a]$
$\mathbf{pk}_2, \{\mathbf{c}_2^1, \mathbf{c}_2^2\}$
$C^1 = [c_{1,1}^1 + c_{2,1}^1, \alpha], C^2 = [c_{1,1}^2 + c_{2,1}^2, \alpha]$
$\mathbf{pk}^* = [pk_{1,1} + pk_{2,1}, a]$
$\overline{\mathbf{z}}_1^1 \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \Longrightarrow \mathbf{z}_1^1 = [z_{1,1}^1 + pe_{\texttt{Eval},1}, z_{1,2}^1]$
$\overline{\mathbf{z}}_1^2 \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \Longrightarrow \mathbf{z}_1^2 = [z_{1,1}^2 + pe_{\texttt{Eval},2}, z_{1,2}^2]$
$\hat{C}_1^1 = sk_1 C^1 + \mathbf{z}_1^1, \hat{C}_1^2 = sk_1 C^2 + \mathbf{z}_1^2$
$\hat{C}_1^1, \hat{C}_1^2$
$\overline{\mathbf{z}}_2^1 \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \Longrightarrow \mathbf{z}_2^1 = [z_{2,1}^1 + pe_{\texttt{Eval},1}, z_{2,2}^1]$
$\overline{\mathbf{z}}_2^2 \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(0) \Longrightarrow \mathbf{z}_2^2 = [z_{2,1}^2 + pe_{\texttt{Eval},2}, z_{2,2}^2]$
$\hat{C}_2^1 = sk_2 C^1 + \mathbf{z}_2^1, \hat{C}_2^2 = sk_2 C^2 + \mathbf{z}_2^2$
$\mathbf{ek}^* = \hat{C}_1^1 + \hat{C}_1^2 + \hat{C}_2^1 + \hat{C}_2^2$
$\hat{C}_2^1, \hat{C}_2^2$
$\mathbf{ek}^* = \hat{C}_1^1 + \hat{C}_1^2 + \hat{C}_2^1 + \hat{C}_2^2$
Encryption & Homomorphic Evaluation (product)
$\mathbf{c} \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(m), \mathbf{c}' \leftarrow \texttt{Encrypt}_{\mathbf{pk}^*}(m')$
$\mathbf{c}, \mathbf{c}'$
$\mathbf{c}_{mult} = \mathbf{c} \cdot \mathbf{c}' = [c_1c_1' + ek_1^* c_2c_2', c_1c_2' + c_2c_1' + ek_2^* c_2c_2']$
Decryption
$\omega_2 = c_{mult,2} sk_2 + pe_{\texttt{Dec},2}$
$\mathbf{c}_{mult}$
$\omega_1 = c_{mult,2} sk_1 + pe_{\texttt{Dec},1}$
ω_1
$m \cdot m' = (c_{mult,1} + \omega_1 + \omega_2 \mod q) \mod p$

Figure 5.8: Verifiable decryption protocol for the BGV scheme using threshold FHE.

An adversary that deviates from the protocol to attack its correctness or the integrity of the result is denoted as *malicious*. These are the strongest adversaries usually considered in the literature. It is important to note that, if a protocol is secure for malicious adversaries, then it is secure for honest-but-curious ones. On the contrary, a protocol secure for honest-but-curious adversaries may not be secure for malicious ones.

The authors in [4] consider the typical techniques to shift from security for honest-but-curious to security for malicious adversaries: commitments and zero-knowledge proofs. However, they do not give any example of what technique to use, and neither OpenFHE extended this functionality in their implementation (see [1] for the details).

In conclusion, the protocol described in Section 5.3.2 allows to define a VD protocol, as can be seen in Figure 5.8. This protocol requires a little number of interactions between Alice and Bob and, as such, simplifies its implementation. Moreover, the threshold FHE protocol is already implemented in OpenFHE, which makes it easy to implement the protocol in this library. Nevertheless, this protocol is only secure if both Alice and Bob are to behave honestly during its execution. If Alice decides to deviate from the protocol, then Bob cannot be guaranteed that the result is correct. This is an issue that does not happen in the protocol from [50].

Chapter 6

Implementation and performance

Both the approximation for computing non-linear functions (Algorithm 1) and the VD protocol using threshold FHE (Figure 5.8) have been implemented. This chapter presents the resulting algorithms and their performance depending on the level of security (*i.e.,* the parameters chosen for running BGV).

Currently, there are mainly 4 open-source frameworks implementing the BGV and BFV schemes. Table 6.1 describes them and their principal properties:

	LATTIGO	**HElib**	**SEAL**	**OpenFHE**
Developer	Tune insight	homenc (IBM)	Microsoft	OpenFHE Org
Language	Go	C++17	C++17	C++11
Schemes	BGV/BFV, CKKS	BGV, CKKS	BGV/BFV, CKKS	BGV/BFV, CKKS, TFHE
License	Apache 2.0	Apache 2.0	MIT	BSD 2-Clause

Table 6.1: The frameworks implementing BGV and BFV.

From Table 6.1, HElib and SEAL result to be similar frameworks, since both are implemented using C++17 and implement FHE schemes for both integers and complex numbers. LATTIGO is different from the others because it is implemented in Golang, which is a more portable programming language. However, the most complete one is OpenFHE, which is a library not only implementing block-wise FHE, but bit-wise FHE too.

For this reason, OpenFHE has been used for the implementation. The OpenFHE community developed a CMake file for users to be able to build their own applications using their library. They also have an extensive documentation site[1] with useful information for both new and advanced users. In fact, they show how to build your own application using OpenFHE in `https://openfhe-development.readthedocs.io/en/latest/sphinx_rsts/intro/building_user_applications.html`.

[1]See `https://openfhe-development.readthedocs.io/en/latest/`.

6.1 Non-linear operations on BGV.

The milestone to be able to implement approximations for different non-linear operations consists of implementing correctly Algorithm 1. This algorithm is mainly composed by two subroutines:

- Step 2: Computing the coefficients of Lagrange's interpolation polynomial $L(x)$ using Lagrange's Interpolation Formula.
- Step 4: Computing all the powers of $\mathbf{c} \rightarrow \{\mathbf{c}, \mathbf{c}^2, \ldots, \mathbf{c}^{p-1}\}$ using the Fast Modular Exponentiation Algorithm.

Consequently, auxiliary libraries have been created implementing the above two functionalities. The code for both libraries is shown in Appendix A.

With these subroutines implemented, it is now possible to compute non-linear operations applying Algorithm 1. As an example, assume that Bob wants to compute a non-linear operation over two ciphertexts $\mathbf{c}$ and $\mathbf{c}'$. Such operation could be, for instance, to compute if $\mathbf{c} = \mathbf{c}'$. Let us see how Bob can do this using Algorithm 1. This is described in Algorithm 8.

In Algorithm 8, indeed, observe that for $\mathbb{Z}_5$ and $L(x) = 1 + 4x^4$:

- If $x = 0$ then $L(0) = 1$.
- If $x = 1$ then $L(1) = 1 + 4 = 5 \equiv 0 \mod 5$.
- If $x = 2$ then $L(2) = 1 + 4 \cdot 2^4 = 65 \equiv 0 \mod 5$.
- If $x = 3$ then $L(-2) = L(3)1 + 4 \cdot 3^4 = 325 \equiv 0 \mod 5$.
- If $x = 4$ then $L(-1) = L(4) = 1 + 4^5 = 1025 \equiv 0 \mod 5$.

For this reason, if Bob wants to see if two integers $a, b \in \mathbb{Z}_5$ are equal, he only needs to compute their difference $a - b$ and evaluate $L(x)$ over that difference, *i.e.*, compute $L(a - b)$.

For instance, let $a = 2$ and $b = 3$. Then, $a - b = -1 \equiv 4 \mod 5$. Since $L(4) = 0$, Bob knows a and b are not equal. If $a = b = 4$, however, $a - b = 0$ and then $L(0) = 0$, so Bob knows a and b are equal. Algorithm 8 does exactly this over the ciphertexts.

In Appendix B an implementation of this algorithm is given. There have been implemented other non-linear operations too, and the algorithms are very similar to Algorithm 8, mainly differing on the function to be evaluated in step 1.

- Greater than: in this case $f(x)$ is

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } 0 < x \leq \frac{p-1}{2} \\ 0 & \text{otherwise} \end{cases}$$

- Greater or equal than: in this case $f(x)$ is

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 1 & \text{if } 0 < x \leq \frac{p-1}{2} \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 8: Homomorphic approximation of the equality test using Lagrange's interpolation.

Input: Two ciphertexts $\mathbf{c}$ and $\mathbf{c}'$.
Output: A ciphertext $\mathbf{c}_{\texttt{Eq}}$ telling if $\mathbf{c} = \mathbf{c}'$.
Initialization:

1. For every $x \in \mathbb{Z}_p$, compute:
$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$
   Thus, at this point Bob knows X and Y. For example, if $p = 5$
   - $X = \mathbb{Z}_p = \{0, 1, 2, -2, -1\}$
   - $Y = \{f(0) = 1, f(1) = 0, f(2) = 0, f(-2) = 0, f(-1) = 0\}$.
2. Compute $L(x)$ using Lagrange's interpolation formula over $\mathbb{Z}_p$ and the sets X and Y from the previous step. Using Lagrange's Interpolation Library, for $p = 5$ and the above sets we obtain:
$$L(x) = 1 + 4x^4.$$
3. Encrypt every coefficient of $L(x)$. In other words, compute $\mathbf{s}_i = \texttt{Encrypt}(l_i)$, for $i = 0, \dots, p-1$, using $\mathbf{pk}$. Store the encrypted values, for $p = 5$ we assume them to be $\{\mathbf{s}_0, \dots, \mathbf{s}_4\}$.
4. Compute homomorphically $\mathbf{c}_{\texttt{Sub}} = \mathbf{c} - \mathbf{c}'$. Note that, if $\mathbf{c} = \mathbf{c}'$, then $\mathbf{c}_{\texttt{Sub}}$ encapsulates a 0, and other element in $\mathbb{Z}_p$ otherwise.
5. Compute homomorphically the set of powers $\{\mathbf{c}_{\texttt{Sub}}, \mathbf{c}^2_{\texttt{Sub}}, \dots, \mathbf{c}^{p-1}_{\texttt{Sub}}\}$ using the Fast Modular Exponentiation Library. Following the example of $p = 5$ we have computed $\{\mathbf{c}_{\texttt{Sub}}, \mathbf{c}^2_{\texttt{Sub}}, \mathbf{c}^3_{\texttt{Sub}}, \mathbf{c}^4_{\texttt{Sub}}\}$.
6. Compute $L(\mathbf{c}_{\texttt{Sub}})$. In the example we have
$$\mathbf{c}_{\texttt{Eq}} = \mathbf{s}_0 + \mathbf{s}_1\mathbf{c}_{\texttt{Sub}} + \cdots + \mathbf{s}_4\mathbf{c}^4_{\texttt{Sub}}.$$
7. Return $\mathbf{c}_{\texttt{Eq}}$. This ciphertext is encrypting the evaluation of $L(m - m')$, where m and m' are the messages encapsulated by $\mathbf{c}$ and $\mathbf{c}'$, respectively.

- Lower than: in this case $f(x)$ is

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 0 & \text{if } 0 < x \leq \frac{p-1}{2} \\ 1 & \text{otherwise} \end{cases}$$

- Lower or equal than: in this case $f(x)$ is

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } 0 < x \leq \frac{p-1}{2} \\ 1 & \text{otherwise} \end{cases}$$

- Sign function: in this case $f(x)$ is

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } 0 < x \leq \frac{p-1}{2} \\ -1 & \text{otherwise} \end{cases}$$

- $\min(a, b)$: in this case $f(x)$ is the same as in 'lower or equal', but it is evaluated two times: for $a - b$ and for $b - a$. Then the minimum can be computed as follows:

$$\min(a, b) = f(a - b) \cdot a + f(b - a) \cdot b.$$

- $\max(a, b)$: in this case $f(x)$ is the same as in 'greater or equal', but it is evaluated two times: for $a - b$ and for $b - a$. Then the maximum can be computed as follows:

$$\max(a, b) = f(a - b) \cdot a + f(b - a) \cdot b.$$

- Integer division (by known plaintext d): in this case $f(x)$ is

$$f(x) = \frac{x}{d}.$$

  The division is understood as integer division (without decimals).

- Integer division (by unknown ciphertext $\mathbf{c}$): in this case, the execution of Algorithm 8 computing the integer division by a known plaintext must be repeated for all plaintext $d \in \mathbb{Z}_p$. In other words, Algorithm 8 must be executed $p - 1$ times to obtain:

$$\{\mathbf{c}_{\texttt{Div},1}, \ldots, \mathbf{c}_{\texttt{Div},p-1}\},$$

  where $\mathbf{c}_{\texttt{Div},k}$ is the ciphertext encrypting $\frac{m}{k}$ for every $k \in \mathbb{Z}_p$. Similarly, Algorithm 8 is also executed for the equality test another $p - 1$ times obtaining

$$\{\mathbf{c}_{\texttt{Eq},1}, \ldots, \mathbf{c}_{\texttt{Eq},p-1}\}.$$

  Then, the ciphertext

$$c_{\texttt{PrivDiv}} = \mathbf{c}_{\texttt{Div},1}\mathbf{c}_{\texttt{Eq},1} + \cdots + \mathbf{c}_{\texttt{Div},p-1}\mathbf{c}_{\texttt{Eq},p-1}$$

  encrypts the result of the private division. Observe from this last equation that all the $\mathbf{c}_{\texttt{Div},k}$ are multiplied by 0 but one: $\mathbf{c}_{\texttt{Div},d}$.

All the above algorithms have been implemented and the code is available in the GitHub repository in [7]. Just follow the instructions available in the README file in order to use the library.

6.2 Implementation of the VD protocol on OpenFHE.

Similarly to the implementation of non-linear operations over BGV, for the implementation of the VD protocol on OpenFHE, a wrapper has been created encapsulating the main functionalities of threshold FHE. The contents of this wrapper can be seen in Appendix A.3.

The non-linear operations implemented in Section 6.1 have been implemented using the threshold FHE wrapper shown in Appendix A.3 too. This way, Bob can compute those non-linear operations making it possible for him to know the result without knowing Alice's inputs. In other words, we implemented the verifiable decryption protocol shown in Figure 5.8 for every operation implemented in Section 6.1.

An example code for computing the equality test can be seen in Appendix C. This implementation of the computation of non-linear operations using verifiable decryption is also available in the same GitHub repository in [7], where the non-linear operations were implemented.

6.3 Performance.

In this section, we give an overview of the performance results of our solution, for both the classical BGV encryption and threshold FHE settings. The OpenFHE framework implements different security levels for the computations:

- Without security.
- 128-bit security.
- 192-bit security.
- 256-bit security.

The obtained performance results can be seen in Table 6.2. All the executions have been carried out on a 16GB RAM and 8 CPU computer with an 11th Gen Intel(R) Core(TM) i7-1165G7 processor of 2.80GHz. All timings in Table 6.2 are given in seconds.

It is important to note that the cryptographical context (mainly the plaintext space $\mathcal{P} = \mathbb{Z}_p$ and the degree of the cyclotomic polynomial n) differ a lot depending on the security level considered to perform the executions. Besides, in OpenFHE n must divide $p-1$. Namely, p and n are set to the following depending on the security level:

- Without security: no conditions on choosing p and n are required but the typical ones (n a power of two and $n|p-1$). For instance, $p = 257$ and $n = 2$.
- With any kind of security: the cyclotomic polynomial degree must be at least $n = 131072$, and the lowest polynomial satisfying $n|p-1$ is $p = 786433$.

		Security			
		Not set	128-bit	192-bit	256-bit
Classic BGV	Equality	0.28	396.89	379.72	386.06
	Greater	0.32	415.36	411.36	405.17
	Greater or equal	0.30	413.27	384.30	390.18
	Lower	0.36	437.28	419.83	410.64
	Lower or equal	0.32	428.48	407.93	430.18
	Sign	0.35	450.99	426.98	442.07
	Minimum	0.65	820.66	822.95	825.94
	Maximum	0.61	831.32	830.21	836.50
	Division (public)	0.29	403.26	383.69	408.22
	Division (private)	95.47	106202.19	100218.33	104245.67
Threshold	Key generation	0.01	2.49	2.46	2.49
	Equality	0.32	453.65	429.23	456.88
	Greater	0.35	480.83	480.64	488.96
	Greater or equal	0.35	469.33	464.36	455.97
	Lower	0.36	476.62	473.86	468.46
	Lower or equal	0.39	486.95	487.81	473.37
	Sign	0.31	462.01	457.48	470.97
	Minimum	0.63	948.18	938.98	943.54
	Maximum	0.62	945.87	941.56	943.46
	Division (public)	0.31	448.18	447.43	458.36
	Division (private)	113.44	142941.76	115121.56	118564.78

Table 6.2: Performance of the implemented algorithms for different security settings.

Observe that, to give the system a minimum level of security (*i.e.*, 128-bit) the prime p must be very large. What is more, this prime seriously affects the performance of our solution: Algorithm 1 needs to loop over p^3 to compute step 2, and step 4 is very heavy to compute too. Choosing the lowest parameters possible, the interpolation polynomial computation would last more than a week. For this reason, from 128-bit security on, only the integer points from [-128, 128] have been used (for the polynomial length to be equal to the non-secure setting). This way, the execution time is highly reduced and we continue to be able to compute correctly the functions for the first integers.

Chapter 7

Conclusions and further work.

The present document, in the first place, serves as a state of the art of the most important current FHE constructions. Mainly focusing on BGV and BFV, also CKKS and TFHE have been introduced and the differences among these four systems have arised throughout the document.

Nevertheless, BGV and BFV stand out from the other schemes, firstly because, despite being one of the first FHE schemes, they continue to be used and still are considered to be secure. Secondly, they are simple schemes, which makes them easy to use and implement. BGV and BFV allow to compute linear operations homomorphically over the integers.

Non-linear operations, however, are not trivial to compute on those schemes. For this reason, this work presented a solution to approximate several non-linear operations using Lagrange's polynomial interpolation homomorphically: equalities, comparisons, minimums and maximums, and integer divisions. The interpolating technique and all those operations have been implemented in OpenFHE, an open-source FHE library implementing different schemes that results to be the most complete one because of its several implementations and their underlying security.

In addition, this document also introduces the verifiable decryption problem in FHE. Verifiable decryption is useful when the result of the homomorphic evaluation is needed by someone else than the data owner. For that person to be sure that the received result is the correct decryption of the evaluation, two protocols have been described: the one using lattice-based zero-knowledge proofs in [50] and the threshold BGV protocol built in [4]. Despite being less secure than the former, the latter is straightforward to implement in OpenFHE, since they have developed a library implementing this threshold protocol.

In consequence, both the approximations for non-linear operations using Lagrange's interpolator and the threshold FHE based verifiable decryption protocol have been implemented. The implementation can be seen in [7]. We also have given some performance results in Table 6.2. The performance of the implemented operations is not good (mainly for secure BGV implementations), but considering the functionality that these non-linear operations may offer to the BGV scheme, they could be bearable in several situations.

There are some important issues that have to be considered, though.

- Steps 2 and 4 from Algorithm 1 are too time consuming. In this work, it is shown that it is possible to compute those polynomials for low primes, such as $p = 257$. However, the execution time grows quickly when trying to give security to the system. This is a problem to be considered in the future, either trying to find other approximation techniques, or relaxing the security conditions of the BGV scheme.

- The threshold protocol implemented in OpenFHE is secure only for honest but curious adversaries. However, even without having much technical background, Alice is able to deviate from the threshold protocol, making the correctness of the final decryption to fail with Bob not noticing it. There is work to be done in this setting, either trying to extend the protocol in [4] to one secure against malicious adversaries, or trying to implement the VD protocol in [50].
- In this document, only BGV (and BFV by extension) has been considered as the FHE scheme to be extended. However, the other FHE schemes (CKKS or TFHE) suffer from the same issues than the former. We leave extending the functionalities of this work to other FHE schemes as further work to be done.

In conclusion, this document is the result of a twofold research: the implementation of non-linear operations in BGV and the implementation of a threshold protocol in the same FHE scheme. Extending the BGV to a more complete FHE scheme implies making it usable for more applications, which was the main goal of the present document.

Bibliography

[1] Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D. B., Erabelli, S., Genise, N., ... & Zucca, V. (2022, November). OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (pp. 53-63).

[2] Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., ... & Vaikuntanathan, V. (2021). Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, 31-62.

[3] Aranha, D. F., Baum, C., Gjøsteen, K., Silde, T., & Tunge, T. (2021, May). Lattice-based proof of shuffle and applications to electronic voting. In Topics in Cryptology–CT-RSA 2021: Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17–20, 2021, Proceedings (pp. 227-251). Cham: Springer International Publishing.

[4] Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., & Wichs, D. (2012). Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31* (pp. 483-501). Springer Berlin Heidelberg.

[5] Baum, C., Bootle, J., Cerulli, A., Del Pino, R., Groth, J., & Lyubashevsky, V. (2018, July). Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II* (pp. 669-699). Cham: Springer International Publishing.

[6] Baum, C., Damgård, I., Lyubashevsky, V., Oechsner, S., & Peikert, C. (2018, August). More efficient commitments from structured lattice assumptions. In Security and Cryptography for Networks: 11th International Conference, SCN 2018, Amalfi, Italy, September 5–7, 2018, Proceedings (pp. 368-385). Cham: Springer International Publishing.

[7] Bernabé-Rodríguez, J. (2023): OpenFHE extend BGV. GitHub repository: `https://github.com/openfheorg/openfhe-development/blob/main/src/pke/examples/threshold-fhe.cpp`.

[8] Bendlin, R., & Damgård, I. (2010). Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *Theory of Cryptography: 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings 7* (pp. 201-218). Springer Berlin Heidelberg.

[9] Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P. M., & Sahai, A. (2018). Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38* (pp. 565-596). Springer International Publishing.

[10] Boschini, C., Camenisch, J., Ovsiankin, M., & Spooner, N. (2020). Efficient post-quantum SNARKs for RSIS and RLWE and their applications to privacy. In *Post-Quantum Cryptography: 11th International Conference, PQCrypto 2020, Paris, France, April 15–17, 2020, Proceedings 11* (pp. 247-267). Springer International Publishing.

[11] Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2014): (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3), 1-36.

[12] Brakerski, Z., & Perlman, R. (2016, July). Lattice-based fully dynamic multi-key FHE with short ciphertexts. In *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I* (pp. 190-213). Berlin, Heidelberg: Springer Berlin Heidelberg.

[13] Carr, C., Costache, A., Davies, G. T., Gjøsteen, K., & Strand, M. (2018). Zero-knowledge proof of decryption for FHE ciphertexts. *Cryptology ePrint Archive.*

[14] Chen, H., Chillotti, I., & Song, Y. (2019). Multi-key homomorphic encryption from TFHE. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part II 25* (pp. 446-472). Springer International Publishing.

[15] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017, December): Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security* (pp. 409-437). Springer, Cham.

[16] Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2020): TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1), 34-91.

[17] Costache, A., Smart, N. P., Vivek, S., & Waller, A. (2017). Fixed-point arithmetic in SHE schemes. In *Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers 23* (pp. 401-422). Springer International Publishing.

[18] Damgård, I., Pastro, V., Smart, N., & Zakarias, S. (2012). Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (pp. 643-662). Springer Berlin Heidelberg.

[19] Ducas, L., & Micciancio, D. (2015). FHEW: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia,*

Bulgaria, April 26-30, 2015, Proceedings, Part I 34 (pp. 617-640). Springer Berlin Heidelberg.

[20] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory, 31(4)*, 469-472.

[21] Fan, J., & Vercauteren, F. (2012): Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch., 2012*, 144.

[22] Fiat, A., & Shamir, A. (1987). How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO'86: Proceedings 6* (pp. 186-194). Springer Berlin Heidelberg.

[23] Gennaro, R., Minelli, M., Nitulescu, A., & Orrù, M. (2018, October). Lattice-based zk-SNARKs from square span programs. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 556-573).

[24] Gentry, C. (2009, May): Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing* (pp. 169-178).

[25] Gentry, C., Sahai, A., & Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I* (pp. 75-92). Springer Berlin Heidelberg.

[26] Gjøsteen, K., Haines, T., Müller, J., Rønne, P., & Silde, T. (2022, November). Verifiable decryption in the head. In *Information Security and Privacy: 27th Australasian Conference, ACISP 2022, Wollongong, NSW, Australia, November 28–30, 2022, Proceedings* (pp. 355-374). Cham: Springer International Publishing.

[27] Groth, J. (2010). A Verifiable Secret Shuffle of Homomorphic Encryptions. *Journal of Cryptology, 23*(4).

[28] IEEE Computer Society (2019-07-22). IEEE Standard for Floating-Point Arithmetic. *IEEE STD 754-2019. IEEE. pp. 1–84.* doi:10.1109/IEEESTD.2019.8766229. ISBN 978-1-5044-5924-2. IEEE Std 754-2019.

[29] Iliashenko, I., & Zucca, V. (2021). Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies, 2021*(3), 246-264.

[30] Jain, A., Rasmussen, P. M., & Sahai, A. (2017). Threshold fully homomorphic encryption. *Cryptology ePrint Archive.*

[31] Joye, M. (2021). Guide to fully homomorphic encryption over the [discretized] torus. *Cryptology ePrint Archive.*

[32] Libert, B., Ling, S., Nguyen, K., & Wang, H. (2018, July). Lattice-based zero-knowledge arguments for integer relations. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II* (pp. 700-732). Cham: Springer International Publishing.

[33] Lindell, Y. (2020). Secure multiparty computation. *Communications of the ACM, 64(1),* 86-96.

[34] Lyubashevsky, V., & Neven, G. (2017). One-shot verifiable encryption from lattices. In *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part I 36* (pp. 293-323). Springer International Publishing.

[35] Lyubashevsky, V., Nguyen, N. K., & Seiler, G. (2020, October). Practical lattice-based zero-knowledge proofs for integer relations. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (pp. 1051-1070).

[36] Lyubashevsky, V., Nguyen, N. K., & Seiler, G. (2021, May). Shorter lattice-based zero-knowledge proofs via one-time commitments. In *Public-Key Cryptography–PKC 2021: 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10–13, 2021, Proceedings, Part I* (pp. 215-241). Cham: Springer International Publishing.

[37] Lyubashevsky, V., Peikert, C., & Regev, O. (2013). On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM), 60(6)*, 1-35.

[38] Micciancio, D. (2001). The shortest vector in a lattice is hard to approximate to within some constant. *SIAM journal on Computing, 30(6)*, 2008-2035.

[39] Morais, E., Koens, T., Van Wijk, C., & Koren, A. (2019). A survey on zero knowledge range proofs and applications. *SN Applied Sciences, 1*, 1-17.

[40] Morimura, K., Maeda, D., & Nishide, T. (2022, August). Improved Integer-wise Homomorphic Comparison and Division based on Polynomial Evaluation. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (pp. 1-10).

[41] Mukherjee, P., & Wichs, D. (2016). Two round multiparty computation via multi-key FHE. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35* (pp. 735-763). Springer Berlin Heidelberg.

[42] Narumanchi, H., Goyal, D., Emmadi, N., & Gauravaram, P. (2017, March). Performance analysis of sorting of FHE data: integer-wise comparison vs bit-wise comparison. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)* (pp. 902-908). IEEE.

[43] Okada, H., Cid, C., Hidano, S., & Kiyomoto, S. (2019). Linear depth integer-wise homomorphic division. In *Information Security Theory and Practice: 12th IFIP WG 11.2 International Conference, WISTP 2018, Brussels, Belgium, December 10–11, 2018, Revised Selected Papers 12* (pp. 91-106). Springer International Publishing.

[44] Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the*

Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18 (pp. 223-238). Springer Berlin Heidelberg.

[45] Peikert, C., & Shiehian, S. (2016, October). Multi-key FHE from LWE, revisited. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II* (pp. 217-238). Berlin, Heidelberg: Springer Berlin Heidelberg.

[46] Regev, Oded (2005). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-Seventh annual ACM symposium on Theory of computing - STOC '05.* New York, NY, USA: ACM. pp. 84–93.

[47] Rivest, R. L., Adleman, L., & Dertouzos, M. L. (1978). On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11), 169-180.

[48] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 120-126.

[49] Schnorr, C. P. (1990). Efficient identification and signatures for smart cards. In Advances in Cryptology—CRYPTO'89 Proceedings 9 (pp. 239-252). Springer New York.

[50] Silde, T. (2021). Verifiable decryption for BGV. *Cryptology ePrint Archive.*

Appendix A

Auxiliary libraries.

The two main auxiliary libraries are shown in this section: library for Lagrange's polynomial interpolation and library for Fast Modular Exponentiation.

A.1 Lagrange's Polynomial Interpolation library.

```
// Created on February 13 2023
// By Julen Bernabe Rodriguez <julen.bernabe@tecnalia.com>
// Copyright (c) 2023 Tecnalia Research & Innovation

/*
 * BGV basics
 */

#include "scheme/bgvrns/cryptocontext-bgvrns.h"
#include "gen-cryptocontext.h"

#include <iostream>
#include <fstream>
#include <limits>
#include <iterator>
#include <random>

using namespace lbcrypto;

/**
 * @brief Structure for storing X and Y sets from Step 1
 *
 * @param x: array containing the elements of X
 * @param fx: array containing the elements of Y (the f(x)'s)
 */
struct Points {
    std::vector<int64_t> x;
    std::vector<int64_t> fx;
};

typedef struct Points interpolationPoints;
/**
 * @brief Compute the product between two polynomials mod p
 *
 * @param px first polynomial
 * @param qx second polynomial
 * @param p prime number
 * @return std::vector<int64_t> containing coefficients of result
 */
std::vector<int64_t> polyProd(std::vector<int64_t> px, std::vector<int64_t> qx, int p) {
    std::vector<int64_t> rx;
    int rsize = px.size() + qx.size() - 1;
    for (int i = 0; i < rsize; i++) {
         rx.push_back(0);
    }
    for (uint i = 0; i < px.size(); i++) {
       for (uint j = 0; j < qx.size(); j++) {
          rx[i + j] += px[i] * qx[j];
          rx[i + j] = rx[i + j] % p;
       }
    }
    return rx;
}
/**
 * @brief Compute the addition of two polynomials mod p
 *
 * @param px first polynomial
 * @param qx second polynomial
 * @param p prime number
 * @return std::vector<int64_t> containing coefficients of result
 */
std::vector<int64_t> polyAdd(std::vector<int64_t> px, std::vector<int64_t> qx, int p) {
    std::vector<int64_t> rx;
    uint maxDegree = std::max(px.size(), qx.size());
    for (uint i = 0; i < maxDegree; i++) {
        rx.push_back(0);
        if (i < px.size()) {
            rx[i] = (rx[i] + px[i]) % p;
        }
        if (i < qx.size()) {
            rx[i] = (rx[i] + qx[i]) % p;
        }
    }
    return rx;
}
/**
 * @brief Make coefficients of polynomial be mod p
 *
 * @param px polynomial to be normalized
 * @param p prime number
 * @return std::vector<int64_t> containing the normalized coefficients
 */
std::vector<int64_t> normalizePoly(std::vector<int64_t> px, int p) {
    std::vector<int64_t> rx;
    for (uint i = 0; i < px.size(); i++) {
        rx.push_back(0);
        if (px[i] < 0) {
            rx[i] = uint(p + px[i]);
        } else {
            rx[i] = uint(px[i]);
        }
    }
    return rx;
}
/**
 * @brief Get the Lagrange Polynomial using Lagrange's Interpolation Formula
 *
 * @param ip interpolation points (struct containing X and Y from Step 1)
 * @param p prime number
 * @return std::vector<int64_t>
 */
std::vector<int64_t> getLagrangePoly(interpolationPoints ip, int p) {
    std::vector<int64_t> result;
    for (uint i = 0; i < ip.x.size(); i++) {///< initialize polynomial for the result
        result.push_back(0);
    }
    for (uint i = 0; i < ip.x.size(); i++) {///< loop for computing the summation
        std::vector<int64_t> roundResult = {1};
        int64_t denominator = 1;
        for (uint j = 0; j < ip.x.size(); j++) {///< loop for computing prod(x-xj)(xi-xj)
            if (i != j) {
                std::vector<int64_t> monomial = {-ip.x[j], 1};
                roundResult = polyProd(roundResult, monomial, p);
                denominator = (denominator * (ip.x[i] - ip.x[j])) % p;
            }
        }
        std::vector<int64_t> scalar = {inverse(denominator, p) * ip.fx[i]};///< compute
            inverse of prod(xi-xj)
        roundResult = polyProd(roundResult, scalar, p);
        result = polyAdd(roundResult, result, p);
    }
    return normalizePoly(result, p);
}
/**
 * @brief Encrypt the coefficients of the interpolation polynomial
 *
 * @param poly polynomial to be encrypted
 * @param cc cryptographical context for the encryption
 * @return std::vector<Ciphertext<DCRTPoly>> array of ciphertexts with the encryption of
     each coefficient
 */
std::vector<Ciphertext<DCRTPoly>> encryptInterpolator(std::vector<int64_t> poly,
    cryptoTools cc) {
    std::vector<Ciphertext<DCRTPoly>> result;
    for (uint i = 0; i < poly.size(); i++){
        result.push_back(encrypt(poly[i], cc));
    }
    return result;
}
/**
 * @brief Evaluate Lagrange's Polynomial for some ciphertext c
 *
 * @param powers the powers of c (i.e. {c, c^2, ..., c^{p-1}})
 * @param polynomial Lagranges Interpolation Polynomial
 * @param cc cryptographical context
 * @return Ciphertext<DCRTPoly> ciphertext containing the result of the evaluation
 */
Ciphertext<DCRTPoly> evalInterpolator(std::vector<Ciphertext<DCRTPoly>> powers, std::vector
    <Ciphertext<DCRTPoly>> polynomial, cryptoTools cc) {
    Ciphertext<DCRTPoly> result = encrypt(0, cc);
    result = cc.cryptoContext->EvalAdd(result, polynomial[0]);
    for (uint i = 0; i < powers.size(); i++) {
        Ciphertext<DCRTPoly> product = cc.cryptoContext->EvalMult(powers[i], polynomial[i
            +1]);
        result = cc.cryptoContext->EvalAdd(result, product);
    }
    return result;
}
```

A.2 Fast Modular Exponentiation library.

```
// Created on February 13 2023
// By Julen Bernabe Rodriguez <julen.bernabe@tecnalia.com>
// Copyright (c) 2023 Tecnalia Research & Innovation

/*
 * Comparaciones en BGV
 */

#include "scheme/bgvrns/cryptocontext-bgvrns.h"
#include "gen-cryptocontext.h"

#include <iostream>
#include <fstream>
#include <limits>
```

```
#include <iterator>
#include <random>
#include <time.h>

using namespace lbcrypto;

/**
 * @brief Compute vector of ciphertexts containing: {c^{2^0}, c^{2^1}, c^{2^2}, ..., c^{2^{
     bit-length}}
 *
 * @param ciphertext the ciphertext to use as input
 * @param bitLength bit length of p
 * @param cc cryptographical context
 * @return std::vector<Ciphertext<DCRTPoly>> containing {c^{2^0}, c^{2^1}, c^{2^2}, ..., c
     ^{2^{bit-length}}
 */
std::vector<Ciphertext<DCRTPoly>> powersOfTwo(Ciphertext<DCRTPoly> ciphertext, uint
    bitLength, cryptoTools cc) {
    // Initialize vector of ciphertexts containing: {c^{2^0}, c^{2^1}, c^{2^2}, ..., c^{2^{
        bit-length}}
    std::vector<Ciphertext<DCRTPoly>> preComputedValues;

    // Add c^{2^0} = c to preComputedValues
    preComputedValues.push_back(ciphertext);

    // Fill preComputedValues with remaining powers
    for (uint i = 1; i <= bitLength; i++) {
        preComputedValues.push_back(cc.cryptoContext->EvalMult(preComputedValues[i-1],
            preComputedValues[i-1]));
    }
    return preComputedValues;
}

/**
 * @brief Compute array of powers of ciphertexts: {c, c^2, ..., c^{p-1}}
 *
 * @param ciphertext the ciphertext to use as input
 * @param cc cryptographical context
 * @return std::vector<Ciphertext<DCRTPoly>> containing {c, c^2, ..., c^{p-1}}
 */
std::vector<Ciphertext<DCRTPoly>> powers(Ciphertext<DCRTPoly> ciphertext, cryptoTools cc) {
    uint max = (cc.cryptoContext->GetCryptoParameters()->GetPlaintextModulus()) - 1;

    // Compute binary representation of exponent
    std::vector<uint> binaryRep = binaryRepresentationOfExp(max);

    // Compute vector {c^{2^0}, c^{2^1}, c^{2^2}, ..., c^{2^{bit-length}}
    std::vector<Ciphertext<DCRTPoly>> preComputedValues = powersOfTwo(ciphertext, binaryRep
        .size(), cc);

    // Structure containing the result
    std::vector<Ciphertext<DCRTPoly>> result;

    // Iterate over the exponents of c
    for (uint i = 1; i <= max; i++) {
        // Compute binary representation of exponent
        std::vector<uint> binaryRep = binaryRepresentationOfExp(i);

        // Create new ciphertext to compute the result. Ciphertext is initialized by
            encrypting a 1.
        std::vector<int64_t> vectorOfOne = {1};
        Plaintext plaintextOne                  = cc.cryptoContext->MakePackedPlaintext(
            vectorOfOne);
        Ciphertext<DCRTPoly> ciphertextResult = cc.cryptoContext->Encrypt(cc.keyPair.
            publicKey, plaintextOne);

        // Compute encrypted results using preComputedValues
        for (uint j = 0; j <= binaryRep.size(); j++) {
            if (binaryRep[j] == 1) {
                ciphertextResult = cc.cryptoContext->EvalMult(ciphertextResult,
                    preComputedValues[j]);
            }
        }
        result.push_back(ciphertextResult);
    }
    return result;
}
```

A.3 Wrapper for VD protocol using threshold FHE.

```
/**
 * @ Author: Julen Bernabe Rodriguez <julen.bernabe@tecnalia.com>
 * @ Create Time: 2023-06-14 11:50:19
 * @ Description: Copyright (c) 2023 Tecnalia Research & Innovation
 */

/*
 * BGV basics
 */

#include "scheme/bgvrns/cryptocontext-bgvrns.h"
#include "gen-cryptocontext.h"

#include <iostream>
#include <fstream>
#include <limits>
#include <iterator>
#include <random>

using namespace lbcrypto;

/**
 * @brief crypto contains all the cryptographical information used in threshold decryption
 *
 * @param cryptoContext contains the parameters definings the BGV encryption (n, p, q...)
 * @param pks contains the list of public keys forming pk*
 * @param sks contains the list of secret keys forming sk* (only used for decryption)
 * @param lastKey contains the index of the last public key, for players to be able to use
     it during encryption
 */
struct crypto {
    CryptoContext<DCRTPoly> cryptoContext;
    std::vector<PublicKey<DCRTPoly>> pks;
    std::vector<PrivateKey<DCRTPoly>> sks;
    uint lastKey;
};

/**
 * @brief threshold contains the evaluation key
 *
 * @param AddedKey contains the partial evaluation key encapsulating (sk1+...+skn)
 * @param MultKey contains the evaluation key
 */
struct threshold {
    EvalKey<DCRTPoly> AddedKey;
    EvalKey<DCRTPoly> MultKey;
};

typedef struct crypto cryptoTools;
typedef struct threshold thresholdTools;

/**
 * @brief generate the cryptographical context for threshold BGV using the security
     parameters
 *
 * @param ptm plaintext modulus
 * @param multDepth max number of products that we want to compute over the same ciphertext
 * @param level ring dimension
 * @return CryptoContext<DCRTPoly> cryptographical context
 */
CryptoContext<DCRTPoly> GenerateThresholdBGVrnsContext(usint ptm, usint multDepth, usint
    level) {
    CCParams<CryptoContextBGVRNS> parameters;
    parameters.SetPlaintextModulus(ptm);
    parameters.SetMultiplicativeDepth(multDepth);
    parameters.SetSecurityLevel(HEStd_NotSet);
    parameters.SetMultipartyMode(NOISE_FLOODING_MULTIPARTY);
    parameters.SetRingDim(level);
    parameters.SetKeySwitchTechnique(HYBRID);
    parameters.SetScalingTechnique(FIXEDAUTO);

    CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
    cc->Enable(PKE);
    cc->Enable(KEYSWITCH);
    cc->Enable(LEVELEDSHE);
    cc->Enable(ADVANCEDSHE);
    cc->Enable(MULTIPARTY);

    return cc;
}

/**
 * @brief generate cryptoTools (cryptographical context + public keys + secret keys)
 *
 * @param p plaintext modulus
 * @param level ring dimension
 * @return cryptoTools (cryptographical context + public keys + secret keys)
 */
cryptoTools genThresholdBGVCryptoTools(usint p, usint level) {
    cryptoTools cc;

    // compute binary representation of p-1
    std::vector<uint> binaryRep = binaryRepresentationOfExp(p-1);

    // Define parameters of BGV cryptographic context
    usint ptm                   = p;
    usint depth                 = binaryRep.size() + 1;      // Maximum depth needed is the
        binary representation of p-1

    // Generate context with above parameters
    cc.cryptoContext = GenerateThresholdBGVrnsContext(ptm, depth, level);

    // Key generation
    KeyPair<DCRTPoly> keys = cc.cryptoContext->KeyGen();
    cc.pks.push_back(keys.publicKey);
    cc.sks.push_back(keys.secretKey);
    cc.lastKey = 0;
    return cc;
}

/**
 * @brief generate ck encapsulating sk_k for Pk
 *
 * @param sk secret key sk1
 * @param cc cryptographical context
 * @return thresholdTools
 */
thresholdTools init(PrivateKey<DCRTPoly> sk, CryptoContext<DCRTPoly> cc) {
    thresholdTools tt;

    // Generate c11 encapsulating sk1 for P1
    tt.AddedKey = cc->KeySwitchGen(sk, sk);
    return tt;
}

/**
 * @brief generate pk* derived from a previous threshold pk' and the pk_k for player Pk (i.
     e. pk* = pk_k + pk')
 *
 * Observation: If every player Pk calls this function, the last player would obtain pk*
 *
 * @param cc cryptographical context
 * @return cryptoTools
 */
cryptoTools newMultiPartyKey(cryptoTools cc) {
    // MultiPartyKeyGen generates a key pair (sk_k, pk*) where pk* is the shared public key
         for k players
    KeyPair<DCRTPoly> keyPair2 = cc.cryptoContext->MultipartyKeyGen(cc.pks[cc.lastKey]);
    // Add pk* at the end of list of public keys in cryptoTools
    cc.pks.push_back(keyPair2.publicKey);
    // Add sk_k at the end of list of secret keys in cryptoTools
    cc.sks.push_back(keyPair2.secretKey);
    // Update last key index
    cc.lastKey += 1;
    return cc;
}

/**
 * @brief compute Ck encapsulating (sk_k + sk*) for player Pk
 *
 * @param previousKey sk* = sk_1 + ... + sk_{k-1}
 * @param sk sk_k
 * @param pk pk*
 * @param cc cryptographical context
 * @return EvalKey<DCRTPoly> ek*
 */
EvalKey<DCRTPoly> updateAddedMultKey(EvalKey<DCRTPoly> previousKey, PrivateKey<DCRTPoly> sk
    , PublicKey<DCRTPoly> pk, CryptoContext<DCRTPoly> cc) {
    // Generate ck encapsulating sk_k for Pk (similar to function init() but considering
        previous evalkey C_{k-1}=c1+...+c{k-1})
    auto newKey = cc->MultiKeySwitchGen(sk, sk, previousKey);
    // Compute Ck = ck + C_{k-1}
    return cc->MultiAddEvalKeys(previousKey, newKey, pk->GetKeyTag());
}

/**
 * @brief compute ^C1 = sk x C + z
 *
 * @param addedKey is C = c1 + ... + cn
 * @param sk is sk_k
 * @param pk is pk*
 * @param cc cryptographical context
 * @return EvalKey<DCRTPoly> is ^C1
 */
EvalKey<DCRTPoly> initFinalMultKey(EvalKey<DCRTPoly> addedKey, PrivateKey<DCRTPoly> sk,
    PublicKey<DCRTPoly> pk, CryptoContext<DCRTPoly> cc) {
    // compute ^C1
    return cc->MultiMultEvalKey(sk, addedKey, pk->GetKeyTag());
}

/**
 * @brief compute ^Ck = (sk x C + z) + ^C
 *
 * @param addedKey is C = c1 + ... + cn
 * @param previousKey is ^C = ^C1 + ^C{k-1}
 * @param sk is sk_k
 * @param pk is pk*
 * @param cc cryptographical context
 * @return EvalKey<DCRTPoly> is ^Ck
 */
```

```
EvalKey<DCRTPoly> updateFinalMultKey(EvalKey<DCRTPoly> addedKey, EvalKey<DCRTPoly>
    previousKey, PrivateKey<DCRTPoly> sk, PublicKey<DCRTPoly> pk, CryptoContext<DCRTPoly>
    cc) {
    // compute c' = (sk x C + z)
    auto newKey = cc->MultiMultEvalKey(sk, addedKey, pk->GetKeyTag());
    // Compute ^Ck = c' + ^C
    return cc->MultiAddEvalMultKeys(newKey, previousKey, newKey->GetKeyTag());
}

/**
 * @brief set ^C = ^C1 + ... + ^Cn as the evalKey in cryptographical context
 *
 * @param finalMultKey is ^C
 * @param cc cryptographical context
 * @return CryptoContext<DCRTPoly> the new cryptographical context has evalKey inserted
 */
CryptoContext<DCRTPoly> setFinalMultKey(EvalKey<DCRTPoly> finalMultKey, CryptoContext<
    DCRTPoly> cc) {
    cc->InsertEvalMultKey({finalMultKey});
    return cc;
}

/**
 * @brief encrypt integer using threshold cryptographical context
 *
 * @param n integer to be encrypted
 * @param pk threshold public key
 * @param cc cryptographical context
 * @return Ciphertext<DCRTPoly> ciphertext encrypting n
 */
Ciphertext<DCRTPoly> encryptThresholdBGV(int n, PublicKey<DCRTPoly> pk, CryptoContext<
    DCRTPoly> cc) {
    // Generate vector with the integer
    std::vector<int64_t> vectorOfInts = {n};

    // Encode vector as plaintext
    Plaintext plaintext                = cc->MakePackedPlaintext(vectorOfInts);

    // Encrypt plaintext
    Ciphertext<DCRTPoly> ciphertext    = cc->Encrypt(pk, plaintext);
    return ciphertext;
}

/**
 * @brief compute partial decryption of ciphertext for player P1 using sk_1
 *
 * @param c ciphertext to be decrypted
 * @param sk partial secret key
 * @param cc cryptographical context
 * @return std::vector<Ciphertext<DCRTPoly>> containing only one element: this partial
     decryption
 */
std::vector<Ciphertext<DCRTPoly>> partialDecryptBGVLead(Ciphertext<DCRTPoly> c, PrivateKey<
    DCRTPoly> sk, CryptoContext<DCRTPoly> cc) {
    // Initialize plaintext for result
    Plaintext partialPlaintextResult;
    // compute w1, the partial decryption of c using sk1
    auto ciphertextPartial = cc->MultipartyDecryptLead({c}, sk);
    // initialize array to store future partial decryptions
    std::vector<Ciphertext<DCRTPoly>> partialCiphertextVec;
    // add w1 to the previous array
    partialCiphertextVec.push_back(ciphertextPartial[0]);
    return partialCiphertextVec;
}

/**
 * @brief compute partial decryption of ciphertext for player Pk using sk_k
 *
 * @param c ciphertext to be decrypted
 * @param sk is sk_k
 * @param cc cryptographical context
 * @param partialCiphertextVec vector containing previous partial decryptions {w1, ..., w{k
     -1}}
 * @return std::vector<Ciphertext<DCRTPoly>> vector containing {w1, ..., wk}
 */
std::vector<Ciphertext<DCRTPoly>> partialDecryptBGVMain(Ciphertext<DCRTPoly> c, PrivateKey<
    DCRTPoly> sk, CryptoContext<DCRTPoly> cc, std::vector<Ciphertext<DCRTPoly>>
    partialCiphertextVec) {
    // Initialize plaintext for result
    Plaintext partialPlaintextResult;
    // compute wk using sk_k
    auto ciphertextPartial = cc->MultipartyDecryptMain({c}, sk);
    // add wk at the end of partialCiphertextVec
    partialCiphertextVec.push_back(ciphertextPartial[0]);
    return partialCiphertextVec;
}

/**
 * @brief compute final decryption using all partial decryptions
 *
 * @param partialCiphertextVec array containing all partal decryptions {w1, ..., wn}
 * @param cc cryptographical context
 * @return std::vector<int64_t> containing the result
 */
std::vector<int64_t> decryptThresholdBGV(std::vector<Ciphertext<DCRTPoly>>
    partialCiphertextVec, CryptoContext<DCRTPoly> cc) {
    // Initialize plaintext for result
    Plaintext plaintextResult;
    // compute c + w1 + ... + wn to finally decrypt ciphertext
    cc->MultipartyDecryptFusion(partialCiphertextVec, &plaintextResult);
    // decode plaintext to obtain message (as vector of coefficients)
    std::vector<int64_t> result = plaintextResult->GetPackedValue();
    return result;
}
```

Appendix B

Implementation of the equality test in OpenFHE.

```
// Created on February 13 2023
// By Julen Bernabe Rodriguez <julen.bernabe@tecnalia.com>
// Copyright (c) 2023 Tecnalia Research & Innovation

/*
 * Comparaciones en BGV
 */

#include "scheme/bgvrns/cryptocontext-bgvrns.h"
#include "gen-cryptocontext.h"

#include <iostream>
#include <fstream>
#include <limits>
#include <iterator>
#include <random>
#include <time.h>

using namespace lbcrypto;
/**
 * @brief Compute the X and Y sets for f(x) = 1 (if x = 0) and f(x) = 0 otherwise
 *
 * @param p prime number
 * @return interpolationPoints containing the X and Y sets
 */
interpolationPoints evalEqualPoints(int p) {
    interpolationPoints ip;
    // First we add X={0} and Y={1} because f(0)=1
    ip.x.push_back(0);
    ip.fx.push_back(1);
    // All the other elements from 1 to p-1 will be f(x)=0
    for (int i = 1; i < p; i++) {
        if (i <= (p-1)/2) {
            // Add i to X until i=(p-1)/2
            ip.x.push_back(i);
            // Add 0 to Y
            ip.fx.push_back(0);
        } else if (i > (p-1)/2) {
            // Add -i to X if i>(p-1)/2 (we have negative numbers)
            ip.x.push_back(-(p-i));
            // Add 0 to Y
            ip.fx.push_back(0);
        }
    }
    return ip;
}

/**
 * @brief Calculate if a ciphertext c is equal to 0
 *
 * @param c the ciphertext to use as input
 * @param cc the cryptographical context
 * @return Ciphertext<DCRTPoly> containing the result of the evaluation of L(c)
 */
Ciphertext<DCRTPoly> equalZero(Ciphertext<DCRTPoly> c, cryptoTools cc) {
    // Get plaintext modulos p
    int p = cc.cryptoContext->GetCryptoParameters()->GetPlaintextModulus();
    // Compute {c, c^2, ..., c^{p-1}} using Fast Modular Exponentiation library
    std::vector<Ciphertext<DCRTPoly>> cPowers = powers(c, cc);
    // Compute X and Y for the equality
    interpolationPoints ip = evalEqualPoints(p);
    // Get interpolation polynomial using Lagrange's Polynomial Interpolation library
    std::vector<int64_t> poly = getLagrangePoly(ip, p);
    // Encrypt interpolation polynomial using cryptographical context
    std::vector<Ciphertext<DCRTPoly>> cPoly = encryptInterpolator(poly, cc);
    // Evaluate the interpolation polynomial over c using Lagrange's Polynomial
        Interpolation library
    Ciphertext<DCRTPoly> evaluation = evalInterpolator(cPowers, cPoly, cc);
    return evaluation;
}

/**
 * @brief Compute if two ciphertexts c1 and c2 are equal
 *
 * @param c1 first ciphertext
 * @param c2 second ciphertext
 * @param cc cryptographical context
 * @return Ciphertext<DCRTPoly> containing the result
 */
Ciphertext<DCRTPoly> equal(Ciphertext<DCRTPoly> c1, Ciphertext<DCRTPoly> c2, cryptoTools cc
    ) {
    // Compute difference = c1 - c2
    Ciphertext<DCRTPoly> difference = cc.cryptoContext->EvalSub(c1, c2);
    // Call the subroutine to see if a ciphertext is equal to 0 or not
    Ciphertext<DCRTPoly> result = equalZero(difference, cc);

    return result;
}
```

Appendix C

Implementation of the VD protocol using threshold FHE in OpenFHE.

```
/**
 * @ Author: Julen Bernabe Rodriguez <julen.bernabe@tecnalia.com>
 * @ Create Time: 2023-06-15 13:21:06
 * @ Description: Copyright (c) 2023 Tecnalia Research & Innovation
 */

/*
 * Comparaciones en BGV
 */

#include "scheme/bgvrns/cryptocontext-bgvrns.h"
#include "gen-cryptocontext.h"

#include <iostream>
#include <fstream>
#include <limits>
#include <iterator>
#include <random>
#include <time.h>
#include "../lib/lib.cpp"
#include "../lib/threshold/threshold-basics.cpp"
#include "../lib/threshold/threshold-power.cpp"
#include "../lib/threshold/threshold-interpolation.cpp"
#include "../lib/threshold/threshold-compare.cpp"

using namespace lbcrypto;

/**
 * @brief calculate if two ciphertexts are equal using threshold BGV encryption
 *
 */
void threshold_equal() {

    /**
     * ##############################################################################
     *
     *                          PART 1: THRESHOLD KEY GENERATION
     *
     * ##############################################################################
     */

    /**
     * @brief generate threshold cryptographical context
     *
     *  @param p Prime number is p = 257
     *  @param n Ring dimension is n = 2
     *
     * @return cc (cryptographical context) containing:
     *   - Secret key sk1 and public key pk1 for Alice
     *   - Last key index = 1
     */

    cryptoTools cc = genThresholdBGVCryptoTools(257, 2);

    /**
     * @brief initialize threshold protocol to obtain ck
     *
     * @param cc.sks[0] Secret key sk1 from Alice
     * @param cc.cryptocontext Cryptographical context
     *
     * @return tt (threshold tools) containing:
     *   - The first version of Ck = C1 (encapsulating sk1)
     */
    thresholdTools tt = init(cc.sks[0], cc.cryptoContext);


    // Now thresholdTools is sent to Bob


    /**
     * @brief Bob generates her multi-party key (sk2, pk*)
     *
     * @param cc Cryptographical context (only pk1 needed)
     *
     * @return updated cryptographical context with:
     *   - Last public key in publicKeys being pk*
     *   - LastKey = 2
     */
    cc = newMultiPartyKey(cc);


    /**
     * @brief Bob uses multi-party key to update threshold tools
     *
     * @param AddedKey Previously added key C1 encapsulating sk1
     * @param cc.sks[1] Bob's secret key sk2
     * @param cc.pks[1] Bob's public key pk2
     * @param cc.cryptocontext The cryptograhical context
     *
     * @return updated threshold tools with:
     *   - C2 encapsulating (sk1 + sk2)
     */
    tt.AddedKey = updateAddedMultKey(tt.AddedKey, cc.sks[1], cc.pks[1], cc.cryptoContext);

    // Now thresholdTools is sent again to Alice to create EvalKey from C2

    /**
     * @brief Alice creates ^C1 = sk1 x C2 + z1 using threshold tools
     *
     * @param AddedKey C2 encapsulating (sk1 + sk2)
     * @param cc.sks[0] Alice's secret key sk1
     * @param cc.pks[0] Alice's public key (only used for tagging the result)
     * @param cc.cryptocontext Cryptographical context
     *
     * @return ^C1 containing:
     *   - ^C1 = sk1 x C2 + z1
     */
    tt.MultKey = initFinalMultKey(tt.AddedKey, cc.sks[0], cc.pks[0], cc.cryptoContext);

    // Now thresholdTools is sent back to Bob to update EvalKey

    /**
     * @brief Bob creates ^C2 = sk2 x C2 + z2 and then computes ^C = ^C1 + ^C2
```

```
     *
     * @param AddedKey C2 encapsulating sk1 + sk2
     * @param MultKey ^C1 = sk1 x ^C2 + z1
     * @param cc.sks[1] Bob's secret key
     * @param cc.pks[1] Bob's public key (only used for tagging the result)
     * @param cc.cryptoContext Cryptographical context
     *
     * @return  ^C encapsulating (sk1 + sk2)(sk1 + sk2) = (sk1 + sk2)^2
     */
    tt.MultKey = updateFinalMultKey(tt.AddedKey, tt.MultKey, cc.sks[1], cc.pks[1], cc.
        cryptoContext);

    /**
     * @brief set final EvalKey for multiplication
     *
     * @param MultKey containing ^C
     * @param cc.cryptoContext cryptographical context
     *
     * @return updated cryptographical context containing final EvalKey ^C
     */
    cc.cryptoContext = setFinalMultKey(tt.MultKey, cc.cryptoContext);

    /**
     * ###############################################################################
     *
     *              PART 2: THRESHOLD ENCRYPTION & HOMOMORPHIC EVALUATION
     *
     * ###############################################################################
     */

    std::cout << "\nTHRESHOLD BGV NON-LINEAR OPERATIONS\n "<< std::endl;
    int p = cc.cryptoContext->GetCryptoParameters()->GetPlaintextModulus();

    // --------------------------------- ALICE'S SIDE ---------------------------------

    /**
     * @brief Alice is asked to introduce two numbers between 0 and 128
     *
     * @param first the first integer
     * @param second the second integer
     */
    int first, second;
    std::cout << "Enter two integers: "<< std::endl;
    std::cout << "\t - First integer: ";
    std::cin >> first;
    while (first > (p-1)/2) {
        std::cout << "\nInteger must be between " << -(p - 1) / 4 << " and " << (p - 1) / 4
             << std::endl;
        std::cout << "First integer: ";
        std::cin >> first;
    }
    while (first < -(p-1)/2) {
        std::cout << "\nInteger must be between " << -(p - 1) / 4 << " and " << (p - 1) / 4
             << std::endl;
        std::cout << "First integer: ";
        std::cin >> first;
    }
    std::cout << "\t - Second integer: ";
    std::cin >> second;
    while (second > (p-1)/2) {
        std::cout << "\nInteger must be between " << -(p - 1) / 4 << " and " << (p - 1) / 4
             << std::endl;
        std::cout << "Second integer: ";
        std::cin >> second;
    }
    while (second < -(p-1)/2) {
        std::cout << "\nInteger must be between " << -(p - 1) / 4 << " and " << (p - 1) / 4
             << std::endl;
        std::cout << "Second integer: ";
        std::cin >> second;
    }

    /**
     * @brief Alice encrypts both integers using material from threshold key generation
     *
     * @param first the first integer
     * @param second the second integer
     * @param lastKey last key in pks containing pk*
     * @param cryptoContext cryptographical context
     *
     * @return c1 and c2, containing the ciphertexts for both first and second integers,
         respectively
     */
    Ciphertext<DCRTPoly> c1 = encryptThresholdBGV(first, cc.pks[cc.lastKey], cc.
        cryptoContext);
    Ciphertext<DCRTPoly> c2 = encryptThresholdBGV(second, cc.pks[cc.lastKey], cc.
        cryptoContext);

    // --------------------------------- ALICE'S SIDE ---------------------------------

    //                        The ciphertexts are sent to Bob

    // ---------------------------------- BOB'S SIDE ----------------------------------

    time_t timer3;
    time_t timer4;
    double seconds1;
    time(&timer3);
    /**
     * @brief compute the equality test using threshold key material
     *
     * @param c1 ciphertext encapsulating first integer
     * @param c2 ciphertext encapsulating second integer
     * @param cc cryptographical context (contains EvalKey)
     */
    Ciphertext<DCRTPoly> cEq = equal(c1, c2, cc);
    time(&timer4);
    seconds1 = difftime(timer4,timer3);

    // ---------------------------------- BOB'S SIDE ----------------------------------

    //                            The ciphertext is sent to Bob

    // --------------------------------- ALICE'S SIDE ---------------------------------

    /**
     * @brief compute partial decryption from Alice's side
     *
     * @param cEq ciphertext containing the result of the equality test
     * @param cc.sks[0] Alice's secret key sk1
     * @param cc.cryptoContext cryptographical context
     *
     * @return partialResultsEq containing the partial decryption from Alice
     */
    std::vector<Ciphertext<DCRTPoly>> partialResultsEq = partialDecryptBGVLead(cEq, cc.sks
        [0], cc.cryptoContext);

    // --------------------------------- ALICE'S SIDE ---------------------------------

    //                            The partial result is sent to Bob

    // ---------------------------------- BOB'S SIDE ----------------------------------

    /**
     * @brief compute partial decryption from Bob's side
     *
     * @param cEq ciphertext containing the result of the equality test
     * @param cc.sks[1] Bob's secret key sk2
     * @param cc.cryptoContext cryptographical context
     * @param partialResultsEq previous partial decryption from Alice
     *
     * @return partialResultsEq containing the partial decryptions from Alice and Bob
     */
    partialResultsEq = partialDecryptBGVMain(cEq, cc.sks[1], cc.cryptoContext,
        partialResultsEq);

    /**
     * @brief compute final decryption
     *
     * @param partialResultsEq previous partial decryptions from Alice and Bob
     * @param cc.cryptoContext cryptographical context
     *
     * @return rEq containing the complete decryption from Bob
     */
    std::vector<int64_t> rEq = decryptThresholdBGV(partialResultsEq, cc.cryptoContext);
    std::cout << first << " == " << second << ": " << rEq[0] << std::endl;
    std::cout << "\nTime used to compare: " << seconds1 << " seconds "<< std::endl;
}

int main() {
    threshold_equal();
}
```