POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

A Fully Homomorphic Encryption Application: SHA256 on Encrypted Input

Supervisor
Prof. Danilo Bazzanella

Candidate
Paolo Tassoni

Company Supervisors
Dr. Marco Rinaudo
Dr. Veronica Cristiano

Accademic Year 2022/2023

Contents

Acknowledgements

This thesis work was carried out at Telsy, the TIM group's competence center for communication security and cybersecurity.
I would like to thank my supervisor, professor Danilo Bazzanella, for giving me the opportunity to work on this thesis and for supporting me in this project.
I am sincerely grateful to the entire Telsy Cryptography research group, especially to my supervisors Marco Rinaudo and Veronica Cristiano whose professionalism and extreme availability were essential to the success of this work. Their guidance has been truly appreciated.

Summary

Nowadays, data privacy plays a key role in the context of the Cloud services, Artificial Intelligence, Internet of Things and other applications. Among all the different approaches in the field of information security and cryptography for preserving the privacy and the secrecy of data, one of the most promising is Fully Homomorphic Encryption (FHE). In fact, FHE enables users to perform computations directly on encrypted data without having to first decrypt it, ensuring confidentiality and preventing the exposure of sensitive information.
This thesis presents a use case application for FHE, more specifically, a homomorphic implementation of the currently most used hash function, SHA256.
The initial part of this thesis is focused on the study of FHE, initially going over some basic fundamentals of cryptography, and then introducing various Homomorphic Encryption schemes, culminating in the FHE scheme called TFHE (Fully Homomorphic Encryption over the Torus).
In the second part of this thesis we present our implementation of SHA256 that operates homomorphically on encrypted input. We then integrate this work in the context of a client-server architecture where the server can compute the hash function without knowing the input giving by the client. We develop this application using the ZAMA Concrete compiler based on the TFHE scheme.
Given the continuous progress in the development of FHE applications, we believe that a homomorphic version of SHA256 might be extremely helpful as a foundation for future complex applications, aiming to increase users' privacy.

Chapter 1

Cryptographic Preliminaries

This chapter aims to provide a basic understanding of cryptography, clarifying the fundamental concepts and terminologies that will be the basis for subsequent chapters.
In addition to the more general concepts regarding cryptography, we will delve into what **symmetric** 1.2 and **asymmetric systems** 1.3 are, by understanding their differences and the different purposes that they can accomplish.
In the last part we will discuss **hash functions** 1.4, giving an overview of existing functions of this type and focusing on the currently most important ones, by comparing their pros and cons.

1.1 Terminology

To introduce the concept of cryptography, we can refer to the following definitions:

"***Cryptography*** *is the discipline that embodies the principles, means, and methods for the providing information security, including confidentiality, data integrity, non-repudiation, and authenticity*" [34].
"*Cryptography is the art and science of keeping messages secure, and it is practiced by* ***cryptographers****. It differs from* ***cryptanalysis****, which is the art and science of breaking a cipher code. The study of secure and secret communication, which includes both cryptography and cryptanalysis is called* ***cryptology***" [46].

A cryptographic system is based on the following elements and operations:

- **Plaintext**: "*Intelligible data that has meaning and can be understood without application of decryption*" [2].
  Generally it corresponds to the message whose confidentiality you want to protect.
- **Ciphertext**: Encrypted text transformed from plaintext.

- **Encryption**: Operation which is the transition from plaintext P to ciphertext C using an encryption algorithm Enc and a key K.
- **Decryption**: Operation which is the reverse transition from ciphertext C to plaintext P using a decryption algorithm Dec and a key K', which may be different from that used previously.
- **Key**: String of bytes which when processed through a cryptographic algorithm allows to randomize a plaintext in order to obtain a ciphertext.

In cryptography, a **cryptosystem** is a "*suite of cryptographic algorithms needed to implement a particular security service, such as confidentiality*" [37].
Tipically it consists of three algorithms: Key Generation, Encryption and Decryption.

There is an important principle in cryptography, called *Kerckhoffs's principle* [26], that states that security of a cryptosystem is based on a strong secret key and its secrecy rather than the secrecy of the algorithm itself. In other word a cryptosystem should remain secure even if all the details of the encryption and decryption algorithms are publicly known, as long as the key remains secret.
This principle is the opposite of the concept of "security through obscurity", that is based on the total secrecy of the algorithms. Kerckhoffs's principle had a profound influence on the design of cryptosystems, promoting the development of algorithms that prioritize the secrecy and the strength of the key over the secrecy of the algorithm itself.

1.2 Symmetric Cryptography

In a **Symmetric** (or Private/Secret Key) **Cryptosystem** the same key K is used for both encryption and decryption.

Example Let assume that a sender wants to exchange data with a receiver.
The process begins with a secret key K shared only between the sender and the receiver. The key K is used to encrypt, through Enc, the plaintext P and then the result (Ciphertext C) is an unintellegible text that is sent to the receiver.

$$C = Enc(K, P)$$

When the encrypted data is delivered to the receiver, it can be decrypted, through Dec, using the same key K.

$$P = Dec(K, C)$$

If a different key is used, an output may be available, but it will not coincide with the original plaintext.

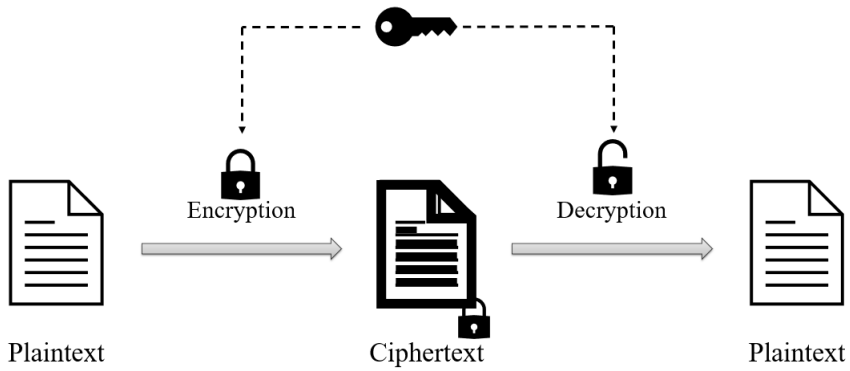

Figure 1.1. Symmetric Criptography

1.2.1 Algorithms

The most used symmetric algorithm is the *Advanced Encryption Standard* (AES) [33], it is a variant of the Rijndael cipher with three different possible key lengths (128, 192 and 256 bit). Other important and historical symmetric key algorithms to mention are the *Data Encryption Standard* (DES) [15] developed in the early 1970s, and the *Triple DES* (3DES) [31] which is a variant that applies the previous one three times to plaintext, but nowadays they are both deprecated by the *National Institute of Standards and Technology* (NIST). Another popular cryptosystem was the *Rivest Cipher 4* (RC4) [41], that was known for its simplicity and speed in software, but that is now deprecated.

1.2.2 Advantages and limitations

One of the main advantages of symmetric cryptography is its efficiency. The same key is used for both encryption and decryption, the process is relatively fast and requires fewer computational resources.
The disadvantage of symmetric cryptography is that it assumes two parties have agreed on a key and have been able to securely exchange that key over a insecure channel prior to communication.
To achieve a balance of security and speed, symmetric algorithms are frequently combined with another tecnique called **public key cryptography**.

1.3 Public Key Cryptography

In a **Public Key** (or Asymmetric) **Cryptosystem** each user has a key pair, that consists of a sK (Secret Key) and a pK (Public Key) that are mathematically related to each other.

- The **Secret Key** sK is the key that must be remain secret and it is known only to the person who generated it, the security of this cryptosystem is based on the key's secrecy.
- The **Public Key** pK is the key that can be publicly distributed without compromising security [48]

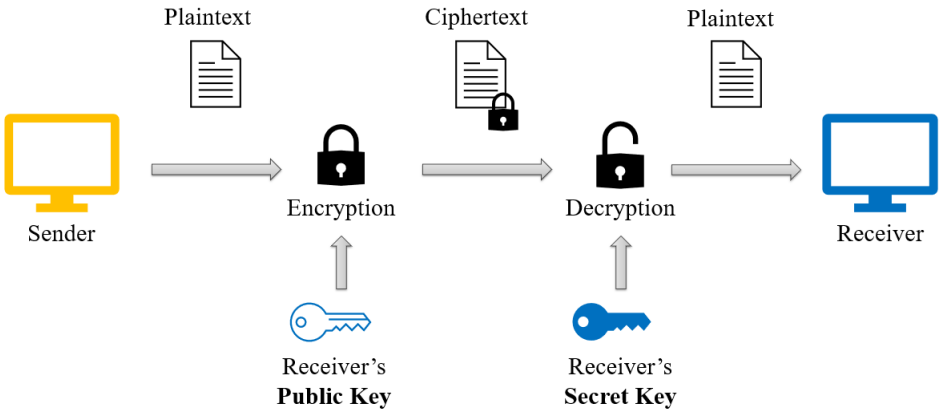

Figure 1.2. Public Key Criptography

Example If in a communication channel the sender wants to send a secret message to the receiver, the sender will encrypt the message with the receiver's public key (pK distributed previously by the receiver) and at this point only the receiver will be able to decrypt the encrypted message with his secret key (sK) as shown in figure 1.2.

1.3.1 Characteristics

The security of public key encryption algorithms is based on one-way functions: these functions f are easy to compute but difficult to invert: given x it is easy to compute $y = f(x)$, but given $y = f(x)$, it is difficult to retrieve x.
The security of these algorithms is guaranteed by the fact that no one, as far as we know, has yet solved the underlying mathematical problems: this also means that if someone were to solve them, these algorithms would suddenly become insecure.

One possible threat to the security of these algorithms is the advent of quantum computers, whose high computational power could lead to the resolution of these mathematical problems [6].
As the scientific community disseminates and circulates information about its findings, this is seen as a guarantee that if one day these problems are solved the fact will immediately be in the public domain [5]. One of the most important algorithm of public key cryptography is *Rivest Shamir Adleman* (RSA) [44], first described in 1978, where the security relies on the practical difficulty of factoring the product of two large prime numbers, the so-called "integer factoring problem".

This kind of cryptography has a high computational load, so it is usually used to **distribute symmetric secret keys** and to create **digital signatures**, it is not used for encrypting large amount of data.

1.4 Hash Functions

Definition. *Given* Σ *an alphabet and* Σ^* *the set of all words (of arbitrary length) obtainable from* Σ*, and* n *an integer, we define a* ***hash function*** *as* $\boldsymbol{H} : \Sigma^* \to \Sigma^n$.

A hash function is a map from an arbitrary binary string to a binary string with a fixed size of n bits, where typically n =128, 256, 384 or 512.
The value $h = H(m)$ is also called **hash value**, **hash code** or **digest**.
This hash value h is usually regarded as fingerprint of the input m.

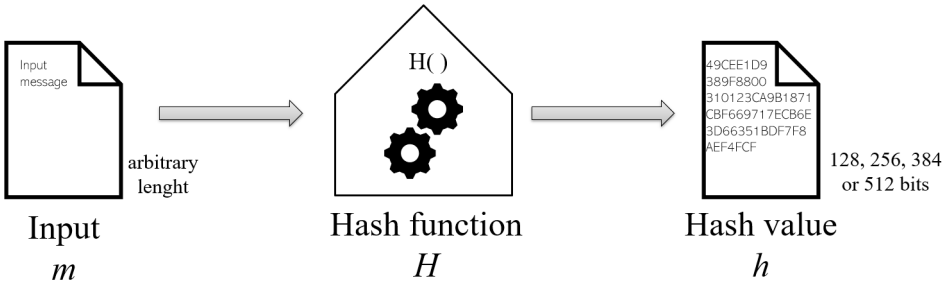

Figure 1.3. Hash Function

The use of hash functions in cryptography is widespread: they are used in digital signatures, public-key encryption, integrity verification, message authentication, password protection, key agreement protocols, and many other cryptographic protocols [1].

1.4.1 Properties

The security level of a cryptographic hash function has been defined using the following properties [37]:

- **One-way**: Given a hash value h and a hash function H, it should be difficult to find any message m such that $h = H(m)$ (*Pre-image Resistance*)
- **Collision Resistance**: It should be difficult to find two different messages m_1 and m_2 such that $H(m_1) = H(m_2)$.
- **Second Pre-image Resistance**: Given an input m_1, it should be difficult to find a different input m_2 such that $H(m_1) = H(m_2)$.

1.4.2 Hash Algorithms

Among the most widely used and famous cryptographic hash functions there is *MD5* [43], devised by Ronald Rivest in 1991, successor to the earlier and less performant MD4 (1990) [42]. As a European alternative to MD4 and MD5, in the 1994 was devised *RIPEMD* (Hans Dobbertin, Antoon Bosselaers and Bart Preneel) [40]. There are several versions, each with different hash lengths. The most used is the one called RIPEMD-160. Over the years, collisions of the original version of RIPEMD have been found, but to date there is no evidence that RIPEMD-160 has ever been broken [3].

Some of the most used hash functions are those in the family of *Secure Hash Algorithm* (SHA) which includes several cryptographic hash functions developed beginning in 1993 and published by NIST as a federal standard by the U.S. government.
The SHA family includes [35]:

- **SHA-1**: A 160-bit hash function, developed by the *National Security Agency* (NSA), which resembles the earlier *MD5* algorithm. A cryptographic weakness was discovered in SHA-1 [30] and its use is now deprecated.
- **SHA-2**: The 4 algorithms that are referred to generically as SHA-2 are *SHA-224*, *SHA-256*, *SHA-384* and *SHA-512*, they produce a digest of bit length equal to the number indicated in their abbreviation.
- **SHA-3** [36]: A hash function based on the *Keccak* permutation that supports the same hash lengths as SHA-2, but its internal structure differs significantly from the rest of the SHA family.

SHA-3 has the advantanges to be cheap to implement in specialized hardware and to have excellent performance on dedicated circuits, but at the same time it is slower than SHA-2 on general-purpose processors [27].
The adoption of SHA-3 is currently proceeding slowly given its very recent introduction and the difficulty of the integration process into existing systems and protocols, also due to the costs associated with the transition from SHA-2.

In contrast, SHA-2 has been widely adopted because of its well-established optimizations, and its software implementations generally demonstrate superior performance, making it an efficient choice for various applications.
Furthermore, the prevalence of SHA-2 is also due to its extensive integration into hardware components like CPUs, GPUs, and cryptographic modules.
Its long-standing presence in cryptographic protocols and applications has solidified its role as the de facto standard hash function in various domains.
The ample diffusion of SHA2, together with the confidence in its security, are the reasons why we chose to implement a homomorphic version of this hash function, as we will show in chapter 4.

Chapter 2

Homomorphic Encryption

In this chapter we will introduce the concept of **Homomorphic Encryption**, starting with an introduction 2.1 where we will explain why it is of great interest nowadays by bringing some real-life use cases as examples, then we will move on to a more in-depth description.
In particular, in section 2.2, we will provide a **background** to better understand Homomorphic Encryption (HE), introducing the core concepts (2.2.1) and an overview of different families of the existing HE schemes (2.2.2), including their origins and historical development (2.2.3).
In section 2.3 we will focus on a specific family of HE, called **Fully Homomorphic Encryption** (FHE), introducing a formal definition of it (2.3.1) and explaining the notion of **bootstrapping** (2.3.2). Later, we will provide an overview of different **FHE schemes** that have been proposed and developed (2.3.4) over the last few years, after understanding the general distinctions between them (2.3.3), showing the diversity of approaches in achieving FHE.
In the end, in section 2.4, we will examine a precise FHE scheme called **TFHE**, on which the library of the main project of this thesis is based. Specifically, we will cover the types of TFHE ciphertexts (2.4.1) and the operations (2.4.2) that can be performed on them, including the description of the **programmable bootstrapping** procedure (2.4.3).

2.1 Introduction

Since data exchange, manipulation and storage are essential to many aspects of modern technology-driven society, this leads to an increasing attention to the security and privacy aspects of data as more and more sensitive and private data are involved. The importance of data cannot be underestimated, the information power drives and improves service quality across all sectors of society.
However, this abundance of data requires safeguards to protect individuals and

organizations from breaches, unauthorized access, and improper uses. The proliferation of data has increased need for solutions to provide access to services that use personal data while protecting its privacy, for this reason information security is focused on developing new strategies for ensuring data confidentiality and efficiently implementing them [47]. Encryption is often an essential safeguard for the privacy of sensitive data, but in order to process data (e.g. apply a Machine Learning model, compute statistics) it is necessary to decrypt it, thus exposing it to external attackers or malicious service providers.
Among all the different approaches in the field of information security and cryptography trying to preserve the privacy and the secrecy of data, one of the most promising is **Homomorphic Encryption** (HE). In fact, it enables users to perform computations directly on encrypted data without having to first decrypt it, ensuring the property of data confidentiality and preventing the exposure of sensitive information.

Real-world applications of HE are in continuos development and several implementations of possible solutions to the challenges encountered are already available in various domains: Databases, Healthcare Research, Cloud Computing, Machine Learning, IoT Computing, Blockchain etc.
The most general use case of HE involves *databases management*, that is the starting point for the other applications. Companies, governments, and organizations store vast amounts of sensitive data in databases, ranging from personal identifiers to financial records. In this context a potential misuse of data managed by third parties is cause of concern, especially when confidentiality agreements and data privacy regulations are violated without the user's permission.
In this context HE would enable the management of data in the database and all subsequent operations to be performed on it so as to ensure the protection of sensitive data and a significantly increase of data confidentiality, privacy and integrity preventing unauthorized access to the stored data.

For example, we can consider the *healthcare* sector as a good field of application for HE where healthcare providers could perform complex data analysis on patient records without ever exposing sensitive medical information. HE allows computations on encrypted health data, enabling medical professionals to collaborate securely while preserving patient privacy. This technology can lead to the development of predictive healthcare models, enhancing diagnosis accuracy and the overall quality of care.
Additionally, in the *finance* sector, where secure data processing is critical. HE can enable banks and financial institutions to conduct risk assessments, fraud detection, and other financial security processes, while keeping client financial information confidential.

We can also mention the technology's field of *Cloud Computing*, which refers to the practice of accessing IT resources via the Internet paying only for what is used, where services are provided by cloud providers such as Amazon Web Services, Google Cloud Provider, Azure, etc. Since cloud providers are unreliable third parties, it is necessary to keep cloud data encrypted to ensure the privacy of user data while allowing cloud services to be able to perform operations on it.
HE technology facilitates the management of ciphertext data while maintaining privacy. It can retrieve and manipulate ciphertext in the cloud and provide the results to users as ciphertext without exposing the processed data to the cloud provider. Compared to methods that decrypt data for manipulating them, an ideal HE algorithm can reduce the cost of communication and computation by eliminating the need for frequent encryption and decryption between the cloud and users.

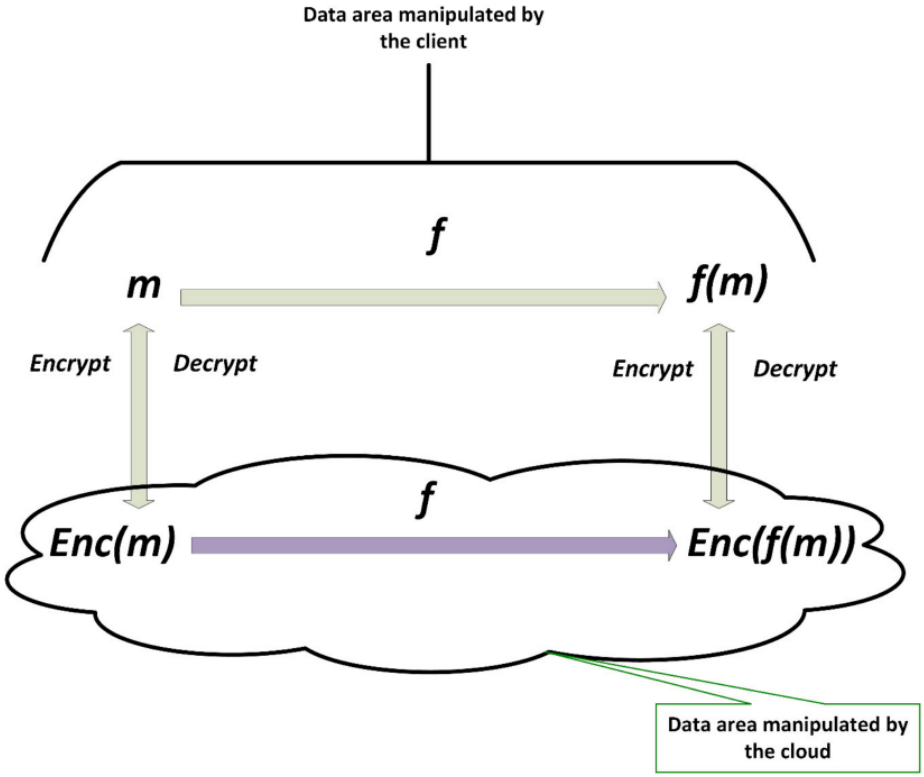

Figure 2.1. HE used in Cloud Computing [47]

Another field of technology where HE can be of great use is *Machine Learning*, an area of research of Artificial Intelligence concerned with the development and study

of statistical methods and algorithms capable of generalizing and performing tasks without explicit instructions. Machine Learning models try to create predictions and decisions based on identified patterns and previous experiences which derive from analyzing massive volumes of data. It is used in various technologies such as facial recognition software and language translation tools.
The training of the machine learning models depend on a large amount of data, which frequently contains sensitive or personal information. The storage and usage of this type of data must be done in secure manner to guarantee an high grade of confidentiality and to prevent potential damage to the users.
In this context, HE allow the computation for all necessary analysis to be done directly on the encrypted data.

Leading companies and research institutions have embraced the potential of HE, utilizing this technology to address challenges related to data security and privacy. Below, we will present some examples of companies and projects making use of Homomorphic Encryption.
Microsoft is one of the tech giants that has heavily invested in Homomorphic Encryption through the *Simple Encrypted Arithmetic Library* (SEAL) project [32]. SEAL is a open-source library that provides a set of homomorphic encryption libraries. This enables software engineers to build end-to-end encrypted data storage and computation services where the customer never needs to share their key with the service.
IBM is another big tech company that has developed its set of products for Homomorphic Encryption with the service called "Security Homomorphic Encryption Services", contributing to the practical adoption of this technology. An example of products released are the Toolkit for database in 2020 [23] and the most recent service HE4Cloud Beta Version [22] in 2022 for integrating the HE in Cloud Computing. These IBM's products are based on HELib [24], a free library and open-source platform software written in C++ that implements various forms of HE developed by IBM.
For the last example we present ZAMA that is a cryptography company building open source homomorphic encryption solutions for blockchain and AI [52]. The main product of ZAMA company is a compiler called Concrete that simplifies the use of HE for developers and permits programming functionalities through python API and provides many other features in writing HE programs.
In the development of the main project of this thesis we chose ZAMA Concrete compiler. We will discuss about it in chapter 3.

2.2 Background

2.2.1 Homomorphic Encryption

The term **Homomorphism** is used in different areas but its etymology comes from ancient Greek from the words "$o\mu o\sigma$" and "$\mu o\varrho\varphi\eta$", meaning respectively "same" and "shape". In abstract algebra a homomorphism is a structure-preserving map between two algebraic structures of the same type (i.e. two groups, two rings ecc.), where a map is simply a function, where elements of one set (inputs) are transformed into elements of a second set (outputs) while maintaining the relationships between the elements.

In the field of cryptography, with the term **Homomorphic Encryption** (HE) we mean a type of encryption technique that enables a third party, such as a cloud provider or service provider, to carry out certain computable operations on the encrypted data while maintaining the features and the structure of the encrypted data.

Example Now an example of HE scheme over a operation of addition; given two elements m_1 and m_2, and Enc the encryption operation, the following expression turns out to be valid:

$$Enc(m_1) + Enc(m_2) = Enc(m_1 + m_2)$$

In this case the additive operation can be applied *directly to the encrypted elements* without first decrypting them, obtaining the same result.

More formally, we can define a homomorphic encryption scheme as follows [45]:

Definition. *Let $\mathcal{M}$ be the set of plaintexts, $\mathcal{C}$ the set of ciphertexts and the two operations*

$$* : \mathcal{M} \times \mathcal{M} \to \mathcal{M} \quad \textit{and} \quad \bullet : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

*An encryption scheme $E : \mathcal{M} \to \mathcal{C}$ is called **homomorphic** with respect to "$*$" and "$\bullet$" if it holds:*

$$E(m_1) \bullet E(m_2) = E(m_1 * m_2)\ \forall m_1, m_2 \in \mathcal{M}$$

Because addition and multiplication are functionally complete sets over finite sets, it is sufficient to merely include these operations in an encryption system in order to enable the homomorphic evaluation of any function. In particular, only *XOR* (addition) and *AND* (multiplication) gates are needed to express any **boolean circuit**, which is a mathematical model based on combinations of logic gates (corresponding to boolean functions) to implement a certain function.

A HE scheme can be created to use different keys for encryption and decryption (asymmetric) or it can also be built to utilize the same key for both operations (symmetric).

2.2.2 Families of HE schemes

An HE scheme is defined by four main operations: **Key Generation** (*KeyGen*), **Encryption** (*Enc*), **Decryption** (*Dec*) and **Evaluation** (*Eval*).

- **KeyGen**: Operation that generates the single key for the symmetric version of a scheme or the key pairs for the asymmetric version.
- **Enc**: Operation of Encryption using the key generated by *KeyGen*.
- **Dec**: Operation of Decryption using the key generated by *KeyGen*.
- **Eval**: Specific operation for the HE scheme that represent the *Homomorphic Property*. It takes the ciphertexts as input and return as output the evaluated ciphertexts, it evaluates a function over the ciphertexts without knowing the plaintexts.

The different HE schemes devised over the years can be categorized according to their main operations, how they execute homomorphic encryption and the related limitations in three families called *Partially Homomorphic Encryption* (**PHE**), *Somewhat Homomorphic Encryption* (**SWHE**) and *Fully Homomorphic Encryption* (**FHE**).

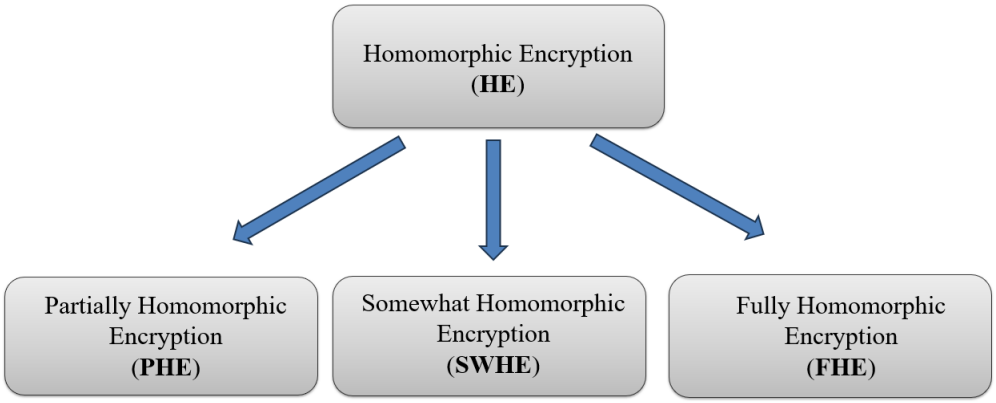

Figure 2.2. HE Families

In the following parts of this section each of the HE families is explored in order to better understand their main properties and characteristics.

Partially Homomorphic Encryption

The schemes that are part of the ***Partially Homomorphic Encryption*** **(PHE)** family allow only one type of operation with no bound on the number of usages. In particular PHE schemes support *Eval* function for only either addition or multiplication.

RSA as PHE One early example of PHE scheme is the public key cryptosystem **RSA** [44], that is *only homomorphic over multiplication*.

- *KeyGen*: For large primes numbers p and q, $n = pq$ and $\phi = (p-1)(q-1)$ are computed. Then e is chosen such that $gcd(e, \phi) = 1$ and d is calculated such that $ed \equiv 1 \mod \phi$. At this point we have the secret key and the public key of the key pair respectively $sK = (d, n)$ and $pK = (e, n)$.
- *Enc*: Given a message $0 \leq m < n$, compute ciphertext C as:

$$C = Enc(m) = m^e \mod n$$

- *Dec*: Using the $sK = (d, n)$ the message m can be recovered from C as follows:

$$m = Dec(C) = C^d \mod n$$

- *Eval*: The Homomorphic property is verified as follows:

$$\begin{aligned} &Enc(m_1) \cdot Enc(m_2) = \\ &= ((m_1)^e \mod n) \cdot ((m_2)^e \mod n) = \\ &= (m_1 \cdot m_2)^e \mod n = Enc(m_1 \cdot m_2) \end{aligned}$$

It shows that $Enc(m_1 \cdot m_2)$ can be directly evaluated by using $Enc(m_1)$ and $Enc(m_2)$ without decrypting them, hence *RSA* results homomorphic over multiplication but it does not allow homomorphic addition of ciphertexts.
In fact, given $Enc(m_1) + Enc(m_2) = ((m_1)^e \mod n) + ((m_2)^e \mod n)$ and $Enc(m_1 + m_2) = ((m_1 + m_2)^e \mod n)$ the homomorphic property is not verified:

$$((m_1)^e \mod n) + ((m_2)^e \mod n) \neq (m_1 + m_2)^e \mod n$$

Somewhat Homomorphic Encryption

The ***Somewhat Homomorphic Encryption*** **(SWHE)** schemes allow some types of operation but with a limited number of times.

BGN as SWHE In 2005 Boneh-Goh-Nissim introduced **BGN** scheme [4], that supports a *limitless number of additions and one multiplication.*

- *KeyGen*: Choose two prime numbers p_1 and p_2 and output (n, G, G_1, e, g, h) where: $n = p_1 p_2$, G, G_1 cyclic groups of order n, g generator of G, $e : G \times G \to G_1$ a bilinear map such that $e(g, g)$ is a generator of G_1 and $h = u^{p_2}$ with $u \neq g$ is another generator of G. Consider

$$sK = p_1$$
$$pK = (n, G, G_1, e, g, h)$$

- *Enc*: Given a message $0 \leq m < p_2$ and with $r \in \{0, ..., n-1\}$ random, compute the ciphertext C as

$$C = Enc(m) = g^m h^r \in G$$

- *Dec*: Given a ciphertext C recover m by computing

$$c' = C^{p_1} \text{ and } g' = g^{p_1}$$

  and solving

$$m = log_{g'}(c')$$

- *Eval*:

  1. Homomorphic property over **addition**: take $r \in \mathbb{Z}_n$ random

     $$E(m_1)E(m_2)h^r = (g^{m_1}h^{r_1})(g^{m_2}h^{r_2})h^r = g^{m_1+m_2}h^{r'} = E(m_1 + m_2) \in G$$

     where $r' = r + r_1 + r_2$.
  2. Homomorphic property over **multiplication**: compute $g_1 = e(g, g)$ and $h_1 = e(g, h)$, take $r \in \mathbb{Z}_n$ random

     $$e(E(m_1)E(m_2))h_1^r = e(g^{m_1}h^{r_1}, g^{m_2}h^{r_2})h_1^r = g^{m_1 m_2}h_1^{r'} = E(m_1 \cdot m_2) \in G_1$$

     where $r' = m_1 r_2 + m_2 r_1 + r + \alpha p_2 r_1 r_2$ with $\alpha \in \mathbb{Z}$ such that $g^{\alpha p_2} = h$.

Note that after the multiplication the resulting ciphertext C is in G_1 instead of G, it still allows an unlimited number of homomorphic additions but it does not allow another homomorphic multiplication in G_1 because there is no pairing from the set G_1.

Fully Homomorphic Encryption

Fully Homomorphic Encryption **(FHE)** is a form of encryption that allows computation (with no bound on the type of operations and number of times that is possible to use) to be performed on encrypted data without the need to decrypt it first.

In 2009 Craig Gentry proposed a scheme [18] that includes both an FHE method and a broad framework for obtaining one. As a result, several researcher have tried to build a safe and useful FHE scheme on top of Gentry's work. Even though Gentry's FHE scheme was very promising, it also had a number of disadvantages, including its computational cost and some sophisticated mathematical principles that make it complex and difficult to implement, therefore new schemes and optimizations have emerged in response to his work [14].

In the next section (2.3) a formal definition of FHE will be given and the fundamental high-level concepts will be explained in details to then focus on the development and programming part.
Before all this, we will provide a brief historical overview of how the concept of HE evolved over time until Gentry's solution, citing some of the best-known schemes proposed during those years.

2.2.3 History

In the late 1970s, precisely in the 1978, the journey of HE began with Ron Rivest, Len Adleman (the "R" and the "A" of *RSA* cryptosystem) and Micheal Dertouzos when the term *homomorphism* was used for the first time in the article "*On data banks and privacy homomorphisms*" [18]. This gave rise to the idea of FHE, which they initially called "*privacy homomorphism*". They state in their paper: "*although there are some truly inherent limitations on what can be accomplished, we shall see that it appears likely that there exist encryption functions which permit encrypted data to be operated on without preliminary decryption of the operands, for many sets of interesting operations. These special encryption functions we call "privacy homomorphisms"; they form an interesting subset of arbitrary encryption schemes*" [45].

Several attempts were made in the years that followed where only a single kind of operation or small number of operations have been made possible on the encrypted data and additionally, some of the approches were particularly restricted to a certain kind of set. We can for example mention Goldwasser and Micali in 1982 [20], El-Gamal in 1985 [16] and Pailler in 1999 [38].

This various attempts led to a categorization of the HE schemes into the three families that we discussed earlier in section 2.2.2, where we analyzed the RSA (1978) [44] and BGN (2005) [4] schemes more in detail. Later, 2009 saw a turning point for FHE thanks to Gentry [18].
In figure 2.3 is shown the timeline of the best-known HE schemes up to Gentry's proposal.

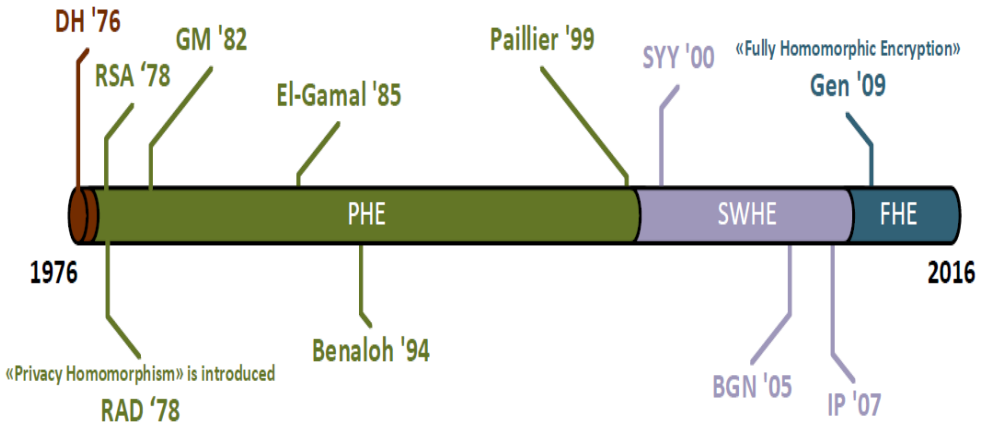

Figure 2.3. Timeline HE [14]

Craig Gentry in his PhD thesis [18] solved the problem of how to do FHE [18], proposing a method that could (inefficiently) do any amount of additions and multiplications. This was made possibile by the introduction of a new fundamental concept (that we will discuss in detail in section 2.3.2) called **Bootstrapping**, which is the intermediate refreshing method of a processed ciphertext. Gentry's proposal is the first practicable FHE scheme: it is based on mathematical objects called ideal-lattices and served as both a description of the scheme and a foundation for the development of subsequent FHE schemes.

However, it is a scheme with obvious limitations because it is particularly expensive in terms of computing in the bootstrapping step. As a result, numerous improvements were put forth and new schemes were proposed in the years that followed.

To conclude this section, we present a famous metaphor in order to better understand the concept of HE.

Gentry's Example: Alice's Jewelry Store

In the article "*Computing Arbitrary Functions of Encrypted Data*" [18] Craig Gentry traces the various stages of the study of homomorphic encryption until he arrives at the innovative concept of Bootstrapping he proposed. To provide a greater understanding of the subject he places alongside the purely technical concepts a metaphor that has been referred to as "*Alice's Jewelry Store*". In the following we present the Gentry's metaphor and we highlight how it is linked to the concept first introduced by American mathematician.

Story Alice is the owner of a jewelry store. She has unrefined expensive materials like gold, diamonds, and silver that she wants her employees to make into intricate rings or necklace. She, on the other hand, is suspicious of her employees and believes that if given the opportunity, they will steal her valuable gems. In essence, she wants her staff to be able to transform the resources into finished products without having direct access to the original materials. Alice adopts a impenetrable glovebox strengthened with a lock that she controls entirely. She places the unprocessed valuable resources inside the secure glovebox, shuts it, and entrusts it to a worker. The worker only assembles the ring or necklace inside the glovebox while wearing protective gloves. The worker recognizes that it is preferable to return the glovebox to Alice with the finished item contained within because it is impenetrable and he cannot access the valuable materials inside. After using her key to open the confinement unit, Alice removes the finished ring or necklace. In conclusion, the worker successfully creates a finished product without having true access to raw materials.

Observation In this first part is explained the general concept of Homomorphic Enchryption where the locked box with the precious materials represent an encryption of initial data m, the homomorphism of encryption scheme is represented by the gloves that workers use and the final products (rings and neckclace) correspond to the function $f(m)$ to apply to encrypted data.

Story Alice, after discovering how to utilize locked gloveboxes to have her employees process her valuable materials into elegant rings and necklaces, makes a purchase from the Acme Glovebox Company. Regrettably, the gloveboxes she receives are faulty. Following just one minute of use, the gloves stiffen and become inoperable. However, some of the most extravagant pieces require up to an hour to assemble. Is there a way I can employ these defective boxes to ensure that the workers securely assemble even the most intricate pieces? She observes that, despite their defects, the boxes possess a characteristic that could prove useful. As anticipated, they possess a unidirectional insertion slot, akin to mail bins in post offices. Moreover, they exhibit flexibility allowing one box to be placed inside

another through the slot. Alice contemplates whether this attribute could play a role in resolving her predicament...

Observation In the analogy, the defective gloveboxes represent a somewhat homomorphic encryption scheme (SWHE), that can perform additions and multiplications operations on ciphertexts for a little while (it can handle the evaluation of functions only for a limited numbers of times).

Story A revelation dawns upon her, she possesses the knowledge to utilize her faulty containers in a manner that guarantees the secure delegation of intricate assemblies.
Following her previous method, she presents a worker with a glovebox labeled #1, containing the raw materials. However, she bestows upon him multiple supplementary gloveboxes. Glovebox #2 safeguards the key to unlock Glovebox #1, while Glovebox #3 conceals the key to unlock Glovebox #2, and so forth. In order to construct the detailed design, the laborer manipulates the materials within Glovebox #1 until the gloves stiffen. Subsequently, he situates Glovebox #1 within Glovebox #2, the latter already containing the necessary key. Utilizing the gloves associated with Glovebox #2, he unlocks Glovebox #1, carefully extracting the partially assembled jewel. The assembly process continues within Glovebox #2 until its gloves stiffen as well. Then, he proceeds to place Glovebox #2 inside Glovebox #3, and the cycle persists. This pattern continues until the worker finally completes the assembly within Glovebox #n, which he then graciously presents to Alice.
Naturally, Alice comes to the realization that this ingenious technique can only succeed if the worker manages to unlock Glovebox #i within Glovebox #(i + 1) and still has ample time to advance the assembly slightly, all before the gloves of Glovebox #(i + 1) become stiff. As long as the unlocking process (along with a modest amount of assembly work) consumes less than a minute and an ample supply of defective gloveboxes is available, the possibility to construct any intricate piece, regardless of complexity, becomes feasible.

Observation This final part of the analogy represent Gentry's innovative solution. It turns out that the decryption function (which is like opening the "encryption box") is the only function that a scheme actually needs to be able to handle, with a tiny amount of room remaining to execute one more operation. If a scheme possesses the self-referential quality of being able to manage its own decryption function (added by a single operation) we refer to it as being **bootstrappable**. This last property is the fundament of Gentry's solution to construct a Fully Homomorphic Encryption scheme.

2.3 Fully Homomorphic Encryption

2.3.1 Definition of FHE

FHE could be formalized in the following way [47]:

$$Enc(x) \rightarrow f'(Enc(x)) = Enc(f(x))$$

1. x is the plaintext, the sensitive data
2. $Enc(x)$ is the operation of Encryption applied to the data x to obtain the ciphertext
3. $f'(Enc(x))$ is the function/operation applied over the encrypted data in the ciphertext space that corresponds to $f(x)$ function in clear, where $f()$ can be any type of function.
4. $f'(Enc(x)) = Enc(f(x))$ is the condition needed by the scheme to be homomorphic

2.3.2 Bootstrapping

The concept of bootstrapping was introduced by Gentry in 2009 as mentioned above, but before giving a definition of it, there is a property of homomorphic ciphertext to introduce: each ciphertext is associated to a **noise**.
Since the majority of FHE schemes are based on hard lattice problems, to ensure the encryption's security, the generated ciphertexts will cointain a certain quantity of noise (depending on the scheme and on the operations performed). Every operation computed homomorphically on the ciphertext increases the value of such noise. To decrypt the ciphertext the level of noise *must be below a specific threshold*, otherwise the noise can overflow the data making the decryption impossible [25].
In figure 2.4 we show an example of a valid ciphertext and another one where the noise exceeds the threshold, giving as result an incorrect decryption.

The **Bootstrapping** is a special procedure first described by Gentry that can be applied to the ciphertext with the aim of *noise reduction*, in order to be able to compute more homomorphic operations on the encrypted data. From the theoretical perspective a HE scheme is **bootstrappable** if it can evaluate its own decryption function homomorphically, using an encryption of the secret key, in addition to at least one extra operation [18].

The homomorphic evaluation of the decryption function (bootstrapping) uses an encrypted secret key on a *exhausted* ciphertext to convert it into an "equivalent" *refreshed* ciphertext by reducing the associated noise level. In an exhausted

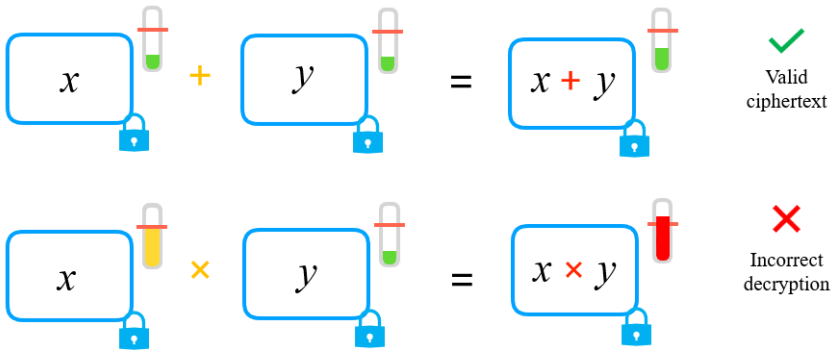

Figure 2.4. Noise in FHE [8]

ciphertext no futher operations can be performed because of the high noise level, whereas the refreshed one can support further homomorphic operations. The encrypted secret key is provided by the secret key holder as part of the public key material and it is called *bootstrapping*, *evaluating* or *refreshing* key [39].

Example

1. We start with a ciphertext which is an encryption of plaintext x represented by the blue box with a lock and the associated noise level that has reached the maximum limit after previous operations. This means that at this point we are not able to perform other homomorphic operations, to continue we must reduce the noise. The noise level of the ciphertext is represented in the figure by a thermometer, next to the box, with a red line as threshold.

2. The general idea is to decrypt the ciphertext because it corresponds to opening the blue box and take out the plaintext x totally deleting the noise, but to decrypt it we need the secret key that we cannot disclose publicly. So we put the ciphertext in a green box that represents the FHE scheme with a little value of noise, this step has no cost in terms of increasing the noise of the blue box.

3. We can perform the decryption of the ciphertext opening the blue box inside the green one using the encrypted secret key called bootstrapping key (blue key in a green box), which is public.

4. The result will be the original plaintext x inside the green box with the level of noise slightly increased by the bootstrapping operation but lower than the starting error, so as to leave some space to perform more homomorphic computation [9].

In figure 2.5 we represent the four steps described above.

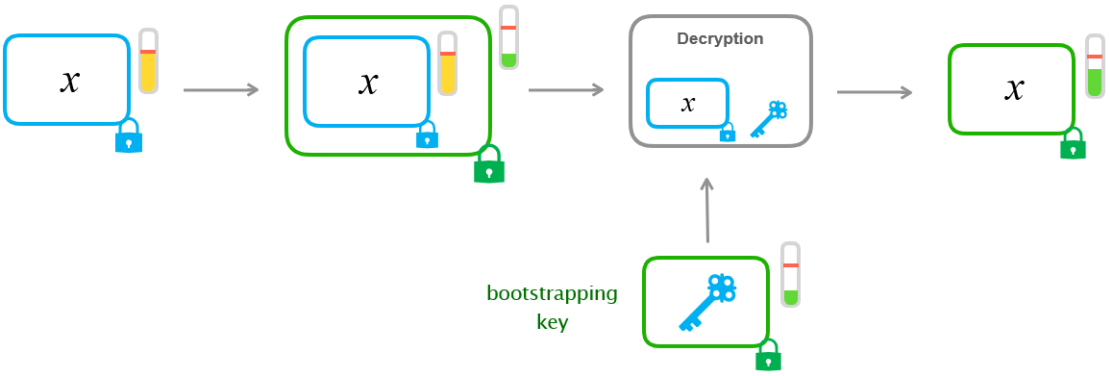

Figure 2.5. Bootstrapping procedure [8]

2.3.3 General Distinctions between Schemes

In practice we can perform homomorphic computations and then bootstrapping as soon as the noise grows: this allows us to evaluate potentially any possible circuit without having any limits on the number of operations to perform. Unfortunately bootstrapping is the most costly tecnique in homomorphic encryption and the advantages of applying it depend on the type of circuit being evaluated. For this reason, the generations of FHE schemes after Gentry's proposal have focused on the control of noise growth and the optimization of bootstrapping. There are two particular approches that have emerged called **Leveled** and **Bootstrapped**.

- The **Leveled** approach (**Less noisy operations**) consists in FHE schemes parameterized to represent a given function with a circuit that will be evaluate homomorphically without resorting to bootstrapping operation. With this approach there is a try to avoid bootstrapping as much as possible and it can be used when the circuit is "**small**" and especially **known**, so that the appropriate parameters can be set.
  A leveled FHE scheme provisions a noise budget to support N level of operations, generally the multiplication is chosen because homomorphically introduces the most noise, where N is the multiplicative depth of the circuit (the largest sequence of consecutive multiplications). Knowing the operations performed inside a small circuit it can be possibile to fix the parameters in order to fit the number of operations inside the amount of noise given.
  The computation's cost and speed depend on the size and depth of the circuit: if the circuit becomes larger the number of parameters also increases and the evaluation becomes slower, just as the depth of the circuit increases the complexity of computation.

- The **Bootstrapped** approach (**Fast Bootstrapping**) consists in a more flexible solution where there is no limitation to the number of operations to perform and the bootstrapping operation is computed every time it is needed. This approach is used when the circuit is **unknown** or "**deep**".
  The schemes that follow a bootstrapped approach focus on improving the performance of the bootstrapping operation. The new generations of FHE schemes guarantee a significant speedup in the bootstrapping step compared to what was possible in Gentry's 2009 scheme.

The use case determines whether to apply an FHE strategy from the first or second approach. Less complex use-cases can be tackled by using Leveled FHE schemes, while Bootstrapped FHE schemes are useful in more complex scenarios, but there is also the possibility to combine both approaches.

Another important distinction involving FHE schemes is how they implement circuits and what type of data they operate on. There are two main classes of circuits called **Boolean** and **Arithmetics**.

- **Boolean Circuits**
  This type of circuit works on bits. Any function can be represented by a boolean circuit, which is made of a series of binary gates that are connected together, where the addition operation of bits corresponds to a XOR gate and the AND gate is the parallel of a multiplication of bits; it is even possible use only the universal $NAND$ gate to express all operations.
  In general this type of circuits are used following the bootstrapped approach.

- **Arithmetics Circuits**
  This type of circuit represents inputs with larger integers composing a series of additions and multiplications. The integers are modulo p for some $p > 2$ (boolean arithmetic corresponds to $p = 2$).
  These circuits are considered mainly in leveled approach and in some cases for schemes operating with real numbers by having approximate computations [25].

2.3.4 FHE Schemes

Since the solution proposed by Gentry in 2009, various schemes and related variants of them based on the idea of bootstrapping have been devised, making significant improvements to the original idea. In this section the most well-known FHE schemes on which nowadays there is more development will be discussed, along with their main features and most significant characteristics.

DM/CGGI

In the **Ducas-Micciancio** (**DM**) cryptosystem [29] (proposed in 2015), also known as **FHEW** and based on the **GSW** scheme [19], the primary goal was to perform a evaluation with the smallest latency of an elementary bootstrapped computation, in this case a boolean $NAND$ gate. Let us recall that the $NAND$ gate can be used to implement any function expressed as a boolean circuit because it is a complete boolean gate, this property permits to a single gate to implement any logical operation. Compared to the original scheme of Gentry that took up to a maximum of 30 minutes for a single bootstrapping operation, the DM cryptosystem is much faster, being the first FHE scheme to perform a bootstrapping operation in less than a second.

A variant of DM cryptosystem that is more efficient in terms of memory is the **Chillotti-Gama-Georgieva-Izabachene** (**CGGI**) cryptosystem [12] proposed in 2016, it has a similar design to DM, but it has stronger security assumptions and it makes use of some additional optimizations that improve bootstrapping efficiency allowing it to achieve a latency of less than 0.1 seconds.

These two cryptosystems, DM and CGGI, have a unique property that differentiates them among all other FHE schemes: they permit to evaluate arbitrary functions during the bootstrapping process. The evaluation can be done by replacing the bits of plaintext with a function of them thanks to a lookup table during the bootstrapping procedure. This operation is called **Functional** or **Programmable Bootstrapping**.

CKKS

To have a scheme that provides nowadays the best efficiency in Machine Learning applications, being optimized for floating point computations, we can refer to the **Cheon-Kim-Kim-Song** scheme (**CKKS**) [7] proposed in 2016. This scheme makes extensive use of polynomial approximations to implement non linear functions, consequently the bootstrapping operation is also approximated, i.e., in the refreshed ciphertext the encrypted message is not equal but "close" to the encrypted message in the exhausted ciphertext [39].

BGV/BFV

The **Brakerski/Fan-Vercauteren** (**BFV**) scheme [17] and the **Brakerski-Gentry-Vaikuntanathan** (**BGV**) scheme [54], proposed respectively in 2012 and 2014, have a very similar structure and use the same bootstrapping strategy of CKKS, but unlike the latter which is optimized for floating point computations, these two schemes are suitable for working on encrypted exact computations with

integer data types. For this reason they can be useful into the category of applications whose main focus is string manipulation and database management.
In the BGV/BFV scheme the bootstrapping procedure is very computational expensive because it requires homomorphic evaluation of high-degree polynomials.

Each scheme has advantages and disadvantages depending on how it is implemented and what strategy it uses to carry out the homomorphic decryption evaluation operation. To be more specific, BGV/BFV and CKKS schemes fall into the category of the **leveled** approach while DM/CGGI fall into that of the **bootstrapped** approach. Each of them can be used in the area that is most relevant to it by exploiting its leading features. In addition to the above mentioned FHE schemes there are others developed after the discovery of Gentry that we will not focus on in this thesis; in figure 2.6 is depicted a timeline where the major FHE schemes in the post-Gentry era are specified.

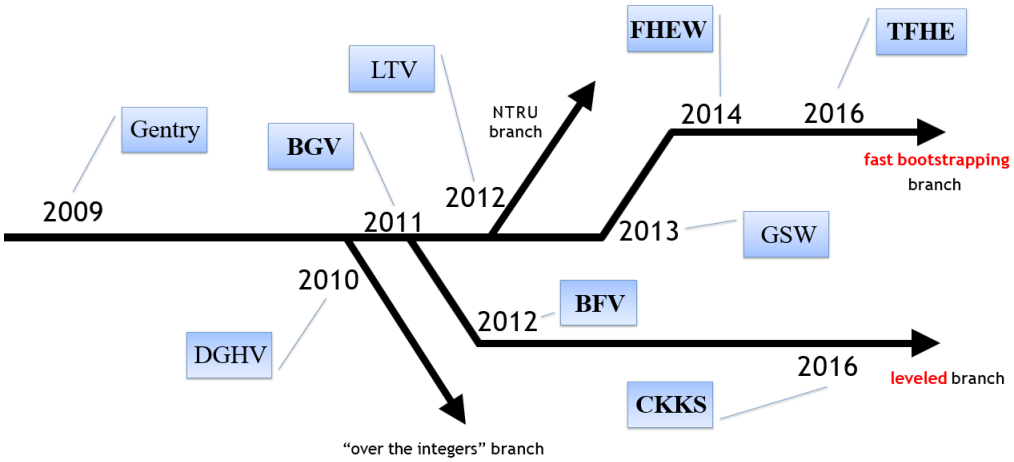

Figure 2.6. FHE schemes timeline after Gentry's solution [8]

2.4 Fully Homomorphic Encryption over the Torus

The *Fully Homomorphic Encryption over the Torus* (TFHE) [13] is a Fully Homomorphic Encryption scheme, also known as CGGI [12], and it was proposed in 2016 as improvement of FHEW. The security of the scheme, as the majority of the FHE scheme nowadays, is based on a hard lattice problem called *Learning With Errors* (LWE) and its variants such as *Ring-LWE* (RLWE). The main feature of

this scheme is the special bootstrapping operation (**Programmable Bootstrapping**) that is able to reduce the noise of ciphertext and to evaluate an arbitrary function at the same time in an extremely fast way.

2.4.1 TFHE ciphertexts

In the TFHE scheme are used three different types of ciphertexts (LWE, RLWE, RGSW) because all of them have different properties useful in the homomorphic operations [10].

1. LWE ciphertext

Given a message m, an integer n and a secret key $\vec{\mathbf{s}}$ a **LWE ciphertext** is defined as a vector of $n+1$ elements

$$\vec{\mathbf{c}} = Enc_{\vec{\mathbf{s}}}(m) = (\vec{\mathbf{a}}, b)$$

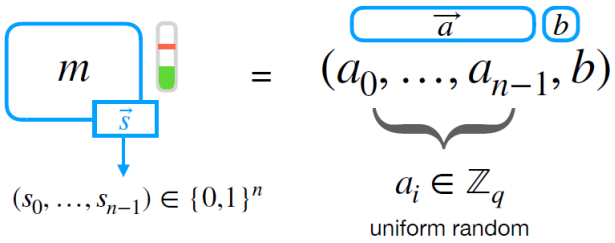

Figure 2.7. LWE Ciphertext visualization [8]

where

- m can be a bit, a modular integer or a real in a interval and $\vec{\mathbf{s}}$ is a vector of random n bits $(s_0, ..., s_{n-1}) \in \{0,1\}^n$
- $\vec{\mathbf{a}}$ is a vector of n random integers $(a_0, ..., a_{n-1}) \in \mathbb{Z}_q$ where q is an integer.
- the $(n+1)$-th element is $b = \sum_{i=0}^{n-1} a_i \cdot s_i + e + \Delta m$
- e is a small error chosen according to a Gaussian Distribution and Δ is a scaling factor

It is possible to **decrypt** the ciphertext as follows

$$b - \vec{\mathbf{a}} \cdot \vec{\mathbf{s}} = \Delta m + e$$

At this point it is possibile obtain the message m performing a **rounding** operation.

It is possibile to perform **linear combinations** with LWE ciphertexts:

- **Addition**: Given two LWE ciphertexts $\vec{\mathbf{c}}$ e $\vec{\mathbf{c}}'$ the addition is defined as $\vec{\mathbf{c}} + \vec{\mathbf{c}}' = (\vec{\mathbf{a}}, b) + (\vec{\mathbf{a}}', b') = (\vec{\mathbf{a}} + \vec{\mathbf{a}}', b + b')$.
- **Costant multiplication**: Given a LWE ciphertxet $\vec{\mathbf{c}}$ and an integer γ the costant multiplication is defined as $\gamma \cdot (\vec{\mathbf{a}}, b) = (\gamma \cdot \vec{\mathbf{a}}, \gamma \cdot b)$.

2. RLWE ciphertext

Given a polynomial $M(X)$ and a secret key $S(X)$ a **RLWE ciphertext** is defined as

$$C(X) = Enc_{S(X)}(M(X)) = (A(X), B(X))$$

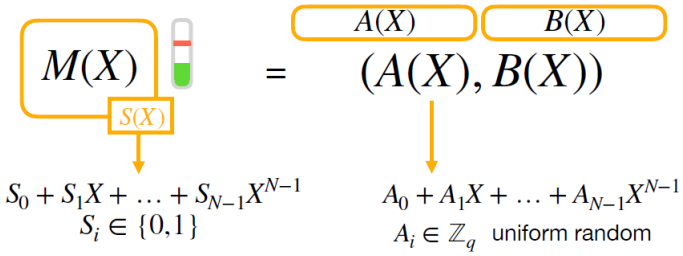

Figure 2.8. RLWE Ciphertext visualization [8]

where

- $M(X)$ is a polynomial modulo $(X^N + 1)$ with N coefficients (where each one of them can represent bit, integer or real) and $S(X)$ is a polynomial with N coefficients that are random bits ($S_0 + S_1X + ... + S_{N-1}X^{N-1}$ with $S_i \in \{0,1\}$).
- $A(X)$ is a polynomial of N coefficients that are random integers ($A_0 + A_1X + ... + A_{N-1}X^{N-1}$ with $A_i \in \mathbb{Z}_q$).
- $B(X) = A(X) \cdot S(X) + E(X) + \Delta M(X)$.
- $E(X)$ is a polynomial with N coefficients that represents the errors ($E_0 + E_1X + ... + E_{N-1}X^{N-1}$ with E_i taken from a Gaussian distribution) and Δ is a scaling factor.

It is possible to **decrypt** the ciphertext as follows

$$B(X) - A(X) \cdot S(X) = \Delta M(X) + E(X) \rightarrow M(X)$$

. In order to recover the plaintext we once again resort to a rounding operation.

It is possibile perform **linear combinations** with RLWE ciphertexts:

- **Addition**: Given two LWE ciphertexts $C(X)$ e $C'(X)$ the addition is defined as $C(X)+C'(X) = (A(X), B(X))+(A'(X), B'(X)) = (A(X)+A'(X), B(X)+B'(X))$.
- **Costant polynomial multiplication**: Given a RLWE ciphertxet $C(X)$ and an integer polynomian $\Gamma(X) \in \mathbb{Z}[X]/(X^N + 1)$ the costant polynomial multiplication is defined as $\Gamma(X)\cdot(A(X), B(X)) = (\Gamma(X)\cdot A(X), \Gamma(X)\cdot B(X))$.

2. RGSW ciphertext

Given a polynomial $\mu(X)$ and a secret key $S(X)$, defined in the same way as in RLWE, a **RGSW ciphertext** is defined as 3-dimensional matrix $2 \times 2 \times l$. The matrix can be seen as a list of l elements, each of which is a 2×2 matrix composed of 4 polynomials where each line is a RLWE ciphertext.

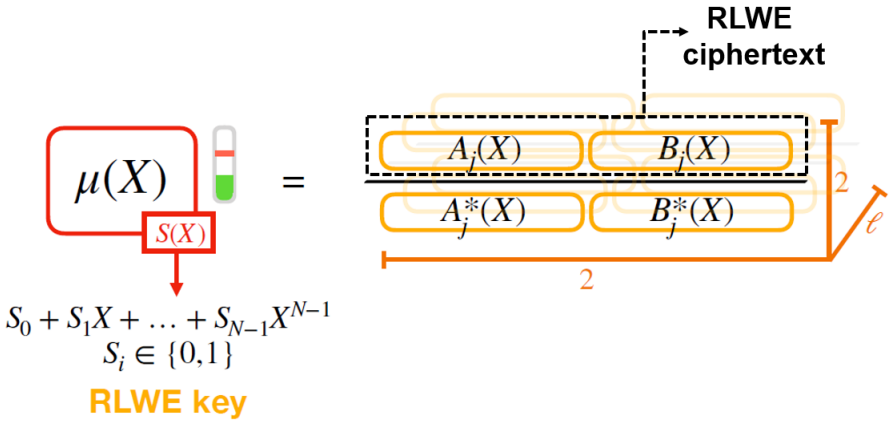

Figure 2.9. RGSW Ciphertext visualization [8]

The important difference of RGSW ciphertext is that, in addition to performing the linear combinations (addition and constant polynomial multiplication) as seen for the other ciphertexts types, it can perform the **multiplication**:

$$Enc_{S(X)}(\mu(X)) \otimes Enc_{S(X)}(\mu'(X)) = Enc_{S(X)}(\mu(X) \cdot \mu'(X))$$

.

2.4.2 Operations on TFHE ciphertexts

The three different ciphertext types can interact through appropriate blocks and operations by exploiting their properties that will be useful in performing

homomorphic operations and bootstrapping. This section will briefly present these operations, considering the notation expressed above.

1. External Product

The **External Product** consists in performing a multiplication between a RLWE and a RGSW ciphertext:

$$Enc_{S(X)}(M(X)) \odot Enc_{S(X)}(\mu(X)) = Enc_{S(X)}(\mu(X) \cdot M(X))$$

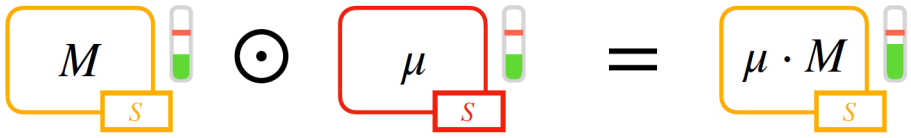

Figure 2.10. External Product [8]

2. CMux Gate

The **Controlled Mux** is a gate that takes in input the elements d_0, d_1, b and, depending the value of $b \in \{0,1\}$, selects as output d_0 or d_1.

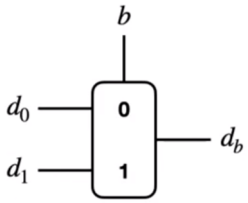

Figure 2.11. CMux Gate

CMux corresponds to a **if condition** on b value and it can be performed in cleartext by evaluating this expression $(d_1 - d_0) \cdot b + d_0 = d_b$. To homomorphically evaluate CMux we encrypt d_0 and d_1 with RLWE and b with RGSW, and then we use the external product and addition operation:

$$(Enc_S(d_1) - Enc_S(d_1)) \odot Enc_S(b) + Enc_S(d_0) = Enc_S(d_b).$$

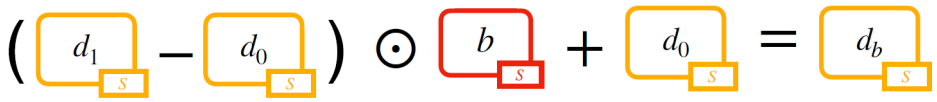

Figure 2.12. Homomorphic CMux [8]

3. Blind Rotation

The **Rotation** in clear consists in rotating a polynomial $M(X)$ of p positions (known) and it can be done by performing the multiplication $M(X) \cdot X^{-p}$ to bring the M_p element in the first position where $M(X) = M_0 + M_1X + ... + M_pX^p + ... + M_{N-1}X^{N-1}$.

The **Blind rotation** consists in rotate an encrypted RLWE polynomial $M(X)$ of p RGSW encrypted positions, with $p = p_0 \cdot 2^0 + ... + p_k \cdot 2^k$ where p_j is secret and 2^j is known $\forall j = 0, ..., k$.
$M(X) \cdot X^{-p}$ can be developed as

$$M \cdot X^{-p_0 2^0 - ... - p_k 2^k} = M \cdot X^{-p_0 2^0} \cdot ... \cdot X^{-p_k 2^k}.$$

At this point to obtain $M(X) \cdot X^{-p}$ it can be built a chain of CMux one concatenated to the other where each one of them correspond to

$$M \cdot X^{p_j 2^j} = \begin{cases} M & \text{if } p_j = 0 \\ M \cdot X^{-2^j} & \text{if } p_j = 1. \end{cases}$$

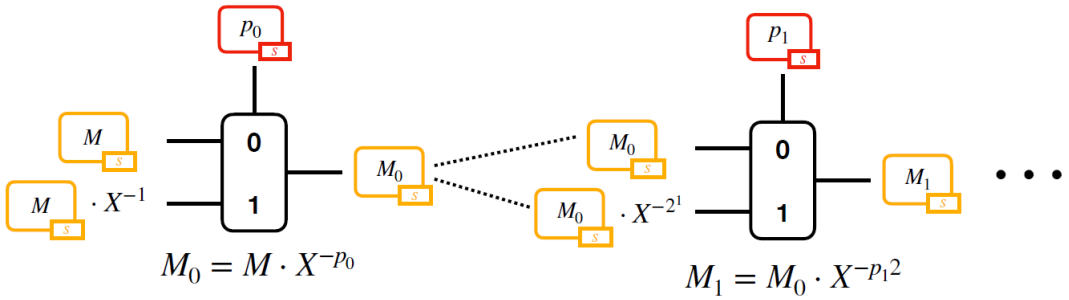

Figure 2.13. Chain of CMux to perform Blind Rotation [8]

4. Sample Extraction

The **sample extraction** takes in input a RLWE ciphertext encrypting a polynomial $M(X)$ and extracts homomorphically one coefficient M_i, putting the result in a LWE ciphertext. This operation can be done easily by reassigning the coefficients of the input in a specific order to the coefficient of the output without increasing the noise.
For example, starting to $Enc_{S(X)}(M(X)) = A(X)B(X)$, to extract M_0 and thus obtain $Enc_{\vec{\mathbf{s}}}(M_0) = (\vec{\mathbf{a}}, b)$ we can do as follows:

- $\vec{\mathbf{s}} = (s_0 = S_0, ..., s_{n-1} = S_{N-1})$ with $n = N$.
- $(\vec{\mathbf{a}}, b) = (a_0 = A_0, a_1 = -A_{n-1}, ..., a_{n-1} = -A_1, b = B_0)$.

5. Key switching

The **key switching** is an operation that allows, for a certain type of ciphertext, to switch from one secret key to another. For example, between two LWE ciphertexts:

$$Enc_{\vec{\mathbf{s}}}(m) = (\vec{\mathbf{a}}, b) \rightarrow Enc_{\vec{\mathbf{s}}'}(m) = (\vec{\mathbf{a}}', b')$$

To perform this operation there is need of a *key-switching key* (public key similar to a bootstrapping key) that is used to switch the key and also to change the parameters. The key switching operation increases the noise associated to the ciphertext.
The possible combinations of ciphertext types among which the key switching can be performed are: LWE to LWE, RLWE to RLWE, LWE to RLWE and many-LWE to RLWE.

2.4.3 Programmable Bootstrapping

The aim of **bootstrapping** is to reduce the noise when it grows too much. In the TFHE scheme it is possible to perform the bootstrapping operation on LWE ciphertext $Enc_{\vec{\mathbf{s}}}(m) = (\vec{\mathbf{a}}, b)$ that consists in evaluating homomorphically the decryption following two steps:

1. Computation of the linear combination $b - \sum_{i=0}^{n-1} a_i \cdot s_i = \Delta m + e$
2. Rescale and rounding: $\lceil \Delta m + e \rfloor = m$

in order to obtain the same encrypted m with less noise.

Initially we consider the message $m \in \{0,1, ..., p-1\}$, $\Delta m + e \in \{0,1, ..., q-1\}$ and $\vec{\mathbf{s}}$ of n elements. Taking in input a LWE ciphertext $Enc_{\vec{\mathbf{s}}}(m) = (\vec{\mathbf{a}}, b)$, the procedure consists in computing a **blind rotation** on a vector V. The vector V is obtained by repeating Δ times each value between 0 and $p-1$ as follows:

Figure 2.14. Values $(\Delta m + e)$ sent to the value m [8]

We call *mega-case* the block containing multiple repetitions of the same value, while the single repetition of a value is called *case*.
In this way, when we rotate V by $\Delta m + e$ positions, if $|e| < \Delta/2$ we end up having in the leftmost position of V the value m. We hence have a way to associate a value $\Delta m + e \in \mathbb{Z}_q$ to a value $m \in \mathbb{Z}_p$ $(p < q)$. In the following figure we give a representation of the values associations:

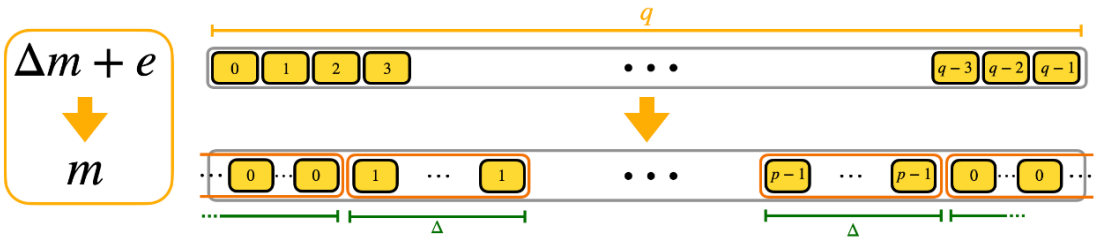

Figure 2.15. Values $(\Delta m + e)$ sent to the value m [8]

At this point we perform a multiplication with the monomial $X^{-(\Delta m+e)}$ where

$$V_n = V \cdot X^{-b+\sum_{i=0}^{n-1} a_i \cdot s_i = \Delta m+e} = V \cdot X^{-(\Delta m+e)}.$$

This operation rotates the input vector V of $\Delta m + e$ positions in order to bring the case corresponding to the plaintext m in first position.

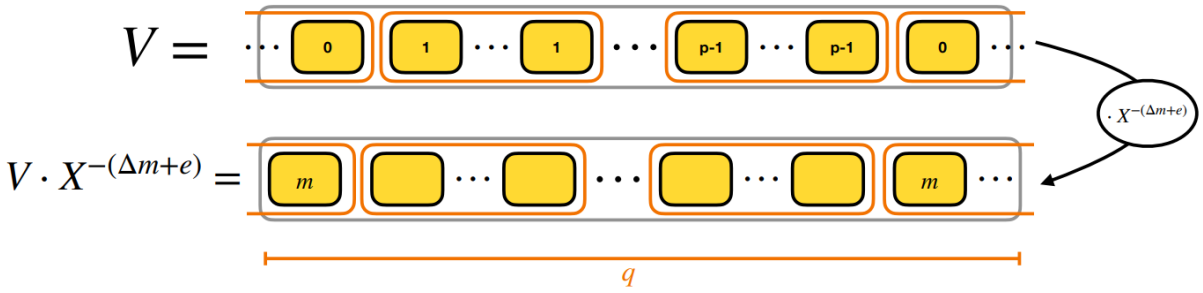

Figure 2.16. Rotation of V [8]

Then it is used the **sample extraction** over the result (RLWE) to obtain the LWE encryption as output in which there will be m encrypted with secret key $\vec{\mathbf{s}}'$ with less noise. In the end the **key switching** is performed to go back to secret key $\vec{\mathbf{s}}$.
The following figure represents the Bootstrapping procedure with the flow of operations performed:

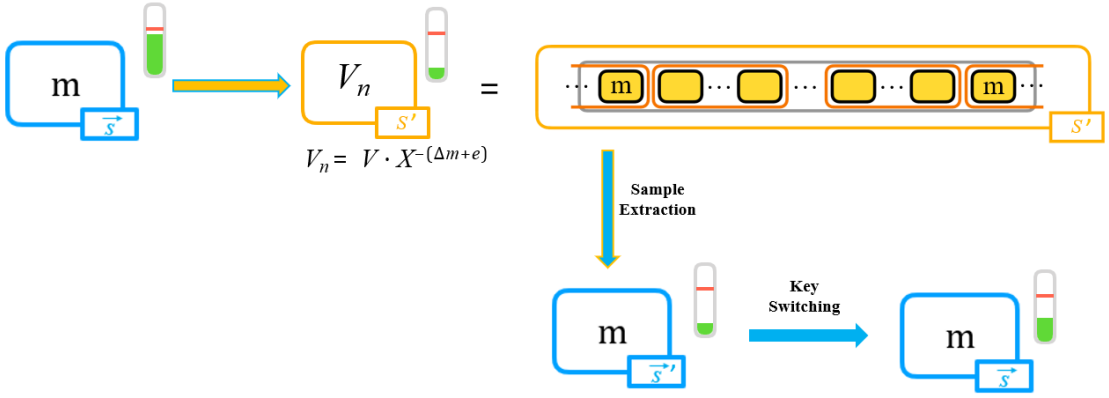

Figure 2.17. Bootstrapping Procedure of TFHE Scheme [8]

In the procedure just described, if the vector V is not built directly with the elements of the inital vector $\{0,1,...p-1\}$ but with elements on which is applied a f function $\{f(0), f(1), ..., f(p-1)\}$, after the step of introducing redundancy it

is possible to use V as **Homomorphic Look-Up Table**.
In this way it is possibile to evaluate a function f and reducing the noise at the same time, this is called **Programmable Bootstrapping**.

Considering the two steps for evaluating homomorphically the decryption mentioned at the beginning of this section (2.4.3), according to the work done by ZAMA [11] the procedure of programmable bootstrapping is summarized by the following statement:
"*To perform* ***programmable bootstrapping*** *the TFHE approach is to put the computation of (the negation of)* $b - \sum_{i=0}^{n-1} a_i \cdot s_i = \Delta m + e$ *in the exponent of a monomial* X*, and then to use this new monomial to rotate a Look-Up Table (LUT) that evaluates the second step of the decryption (rescale and rounding)*".

In order to end up, this is a summary of all the operations showed that, combined in appropriate way between the ciphertext types, can be used to build a large amount of homomorphic operations.

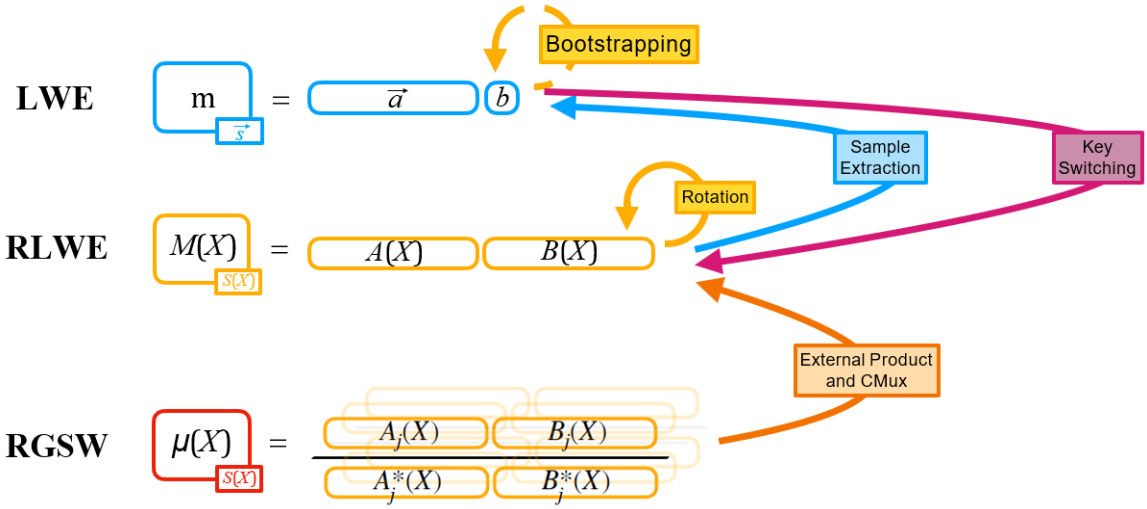

Figure 2.18. Operations in TFHE [8]

Chapter 3

FHE Libraries and tools: Concrete Compiler

In this chapter we will focus on Concrete Compiler and we will describe how to use it to develop programs that use FHE to compute on encrypted data.
We will give a general overview of the product developed by ZAMA. In section 3.1 we will focus on the inner workings of Concrete Compiler, while in section 3.2 we will show how to use Concrete through its Python API by directly analyzing some examples.

3.1 Concrete Overview

ZAMA is a French cryptography company that focuses on building open source homomorphic encryption solutions [52]. ZAMA focuses on the development of applications in the field of Artificial Intelligence and Blockchain and produces open source cryptographic tools with the aim of simplifying data protection.
The core product of ZAMA is a compiler called **Concrete**, designed especially for developers, helping them about the complexity of operating with FHE, managing noise, choosing appropriate crypto parameters and finding the best set and order of operations for a specific computation.

In April 2023 ZAMA released the version of the compiler for TFHE that converts Python programs into FHE equivalents, Concrete v1.0.0 (before known as concrete-numpy). In that release ZAMA states: "Concrete is useful for developers who want to build a high level application that takes encrypted inputs and produces encrypted outputs" [49]. Concrete is written in C++ and it is developed with *Multi-Level Intermediate Representation* (MLIR) that is a open source project used for building reusable and extensible compiler infrastructure [28].

3.1.1 From Python Program to TFHE Circuits

One of the biggest problems is to translate all the operations of a program into their equivalent homomorphic form. It is simple to obtain the homomorphic equivalent of simple operations like addition or multiplication but not for any other type of function.
In TFHE homomorphic *Look Up Tables* (LUTs) can be evaluated for free during the operation of bootstrapping, which reduce the noise. This is a great advantage of TFHE because any univariate function can be represented by a LUT and so TFHE can evaluate any function without approximation (Programmable Bootstrapping 2.4.3).
The translation of a function to LUTs is performed automatically by the Compiler, turning regular functions into univariate ones and then generating the corresponding LUT. On the contrary, the direct use of LUTs within Python API result extremely inefficient.

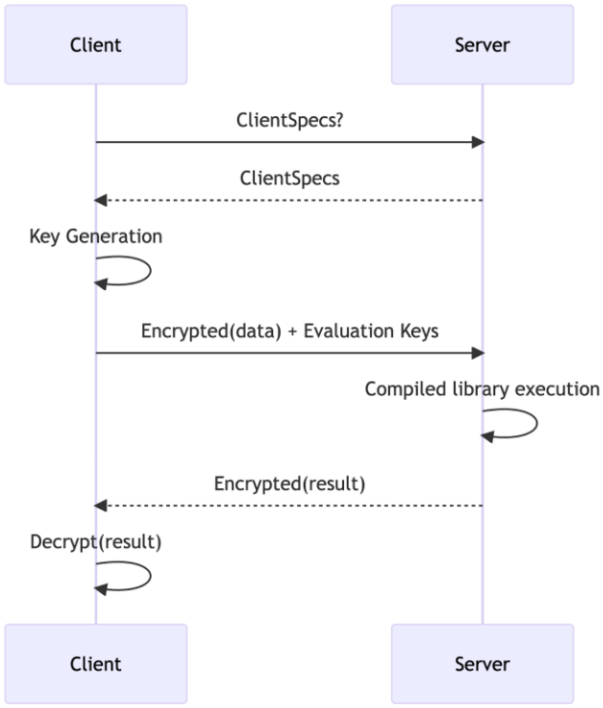

Figure 3.1. Typical Client-Server interaction in Concrete [49]

Each function is represented internally to the compiler by a **circuit**. A circuit is defined as a direct acyclic graph of operations on variables, where each variable can be encrypted or in clear. This circuit is compiled with all the parameters and the result is:

- a dynamic library with FHE operations.
- a JSON file with the cryptographic configuration called **Client Specs**.

At this point the circuit can perform the homomorphic evaluation of the desired function. These operations can be used in the context of a Client-Server interaction, for which we will see a detailed example in the implementation of the project of this thesis in the section 4.3. We show a general example of communication in Figure 3.1.

3.1.2 Parameters Optimization and Supported Data

In the Concrete Compiler there is a TFHE Optimizer that automatically computes the best possible parameters to use in the homomorphic scheme for the program being compiled. In a FHE library the cryptographic parameters define data size, generated noise, maximum number of operations to perform before the programmable bootstrapping execution. These parameters influence performances, exacteness and security of an application. The fact that the choice of these parameters is automatically performed is hence a great advantage for the developer.

Currently Concrete has a strong limitation, given that it supports input variables of maximum size up to 16 bits. It supports simple operations (e.g. addition) on 32 bits but they are extremely inefficient.
Concrete, with its Python API, is compatible with a limited number of Numpy library functions that can be used directly and can make use of the data structures such as Array, Lists, ndArray and Tensor. For such data types Concrete can be used only on Integers and there is no support for Floating Point. There is also an additional limitation regarding the fact that no flow control constructs (e.g. if,for,while) can be used in the homomorphically evaluated function.

3.2 Using Concrete

Now we present and analyze a simple circuit written in Python that performs an homomorphic evaluation of the addition of two encrypted variables.

```
from concrete import fhe

#Function to evaluate
def add(x, y):
    return x + y

#Definitions parameters
compiler = fhe.Compiler(add, {"x": "encrypted", "y": "encrypted"})
inputset = [(2, 3), (0, 0), (1, 6), (7, 7), (7, 1)]

#Compile Circuit
circuit = compiler.compile(inputset)
#Input (x,y)
input = (3, 4)
#Execution
homomorphic_evaluation = circuit.encrypt_run_decrypt(input)
```

Listing 3.1. Circuit Example

- `from concrete import fhe` imports the library module that allows to perform homomorphic evaluation
- `def add(x,y)` is the definition of the function to evaluate in the circuit
- `fhe.Compiler` creates a `compiler` by specifying the function to compile (`add`) and the encryption status of its inputs (`"encrypted"`).
- `inputset` is a data collection that represents the inputs to the function and it is used to determine the size in bit of the variables within the function. The `inputset` must be an iterable set where each element is of the same length as the number of input arguments of the function.
- `compiler.compile` performs the compilation and get resulting the `circuit`.
- `encrypt_run_decrypt` performs the homomorphic evaluation at once, but the three method (`encrypt`, `run`, `decrypt`) can be performed separately and by different parties (e.g. the client encrypts and decrypts data and the server runs the homomorphic circuit).

Chapter 4

Homomorphic SHA256

In this chapter we will discuss the main project developed during this thesis, an implementation of homomorphic evaluation of SHA256 using Concrete Compiler [53]. We will first examine the technical aspect and the internal structure of SHA256 algorithm [35] in section 4.1 (following the *Secure Hash Standard* (SHS) published by NIST [35]), analyzing the inner blocks of the algorithm and the operations needed for the computation. Then we will explain the homomorphic evaluation of SHA256 in section (4.2) by going through the details of the internal data representation and operations, justifying the implementation and design choices taken during the development phase. Finally, in section 4.3 we will present a general Client-Server model in which we will be able to show how the developed implementation is used in real-world communication.

4.1 SHA256 Algorithm

SHA256 is a cryptographic hash function and it part of SHA-2 family of SHS [35] as mentioned in section 1.4.
This algorithm garantees the property of message's integrity: it can process a message in an iterative way to produce a representation of fixed length called **digest**, and any change to original message produce a different digest. The integrity property of the message is very useful in the context of message authentication code and digital signatures.

To understand how data is organized in the algorithm we define the following elements:

- **word**: data unit on which the various operations will be executed. It is a w-bit string where the size is $w = 32$ bits and it is represented in hexadecimal format.

- **block**: structure corresponding to an array of words. The block size is 512 bits and a block is composed by 16 words.
- **digest**: Hash function's final result which is a group of 256 bits, generally represented by a hexadecimal string.

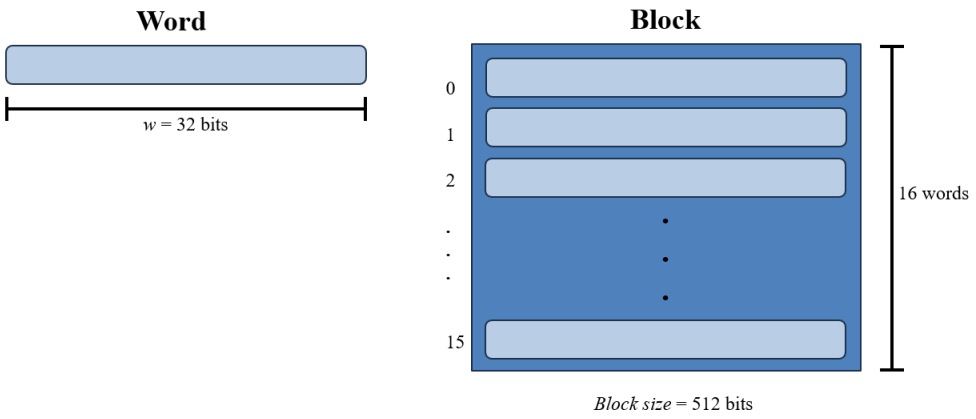

Figure 4.1. Word and Block representation in SHA256 Algorithm

SHA256 algorithm can be described in two phases called **Preprocessing** (4.1.1) and **Hash Computation** (4.1.2).
The preprocessing phase is divided in three steps which consist in padding a message, parsing the padded message and setting initial values for the hash computation phase. In the hash computation phase are used functions and word operations in iterative way on the results from preprocessing phase to generate a series of hash values. In the end the message digest is taken from the final hash value of the computation phase.

4.1.1 Preprocessing

The preprocessing phase prepares the input message by formatting it appropriately and setting constant values for the hash computation phase where the main processing loop takes place.

1. Padding

Given in input a message of length ℓ bits, the padding step ensures that the padded message has a length equal to a multiple of the block size (512 bits). The procedure is as follows:

- Append a single bit with value "1" to the end of message.
- Append k bits with value "0" until the length is 64 bits less than a multiple of 512. k is the smallest non-negative solution to the equation

$$\ell + 1 + k = 448 \mod 512$$

- Append 64 bits with the value equal to the number ℓ expressed in binary representation.

Example: We suppose that the input message is "*abc*" which has length $\ell = 8 \times 3 = 24$bits (8-bit ASCII). To perform padding we append a single bit with value "1" and then $448 - 24 - 1 = 423$ bits with value "0". At the end we append 64 bits with the value of $\ell = 24$ that is "011000" in binary representation. This is an example with one block, in fact we obtain as result a padded message of 512 bits.

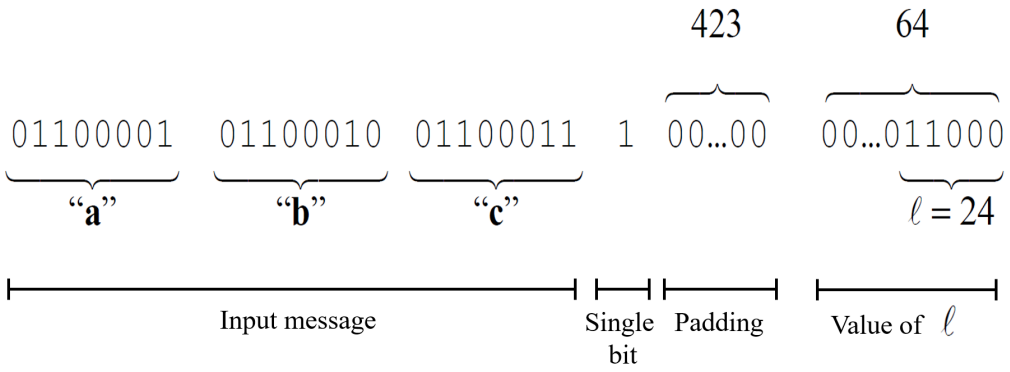

Figure 4.2. Example of padding step in SHA256 [35]

2. Parsing

In the Parsing step, the padded message obtained from the previous step must be divided into N blocks, each one of 512 bits. We call $M^{(i)}$ a block, where in a message i indicates the i-th block, and since a block contains sixteen words we call $M_t^{(i)}$ the t-th word in the i-th block.

Example: Suppose that we have an input message of 980 bits, after the preprocessing step we obtain a message of 1024 bits. The padded message will be parsed in two 512-bits blocks called $M^{(0)}$ and $M^{(1)}$ and for denoting the single words in the i-th block we use the notation $M_0^{(i)}, M_1^{(i)}, M_2^{(i)} \dots M_{15}^{(i)}$.

3. Setting

The Setting step consists in setting the initial hash value $H^{(0)}$ with eight 32-bit words $H_0^{(0)}, H_1^{(0)}, \ldots, H_7^{(0)}$.
The values of these words are known and specified in the SHS [35] as we will see in the next part of this description of algorithm.

4.1.2 Computation

Operations, Functions and Costants

Before continuing with the description of the algorithm, we define the operations and functions necessary for the execution of the second phase called hash computation. All of the following operations are reported in the SHS [35].

Operations Among the various operations the following are the basic ones that operate on words:

- $\wedge$ Bitwise AND operation.
- $\vee$ Bitwise OR operation.
- $\oplus$ Bitwsie XOR operation.
- $\neg$ Bitwsie complement operation (NOT).
- $<<$ Left shift operation.
  $x << n$ is obtained by discarding the left-most n bits of the word x and padding the result with n zeroes on the right.
- $>>$ Right shift operation.
  $x >> n$ is obtained by discarding the right-most n bits of the word x and padding the result with n zeroes on the left.
- $+$ Addition modulo 2^w, where $w = 32$.

We also define the following two operations as follows:

- $ROTR^n(x)$ Rotate right operation.
  It is defined by $ROTR^n(x) = (x >> n) \vee (x << w - n)$, where x is a w-bit word and n is an integer with $0 \leq n < w$.
- $SHR^n(x)$ Right shift operation.
  It is defined by $SHR^n(x) = (x >> n)$

Functions In the hash computation phase of SHA256 algorithm we use six logical functions that operates on words. The result of each function is a new word.
These functions are composed by operations just described and they are defined as follows:

- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
- $\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
- $\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus ROTR^3(x)$
- $\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus ROTR^{10}(x)$

Costants There are two types of constants with known values given in the SHS to be used in the SHA256 algorithm.
The first constants are those used in the setting step from the preprocessing phase, they are eight words denoted by $H_0^(0), H_1^(0), ..., H_7^(0)$.
The second type of constants are a sequence of sixty-four words denoted by $K_0, K_1, ..., K_{63}$; they represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers.

Hash Computation

The hash computation is the main process loop of SHA256 algorithm that operates on padded message parsed in N blocks obtained from preprocessing phase.
Each message block is processed in order (for $i = 1$ to N) using the following four steps:

1. Prepare the message schedule $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq \text{t} \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq \text{t} \leq 63 \end{cases}$$

2. Initialize eights variables called a, b, c, d, e, f, g, h with the $(i-1)^{th}$ hash value:
$a = H_0^{(i-1)}$
$b = H_1^{(i-1)}$
$c = H_2^{(i-1)}$
$d = H_3^{(i-1)}$
$e = H_4^{(i-1)}$
$f = H_5^{(i-1)}$
$g = H_6^{(i-1)}$
$h = H_7^{(i-1)}$

3. Compute the main loop:
For $t = 0$ to 63:
{
$T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_t + W_t$
$T_2 = \Sigma_0(a) + Maj(a, b, c)$
$h = g$
$g = f$
$f = e$
$e = d + T_1$
$d = c$
$c = b$
$b = a$
$a = T_1 + T_2$
}

4. Compute the i^{th} intermediate hash value $H^{(i)}$:
$H_0^{(i)} = a + H_0^{(i-1)}$
$H_1^{(i)} = b + H_1^{(i-1)}$
$H_2^{(i)} = c + H_2^{(i-1)}$
$H_3^{(i)} = d + H_3^{(i-1)}$
$H_4^{(i)} = e + H_4^{(i-1)}$
$H_5^{(i)} = f + H_5^{(i-1)}$
$H_6^{(i)} = g + H_6^{(i-1)}$
$H_7^{(i)} = h + H_7^{(i-1)}$

After repeating steps 1 through 4 a total of N times, the resulting 256-bit digest of the message is computed as:

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}||H_7^{(N)}$$

4.2 Homomorphic evaluation of SHA256

In this section we present the study and analysis that led to our implementation of a homomorphic evaluation of SHA256. For the realization of this implementation, the main tool was the ZAMA's core product, an open-source FHE Compiler called **Concrete**, which simplifies the use of Fully Homomorphic Encryption through a Python API.
We sould point out that the development of this project and all related data refer to two specific versions of compiler that we used: Concrete v1.0 and v2.0 [51].

The analysis performed in the implementation design phase of the project played a key role. Initially, numerous tests were performed to identify the best choice in terms of efficiency, execution speed and occupied memory size, especially with regard to data representation and the interaction with the main internal functions of the algorithm.
All statistics and data reported in the next sections refer to testing and evaluation performed on a personal computer with the following characteristics:

- CPU: AMD Ryzen 5 5500U (4,0 GHz boost frequency, 8MB cache L3, 6 core)
- RAM: 8GB DDR4-3200 MHz (2×4 GB)

4.2.1 Design Choices and Analysis

Data Representation

All the operations and functions of the algorithm operate on a specific unit of data, which we have called *word*, of dimension $w = 32$ bits. There is therefore a main problem that needs to be solved: homomorphic version of SHA256 needs a change of data representation given that Concrete does not support 32 bits Integer types and related functions that can operate on them, but is only possible to operate with Integer variables of length at most 16 bits.
As a solution to this problem, we decided to decrease the unit of data on which to perform operations by dividing a single *word* into smaller pieces called **chunks**. In the implementation we define this variables as follows:

- `CHUNK_SIZE`: The size of a chunk that can be $2, 4, 8$ or 16 bits.
- `CHUNK_NUM`: The number of chunks in a word, computed as `CHUNK_NUM`$= w/$`CHUNK_SIZE`.

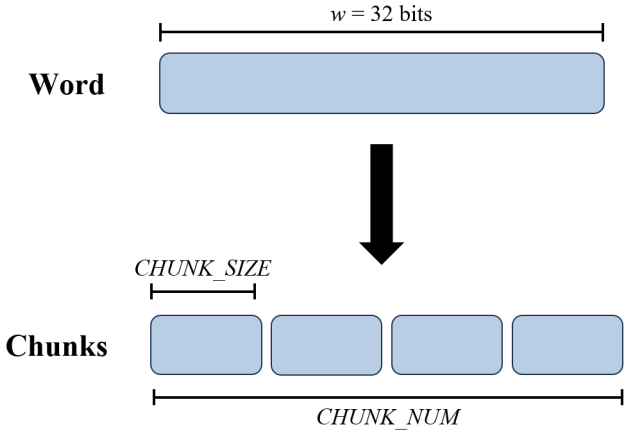

Figure 4.3. Chunks representation

Analysis and Comparisons

The choice of the chunk size is very important because it affects the final parameters of the circuit that we saw in Chapter 3. We can observe how the circuit parameters change as the `CHUNK_SIZE` value changes by performing the operations used inside SHA256. The operations chosen for the analysis are $ROTR^n(x)$, $SHR^n(x)$ and + (Addition modulo 2^{32}).
The circuit parameters considered are the following:

- *Nodes*: Number of nodes of the circuit's graph.
- *GenKey Time*: Generation key time.
- *Circuit Time*: It is composed by *Compile Circuit Time* and *Execution Circuit Time*.

In the following three tables we present the results of this analysis for the previously chosen operations:

$ROTR^n(x)$			
`CHUNK_SIZE`	Nodes	GenKey Time	Circuit Time
16 bit	43	2,7 s	80,21 s
8 bit	83	2,8 s	106,13 s
4 bit	163	2,3 s	1,6 s
2 bit	323	1,61 s	2,05 s

Table 4.1. Table test on $ROTR^n(x)$

$SHR^n(x)$			
CHUNK_SIZE	Nodes	GenKey Time	Circuit Time
16 bit	45	2,6 s	88,33 s
8 bit	87	2,4 s	107,56 s
4 bit	169	2,5 s	1,7 s
2 bit	326	1,8 s	2,1 s

Table 4.2. Table test on $SHR^n(x)$

+ (Addition modulo 2^{32})			
CHUNK_SIZE	Nodes	GenKey Time	Circuit Time
16 bit	42	3,1 s	12,2 s
8 bit	85	2,8 s	17,7 s
4 bit	168	3,1 s	0,7 s
2 bit	324	2,2 s	0,8 s

Table 4.3. Table test on Modular Addition

In order to have a simpler implementation, the best choice would be to use the maximum possible value of CHUNK_SIZE, but in this case we would have a very high total execution time, so the options of 16 bit and 8 bit are not considered. We initially decided to work with CHUNK_SIZE= 4 bit, considering the good ratio observed between number of nodes and circuit time.
We present, in the following table, the resulting circuit parameters for the functions of the algorithm (see 4.1.2) performed with 4-bit chunks:

CHUNK SIZE = 4 bit			
Function	Nodes	GenKey Time	Circuit Time
Σ_0	558	3,3 s	7,6 s
Σ_1	558	3,9 s	33,5 s
σ_0	563	3,5 s	16,9 s
σ_1	571	3,6 s	30,7 s
Maj	165	3,4 s	5,5 s
Ch	132	3,3 s	3,4 s

Table 4.4. Table test on Functions

At this point we can observe that the time to generate the keys is independent from the functions considered, while the circuit time is directly proportional to the complexity of the implementation of the function it refers to.
With complexity of the implementation we mean the number and type of code operations that were required to write a given function into the Python language code. All the details and final versions of the implementations will be discussed in section 4.2.2.

The total complexity is also due to the fact that almost all operations in the Python code must make use of LUTs, as we saw in Chapter 3, to ensure proper execution of the homomorphic circuit. The level of complexity can be considerably lowered by providing an implementation in which native operations (e.g., sum) can be used directly instead of operations that make use of LUTs (e.g., AND or XOR). Following the previous observations, we made changes to the first versions of the function implementations, obtaining an improvement on number of nodes and circuit time. In the following table we show the differences and advantages obtained from the changes made on the three operations analyzed earlier based on 4-bit chunks implementation.

CHUNK SIZE = 4 bit		
Function	Nodes	Circuit Time
$ROTR^n(x)$	163	1,6 s
$ROTR^n(x)$ (modified)	17	0,45 s
$SHR^n(x)$	169	1,7 s
$SHR^n(x)$ (modified)	24	0,47 s
+	168	0,7 s
+ (modified)	67	0,5 s

Table 4.5. Table test on modified operations

During the development phase of this project there was a major update to the compiler Concrete (ZAMA released version 2.0) making internal improvements to the structure of the compiler. This upgrade to a new version allowed us to make new comparisons on the implementation aspects, and we came to the conclusion that it is now better to use 2-bit chunks since it provides better performances than what was possible in the previous version. We will present the performance comparisons of the project completed between the different compiler versions in the final section of this chapter.

Implementing Preprocessing

Since the internal structure is based on 2-bit chunks we represent a word with an array of 16 elements, each one of 2 bit. In this case the parameters for data representation have value `CHUNK_SIZE`=2 and `CHUNK_NUM`=16.
The code that we show below and in the next examples is based on an implementation of the SHA256 algorithm that can take in input up to 959 bits (we refer to the max number of bits that a message can have to be processed in two blocks).
The resulting padded message M after preprocessing phase will be two blocks large and will have size $512 \times 2 = 1024$ bits, where M is represented by a matrix 32×16 of 2-bit elements.

The `preprocessing` function encapsulates three other functions as we can see from the code:

```
import numpy as np
####################  PREPROCESSING  ####################
def preprocessing(text):
    input = input_list(text)
    input_padded = padding(input)
    input_parsed = parsing(input_padded)
    arr = np.array(input_parsed)
    return arr
```

Listing 4.1. Preprocessing Function

The `input_list` function takes the initial input message and converts it to a list formatting elements to facilitate the following operations. The `padding` and `parsing` functions implement the steps required by the algorithm (4.1.1) by padding the input message and dividing it into blocks with 2-bit elements.

```
####################  INPUT LIST  ####################
def input_list(data):
    inp = list(data)
    out = []
    for item in inp:
        out.append(item >> 6)
        out.append(item >> 4 & 0b_0000_0011)
        out.append(item >> 2 & 0b_0000_0011)
        out.append(item & 0b_0000_0011)
    return out
####################  PADDING  ####################
def padding(msg):
    len_msg= len(msg)
    len_bits_64 = len_msg * 2
    num_blocks = 2
    last_64 = []
    for i in range(0, 32):
        last_64.append(len_bits_64 >> (62 - i*2) & 0b_0011)
    len_last_64 = len(last_64)
    # 1. Append the bit '1' to the message
    msg.append(0b_10)
    # 2. Append k '0' bits, where k is the minimum number >= 0
    # such that the resulting message length (in bits) is
   congruent to 448 (mod 512)
    for _ in range(256*num_blocks - len_msg - len_last_64-1):
        msg.append(0b_00)
    # 3. Append length of message (before pre-processing), in bits
   , as 64-bit big-endian integer
    for i in range(len_last_64):
        msg.append(last_64[i])
    return msg
####################  PARSING  ####################
```

```
31 def parsing_2bit(block):
32     M = []
33     for i in range(0, len(block), 16):
34         M.append(block[i:i+16])
35     return M
```

Listing 4.2. InputList Padding and Parsing Function

In this implementation, the preprocessing phase is performed in clear mode, the data is not encrypted since Concrete does not currently support sending multiple encrypted parameters, which would have been needed in the Client-Server architecture that we will see in section 4.3.
The matrix `M`, representing the padded message after the preprocessing phase, will be the initial input to the circuit for homomorphic evaluation. In the next section we see in detail the implementations of the operations and functions of the algorithm.

4.2.2 Operations and Functions

The operations and functions of the algorithm must be implemented by adapting them according to the internal structure chosen with 2-bit chunks and also using already implemented that should be compatible with Concrete, since the Python API that is used in Concrete supports some of the functions that we can found in the famous python library called Numpy [21].

We can divide the operations to be implemented in three groups that we will call in the following way: bitwise operations, manipulation operations, and functions. The bitwise operations are AND, OR, XOR, NOT and we can use them directly because they were introduced in Concrete v1.0. Only the NOT operation is not supported, but the problem is easily solved by implementing it using the XOR operation as we can see:

```
1 import numpy as np
2 from concrete import fhe
3 ############### NOT OPERATION ###############
4 def not_bitwise(word):
5     word_result = fhe.zeros(CHUNK_NUM)
6     word_result = np.bitwise_xor(word, 0b_11)
7     return word_result
```

Listing 4.3. NOT Operation

The manipulation operations includes $ROTR^n(x)$, $SHR^n(x)$ and + (Addition modulo 2^{32}) which must be specially adapted for the implementation. The functions are the Sigmas functions, Maj and Ch which are composed of all the operations just described.

Rotate Right Operation ($ROTR^n(x)$)

The $ROTR^n(x)$ consists in shifting all the bits of the word x to the right by n positions, with the n rightmost bits becoming the leftmost bits after the operation. Each bit of the word moves one step to the right and the bit that was initially at the end of the word is moved to the first position.
For easy representation in the following figure we show the $ROTR^n(x)$ operation on a 4-bit string x with the rotation of $n = 1$ bit:

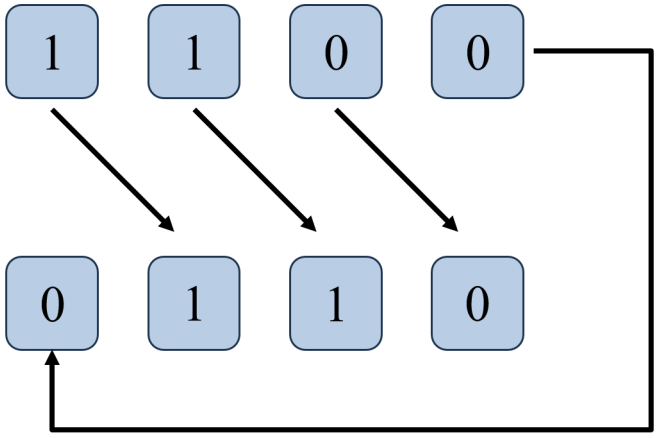

Figure 4.4. $ROTR^1(x)$ on 4-bit string

We cannot directly implement the definition $ROTR^n(x) = (x >> n) \vee (x << w - n)$ given in section 4.1.2 because we cannot directly operates on 32-bit words. Considering that a word x is represented by an array of 16 2-bit chunks, we can implement the $ROTR^n(x)$ operation by distinguishing two cases, when n is even there will be only a chunk rotation, while when n is odd there will be a chunk rotation and a bit rotation.

1. In a rotation $ROTR^n(x)$ if n is multiple of `CHUNK_NUM`=2 will result in rotating the array of chunks.

2. In a rotation $ROTR^n(x)$ if n is not multiple of `CHUNK_NUM`=2, after the step 1 there will be a rotation of `CHUNK_NUM`$-n($ mod 2) bits.

In the function `rotr` the word x and the number of positions n are respectively represented by the variables `word` and `N`. In the following code segment we can see the chunk rotation implemented by a reassignment of positions of the elements of an array and the bits rotation by a sequence of changes on the array using masks and bitwise operations.

```
from concrete import fhe
################ ROTR OPERATION ###############
def rotr(word, N):
    bits_rot = N % CHUNK_SIZE
    blocks_rot = (N // CHUNK_SIZE)
    v = fhe.zeros(CHUNK_NUM)
    v2 = fhe.zeros(CHUNK_NUM)
    mask = fhe.zeros(CHUNK_NUM)
    mask2 = fhe.zeros(CHUNK_NUM)
    #Chunks Rotation
    if(blocks_rot > 0):
        v[0:blocks_rot] = word[(CHUNK_NUM-blocks_rot):CHUNK_NUM]
    v[blocks_rot:CHUNK_NUM] = word[0:(CHUNK_NUM-blocks_rot)]
    #Bits Rotation
    if(bits_rot != 0):
        mask2 = (v & 0b_01) * 0b_10
        mask[1:] = mask2[:-1]
        mask[0] = mask2[CHUNK_NUM-1]
        v2 = v // 2
        result = (v2 + mask)
    return result
```

Listing 4.4. ROTR Operation

To give an idea of the steps that this function performs we present this example: the 2-bit chunks representation of a given word x = "abcd" (8-bit ASCII) extracted from the message resulting from preprocessing would be an array `word` = [1,2,0,1,1,2,0,2,1,2,0,3,1,2,1,0]. We want perform on word x a rotate right operation of 7 positions ($ROTR^7(x)$).

In the next image we present the intermediate values of function `rotr(word,7)` following the previously code implementation:

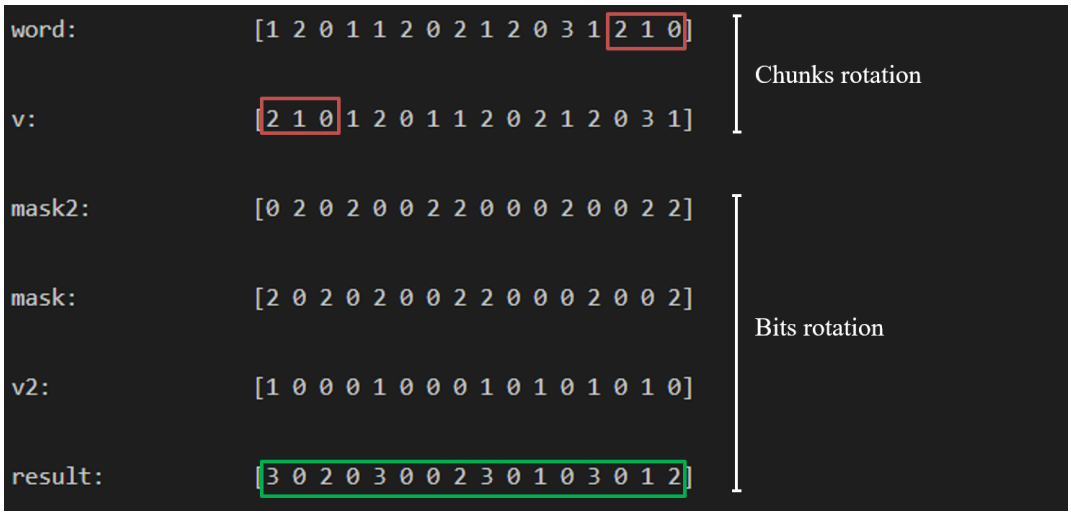

Figure 4.5. Example of Rotate Right Operation

Shift Right Operation ($SHR^n(x)$)

Once the implementation of the `rotr` operation is defined, the Shift Right Operation is simple.
The $SHR^n(x)$ implementation (`shr(word,N)`) consists of setting all the last n positions to value "0" of a word x and then performing a rotate right operation as shown in the following code:

```
import numpy as np
from concrete import fhe
############### SHR OPERATION ###############
def shr(word, N):
    blocks_rot = (N // CHUNK_SIZE)
    bits_rot = N % CHUNK_SIZE
    v = fhe.zeros(CHUNK_NUM)
    #Chunks initialization
    v[:CHUNK_NUM-blocks_rot] = word[:CHUNK_NUM-blocks_rot]
    #Last bits to zero
    if(bits_rot == 1):
        v[CHUNK_NUM-blocks_rot-1] = np.bitwise_and(word[CHUNK_NUM
   -blocks_rot-1],0b_10)
    #Rotation of zeros -> Shift to right
    result = rotr(v, N)
    return result
```

Listing 4.5. SHR Operation

Modular Addition (+)

In SHA256 algorithm the modular 32-bit addition is frequently used in functions and intermediate steps. Concrete supports directly addition of 32-bit numbers, but the management of parameters of this size is totally inefficient. For this reason, we implemented a modular addition that operates directly on 2-bit chunks of the words so as to greatly decrease the execution time.

Considering that we must perform the modular addition (`addition_mod`) of `word1` and `word2`. Starting with the chunks in the last position (`CHUNK_NUM`-1) of `word1` and `word2`, we sum them directly, obtaining a result that could be a 3-bit chunk.
At this point we copy the bit in the first position of the resulting chunk and add it, as if it were a carry, to the next chunks sum, and so on until we arrive at position `CHUNK_NUM`=0 of the array.
At the end the resulting array, that might contains 3-bit chunks, is put in AND operation with a mask (where each chunk has value "11" in binary representation) to get the final result.
With the following segment code we show the implementation:

```
import numpy as np
from concrete import fhe
################ MODULAR ADDITION ################
def addition_mod(word1, word2):
    word_result = fhe.zeros(CHUNK_NUM)
    carry = 0b_00
    for i in range(CHUNK_NUM-1, -1, -1):
        word_result[i] = ((word1[i] + word2[i])+carry)
        carry = word_result[i]
        carry = carry // 4
    word_result = np.bitwise_and(word_result, 0b_11)
    return word_result
```

Listing 4.6. Modular Addition

The following figure illustrates the process of Modular Addition on 2-bit chunks:

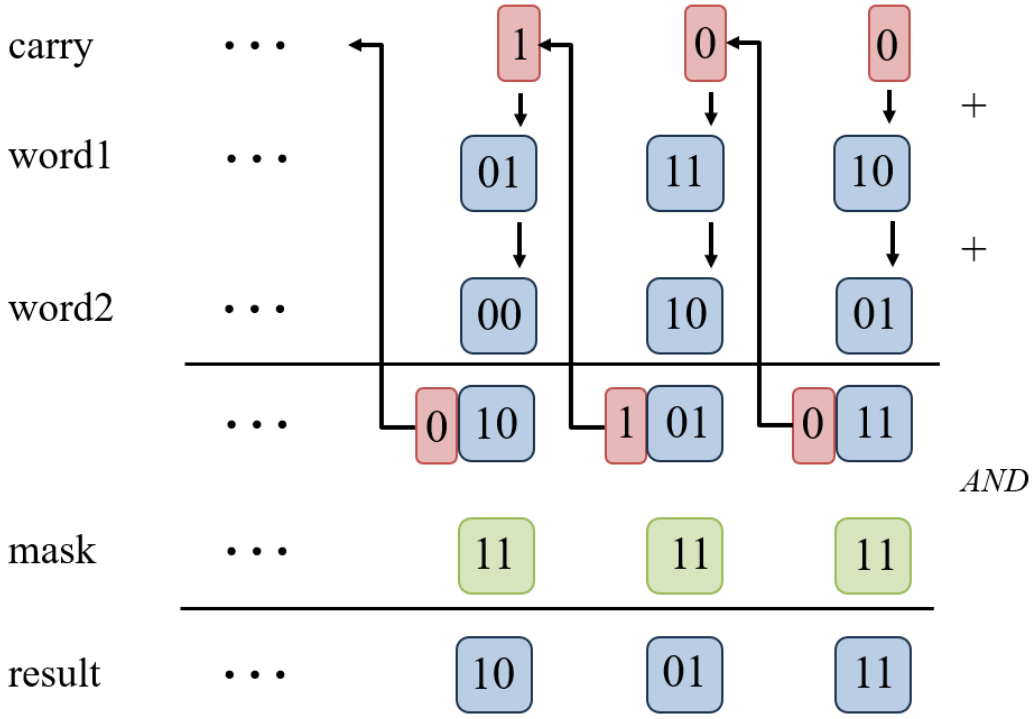

Figure 4.6. Modular Addition on 2-bit chunks

Functions

We defined the functions used in SHA256 algorithm in section 4.1.2 which are the four functions $\Sigma_0(x)$, $\Sigma_1(x)$, $\sigma_0(x)$, $\sigma_1(x)$ that mainly use $ROTR^n(x)$ and $SHR^n(x)$ operations and the two functions $Ch(x)$, $Maj(x)$ that are composed by bitwise operations.
At this point we can implement all these functions using the implemented operations previously described.

```
import numpy as np

############### FUNCTIONS SHA256 ###############
def SIGMA_0(x):
    res = np.bitwise_xor((np.bitwise_xor(rotr(x, 2), rotr(x, 13)))
    , (rotr(x, 22)))
    return res

def SIGMA_1(x):
    res = np.bitwise_xor((np.bitwise_xor(rotr(x, 6), rotr(x, 11)))
    , (rotr(x, 25)))
    return res

def sigma_0(x):
    res = np.bitwise_xor((np.bitwise_xor(rotr(x, 7), rotr(x, 18)))
    , (shr(x, 3)))
    return res

def sigma_1(x):
    res = np.bitwise_xor((np.bitwise_xor(rotr(x, 17), rotr(x, 19))
    ), (rotr(x, 10)))
    return res

def ch(x, y, z):
    res = np.bitwise_xor((np.bitwise_and(x, y)), (np.bitwise_and(
    not_bitwise(x), z)))
    return res

def maj(x, y, z):
    res = np.bitwise_xor(np.bitwise_xor((np.bitwise_and(x, y)), (
    np.bitwise_and(x, z))), (np.bitwise_and(y, z)))
    return res
```

Listing 4.7. Functions of SHA256 Algorithm

4.2.3 SHA256 on Encrypted Input

Implementing Hash Computation

The last phase of SHA256 algorithm is the main loop where the blocks are processed, called *Hash Computation*. As we see in section 4.1.2, the Hash Computation phase is divided in 4 steps whose implementation is shown in the following segment code:

```
from functions import *
from costants import *
from concrete import fhe

def function_sha256(M_list):

# 1. Prepare the message schedule {W[t]}
    W_list = fhe.zeros((64,CHUNK_NUM))
    H_list = fhe.zeros((8,CHUNK_NUM))
    K_list = fhe.zeros((64,CHUNK_NUM))
    H_list[:] = H_2bit[:]
    K_list[:] = K_2bit[:]

    for i in range(0,2): #2 Blocks
# 2. Initialize the eight working variables a, b, c, d, e, f, g, h
    with the (i-1)-st hash value
        a = H_list[0]
        b = H_list[1]
        c = H_list[2]
        d = H_list[3]
        e = H_list[4]
        f = H_list[5]
        g = H_list[6]
        h = H_list[7]

        W_list[0:16] = M_list[(0+16*i):(16+16*i)]

# 3. Computation for the 64 rounds
        for t in range(0,64):
            if (t >=16):
                W_list[t] = W_comp(W_list, t)

            t_1 = addition_mod(addition_mod(addition_mod(
    addition_mod(h, SIGMA_1(e)), ch(e, f, g)), K_list[t]),W_list[t
    ])
            t_2 = addition_mod(SIGMA_0(a), maj(a, b, c))
            h = g
            g = f
            f = e
            e = addition_mod(d, t_1)
            d = c
```

```
            c = b
            b = a
            a = addition_mod(t_1, t_2)

# 4. Compute the final step of a 512-bits block of N blocks (in
    this case N=2)
        H_list[0] = addition_mod(a, H_list[0])
        H_list[1] = addition_mod(b, H_list[1])
        H_list[2] = addition_mod(c, H_list[2])
        H_list[3] = addition_mod(d, H_list[3])
        H_list[4] = addition_mod(e, H_list[4])
        H_list[5] = addition_mod(f, H_list[5])
        H_list[6] = addition_mod(g, H_list[6])
        H_list[7] = addition_mod(h, H_list[7])

    return H_list
```

Listing 4.8. Hash Computation

In our implementation the function `function_sha256` represent the Hash Computation phase and take in input the padded message `M_list` divided in blocks to execute the 4 steps:

1. Initialization of the main variables where `H_2bit` and `K_2bit` are the costants used in the algorithm already in 2-bit chunks format.
2. Initialization of the eight working variables and assignment of values to the first 16 elements of message schedule `W_list`. From this step begins the cycle for N blocks to be processed.
3. Computation of main loop where we have implemented the function `W_comp` in the following way:

```
###### FUNCTION TO CREATE THE t-th ELEMENT OF W ######

def W_comp(w, t):
        Wt = addition_mod(addition_mod(addition_mod(sigma_1(w[
    t-2]), w[t-7]), sigma_0(w[t-15])), w[t-16])
        return Wt
```

Listing 4.9. Schedule Message W Computation

4. Final step where at the N-th computation it will results the final hash value.

4.2.4 Final Results

At this point we are able, through the Python API of Concrete, to perform an homomorphic evaluation of SHA256 with the implemented functions that we have described.
The main code where the circuit is created and executed is the following:

```
from concrete import fhe
import numpy as np
from utils import preprocessing,hex_convert
from sha256 import function_sha256

############              Input            ############
M_list = preprocessing(text)

############     Circuit Configuration    ############
configuration = fhe.Configuration(
    enable_unsafe_features=True,
    use_insecure_key_cache=True,
    insecure_key_cache_location=".keys",
)
inputset = [(np.random.randint(0, 2 ** 2, size=(32,16))) for _ in
    range(100)]

############      Circuit Execution       ############
def main_function(data):
    res = function_sha256(data)
    return res
compiler = fhe.Compiler(main_function, {"data": "encrypted"})
#Compile
circuit = compiler.compile(inputset, configuration)
#Generation Keys
circuit.keygen()
#Run Circuit
result = circuit.encrypt_run_decrypt(M_list)
#Final Result
hash_result = hex_convert(result)
```

Listing 4.10. Main in Python API Concrete

In this case the variable `text` is the input which has been preprocessed in clear mode at the beginning, the resulting variable `M_list` is the input for the execution circuit step.
First of all we configure the standard parameters of the circuit in `configuration` variable and we define the necessary `inputset` to represent the encrypted input. We declare with the `fhe.Compiler` command the `main_function` that we use in the circuit and the number and type of input data (`{ "data":"encrypted"}`). The `compiler.compile` command performs the compile step of the circuit, that is the largest in terms of execution time and memory space usage.
Once the circuit has been compiled we generate the keys with the `circuit.keygen()` command and we perform the circuit execution on encrypted input with `encrypt_run_decrypt(M_list)`. At the end we convert the resulting string with `hex_convert` in hexadecimal representation.

We analyzed the performance of steps in the circuit and found that the two most interesting parameters are compile time and execution time. During the development of the project, we performed comparison tests on three different implementations that differ in internal structure and compiler version used. These analyses are based on a one block execution of sha256 and the three versions are 4-bit chunks Concrete v.1, 2-bit chunks Concrete v.1 and 2-bit chunks Concrete v.2. The choice of changing the internal structure from 4-bit to 2-bit chunks is due to the fact that the parameter of greatest interest is execution time. Compile time is resulting to be the parameter with the largest value, but as we will see in Section 4.3.1, it can be completely separated from circuit execution. We decided to decrease the number of bits in the chunks because even though the compile time has a growth, it results in significantly better times for the execution time.
Finally the upgrade from Concrete v.1 to Concrete v.2 resulted in a general improvement in performance due to internal changes of the compiler.
We show the comparison between these three implementations in the following table:

Comparison implementations 1-block SHA256			
Performance	4-bit v.1	2-bit v.1	2-bit v.2
Compile Time	19085 s	27692 s	13282 s
GenKey Time	12,4 s	2,03 s	1,7 s
Execution Time	1701 s	806 s	593 s
Total Time	20799 s	28500 s	13877 s

Table 4.6. Comparison implementations 1-block SHA256

Another parameter to consider is the memory space used during the compilation step. In Concrete compiler a circuit with such an amount of operations, performed via Python API, is extremely heavy, coming to limit the possibility to compiling on personal computers. We show you the results obtained and all the performances of major interest from the two our final implementations of homomorphic evaluation of SHA256 (based on 2-bit chuncks Concrete v.2) operating on 1 block and 2 blocks in the following table.

Homomorphic Evaluation of SHA256 Performance		
Performance	1 block	2 blocks
Plaintext Input	$<$ 448 bit	$<$ 960 bit
Compile Time	3,7 hours	12,2 hours
GenKey Time	1,7 seconds	3 s
Execution Time	10 minutes	27 minutes
Memory Usage	4 GB	12 GB

Table 4.7. Homomorphic Evaluation of SHA256 Performance

4.3 Client-Server Architecture

4.3.1 SHA256 precompiled implementation

In our implementation we saw that the compilation step takes up to 96% of the total amount of time required to execute the program.
One possible solution to alleviate this problem is the separation of the compilation and execution steps, in fact in Concrete it is possible to save a precompiled file of the entire circuit. It is possible to execute the circuit directly just by calling the precompiled file and providing the necessary inputs. In this way, once the function is defined, we can directly execute it homomorphically.

This solution is extremely useful a client-server communication: in this case, the server can save the precompiled file and execute the circuit every time a client sends a request.
The structure of the code that allows to compile and save a precompiled file is the same as seen above; the only difference is that after compiling the file we do not proceed with the execution commands but we save the file through the `circuit.save("sha256.zip")` command, where "sha256.zip" will be the name of the file.

```
from concrete import fhe
import numpy as np
from utils import preprocessing
from sha256 import function_sha256
############             Input           ############
M_list = preprocessing(text)
############     Circuit Configuration   ############
configuration = fhe.Configuration(
    enable_unsafe_features=True,
    use_insecure_key_cache=True,
    insecure_key_cache_location=".keys",
)
inputset = [(np.random.randint(0, 2 ** 2, size=(32,16))) for _ in
   range(100)]
############      Circuit Execution      ############
def main_function(data):
    res = function_sha256(data)
    return res
compiler = fhe.Compiler(main_function, {"data": "encrypted"})
#Compile
circuit = compiler.compile(inputset, configuration)
#Save precompiled file
circuit.server.save("sha256.zip")
```

Listing 4.11. Precompilation step

4.3.2 Client-Server Communication

We want homomorphically evaluate our implementation of SHA256 in a client-server communication. The client will send an encrypted input to the server, which will have to perform SHA256 without any knowledge of the input, and return to the client a ciphertext that, when decrypted, will result in the final hash value.
We implemented a client-server communication in Python via sockets over TCP protocol where we can perform homomorphically SHA256 using Concrete features designed specifically for this [50].

We now present the main steps of communication by going through the actions performed by the Client and Server in order. The following examples are code extracts from our implementation.

1. *Server*

```
#Initialization
server_fhe = fhe.Server.load("sha256.zip")
#Specific Client
serialized_client_specs: str = server_fhe.client_specs.
    serialize()
#Send SERVER -> CLIENT
server.sendall(serialized_client_specs)
```

Listing 4.12. Server Step 1

The `server_fhe` is initialized by loading the precompiled file and it serializes and sends the specifications that are needed for the initialization of the client.

2. *Client*

```
#Initialization
client_specs = fhe.ClientSpecs.deserialize(
    serialized_client_specs)
client_fhe = fhe.Client(client_specs)
#Keys Generation
client_fhe.keys.generate()
#Serialization Keys and Input
serialized_evaluation_keys: bytes = client_fhe.evaluation_keys
    .serialize()
serialized_args: bytes = client_fhe.encrypt(M_2bits).serialize
    ()
#Send CLIENT -> SERVER
client.sendall(serialized_evaluation_keys)
client.sendall(serialized_args)
```

Listing 4.13. Client Step 1

The `client_fhe` is initialized using the specifications `client_specs` sended by the server.
The `client_fhe.keys.generate()` command generates two types of key:

- A *Master Key* known only by client necessary for initial encryption of the input and final decryption of the result.
- Evaluation Keys (`evaluation_keys`) necessary to the server to perform homomorphically the function.

The `client_fhe` encrypts the input `M_list` and serializes `evaluation_keys` and the encrypted input sending them to the server.

3. *Server*

```
#Deserialization
deserialized_evaluation_keys = fhe.EvaluationKeys.deserialize(
    serialized_evaluation_keys)
deserialized_args  = server_fhe.client_specs.
    deserialize_public_args(serialized_args)
#Running Circuit
public_result = server_fhe.run(deserialized_args,
    deserialized_evaluation_keys)
#Serialize
serialized_public_result: bytes = public_result.serialize()
#Send SERVER -> CLIENT
server.sendall(serialized_public_result)
```

Listing 4.14. Server Step 2

After deserializing the data sent by the client, the server executes the circuit with the command `server_fhe.run` and sends the encrypted result to the client.

4. *Client*

```
#Deserialization
deserialized_public_result = client_fhe.specs.
    deserialize_public_result(serialized_public_result)
#Decrypt Result
result = client_fhe.decrypt(deserialized_public_result)
#Hash value
hash_result = hex_convert(result)
```

Listing 4.15. Client Step 2

The client decrypts `deserialized_public_result` and converts it in hexadecimal format with `hex_convert` obtaining the final hash value.

We show the complete Client-Server communication procedure with the following figure:

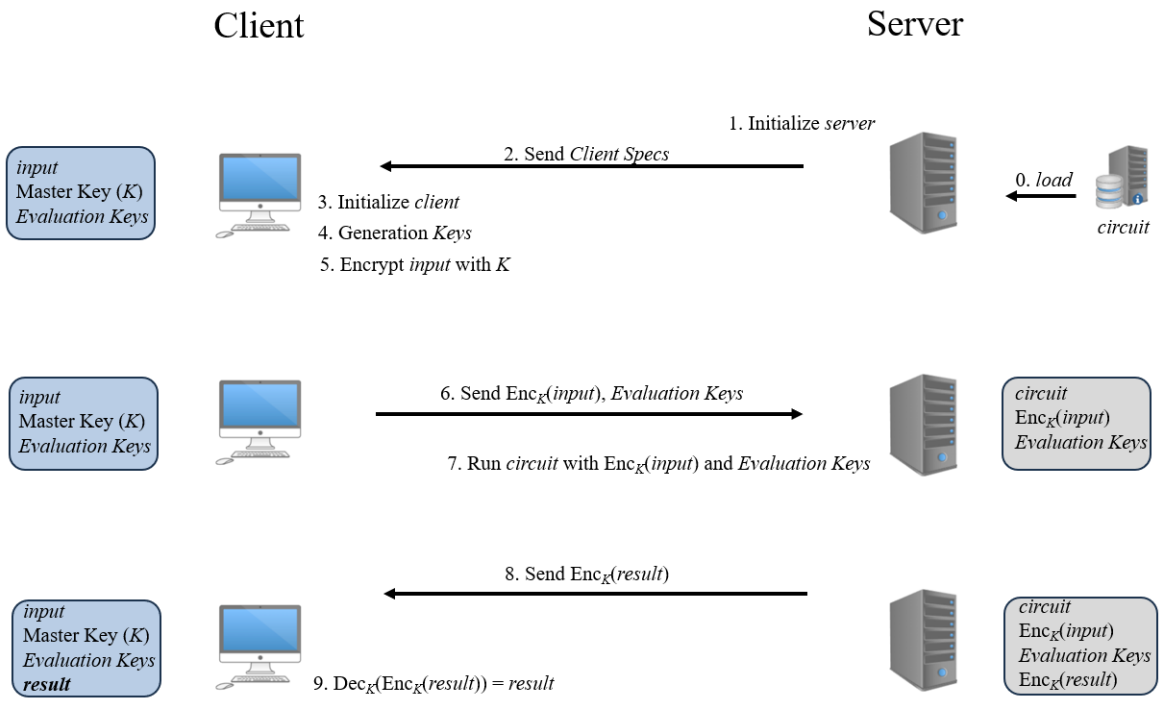

Figure 4.7. Homomorphic SHA256 through Client-Server Communication

With the implementation performance seen in the previous section this example of communication is not yet suitable for the real world. Nevertheless, thanks to the continuous improvements and feature updates of ZAMA on Concrete compiler it is possible to think about exploiting this implementation as a basis to more complex cryptographic algorithms such as Message Authentication Codes or to be used in Digital Signatures.

Conclusions

In this thesis we designed and developed a homomorphic evaluation of the hash function SHA256, using the ZAMA Concrete Compiler.
Firstly, we presented a general overview of the cryptographic protocol called Fully Homomorphic Encryption (FHE). We started with the general concepts of cryptography and then we delved into the concept of Homomorphic Encryption (HE) showing how it has evolved over time by providing a general description of its main features and referring to schemes proposed in the past years.

Secondly, we described the implementation choices that characterized the development work. We focused on TFHE scheme of ZAMA and we used the Concrete compiler to develop the final project of this thesis. By developing a homomorphic evaluation of the hash function SHA256, we observed and analyzed the potential that Concrete offers to developers and the limitations to which it is still subjected. Research on FHE and the development of products such as Concrete that offer the ability to use this protocol in complex applications is continuosly increasing.

We agree that FHE is an extremely powerful tool for ensuring data security and privacy for users in a society that nowadays makes data a huge source of power. At the time of the development of this thesis there are technical limitations and this technology is not mature enough to be able to be applied to very complex applications with a high degree of efficiency.
However, in the last few years FHE had significant improvements and demonstrated great promise.

List of Figures

List of Tables

Listings

Acronyms

NSA *National Security Agency*

AES *Advanced Encryption Standard*

DES *Data Encryption Standard*

3DES *Triple DES*

RC4 *Rivest Cipher 4*

NIST *National Institute of Standards and Technology*

RSA *Rivest Shamir Adleman*

SHA *Secure Hash Algorithm*

PHE *Partially Homomorphic Encryption*

SWHE *Somewhat Homomorphic Encryption*

FHE *Fully Homomorphic Encryption*

TFHE *Fully Homomorphic Encryption over the Torus*

LWE *Learning With Errors*

RLWE *Ring-LWE*

SEAL *Simple Encrypted Arithmetic Library*

SHS *Secure Hash Standard*

LUTs *Look Up Tables*

MLIR *Multi-Level Intermediate Representation*

Bibliography

[1] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2018.

[2] Elaine Barker. *Recommendation for Key Mangement: Part 1 - General*. NIST Special Publication 800-57 Part 1 Revision 5. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf.

[3] D. Bazzanella. *Slides Course Blockchain e criptoeconomia*. 2022.

[4] Eu-Jin Goh Boneh Dan and Kobbi Nissim. «Evaluating 2-DNF formulas on ciphertexts». In: *Theory of cryptography* (2005), pp. 325–341.

[5] Paolo Caressa. *Cos'è la Crittografia?* URL: http://www.caressa.it/pdf/crypto.pdf.

[6] National Cyber Security Centre. *Preparing for Quantum-Safe Cryptography*. URL: https://www.ncsc.gov.uk/pdfs/whitepaper/preparing-for-quantum-safe-cryptography.pdf.

[7] Jung Hee Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Paper 2016/421. 2016. URL: https://eprint.iacr.org/2016/421.

[8] Ilaria Chillotti. *TFHE Deep Dive*. URL: https://cdn.fhe.org/slides/tfhe_deep_dive_ilaria_chillotti.pdf.

[9] Ilaria Chillotti. *TFHE Deep Dive*. URL: https://www.youtube.com/watch?v=LZuEr4jpyUw.

[10] Ilaria Chillotti. *TFHE Deep Dive - Part I - Ciphertext types*. URL: https://www.zama.ai/post/tfhe-deep-dive-part-1.

[11] Ilaria Chillotti. *TFHE Deep Dive - Part IV - Programmable Bootstrapping*. URL: https://www.zama.ai/post/tfhe-deep-dive-part-4.

[12] Ilaria Chillotti et al. *Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds*. Cryptology ePrint Archive, Paper 2016/870. 2016. URL: https://eprint.iacr.org/2016/870.

[13] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption over the Torus*. Cryptology ePrint Archive, Paper 2018/421. 2018. URL: https://eprint.iacr.org/2018/421.

[14] Abbas Acar; Hidayet Aksu; A. Seluk Uluagac; Mauro Conti. *A Survey on Homomorphic Encryption Schemes: Theory and Implementation*. 2017.

[15] W. Diffie and M.E. Hellman. «Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard». In: *Computer* 10.6 (1977), pp. 74–84. DOI: 10.1109/C-M.1977.217750.

[16] T. Elgamal. «A public key cryptosystem and a signature scheme based on discrete logarithms». In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074.

[17] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144. 2012. URL: https://eprint.iacr.org/2012/144.

[18] Craig Gentry. *Computing Arbitrary Functions of Encrypted Data*. 2009. URL: https://dl.acm.org/doi/10.1145/1666420.1666444.

[19] Craig Gentry, Amit Sahai, and Brent Waters. *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based*. Cryptology ePrint Archive, Paper 2013/340. 2013. URL: https://eprint.iacr.org/2013/340.

[20] Shafi Goldwasser and Silvio Micali. «Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information». In: (1982), pp. 365–377. URL: https://doi.org/10.1145/800070.802212.

[21] Charles R. Harris et al. «Array programming with NumPy». In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[22] IBM. *IBM FHE Cloud Service (Beta)*. URL: https://he4cloud.com/public.

[23] IBM. *IBM Releases Fully Homomorphic Encryption Toolkit for MacOS and iOS; Linux and Android Coming Soon*. URL: https://research.ibm.com/blog/ibm-releases-fully-homomorphic-encryption-toolkit-for-macos-and-ios-linux-and-android-coming-soon.

[24] IBM. *Repository Github HElib*. URL: https://github.com/homenc/HElib.

[25] Marc Joye. *Homomorphic Encryption 101*. URL: https://www.zama.ai/post/homomorphic-encryption-101.

[26] Auguste Kerckhoffs. *LA CRYPTOGRAPHIE MILITAIRE*. 1883. URL: https://www.petitcolas.net/kerckhoffs/crypto_militaire_2.pdf.

[27] Swati Sharma; Loveneet; Saba Khanum. *Performance analysis of SHA 2 and SHA 3*. 2022. URL: http://dx.doi.org/10.2139/ssrn.4068861.

[28] Chris Lattner et al. «MLIR: Scaling Compiler Infrastructure for Domain Specific Computation». In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[29] Daniele Micciancio Lèo Ducas. «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* (2015), pp. 617–640.

[30] Marc Stevens; Ellie Bursztein; Pierre Karpman; Ange Albertini; Yarik Markov. *The first collision for full SHA-1*. URL: https://shattered.io/static/shattered.pdf.

[31] R. Merkle and M. Hellman. «On the Security of Multiple Encryption». In: *Communications of the ACM* 24.7 (1981), pp. 465–467.

[32] Microsoft. *SEAL*. URL: https://www.microsoft.com/en-us/research/project/microsoft-seal/.

[33] NIST. *Advanced Encryption Standard (AES)*. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[34] NIST. *Glossary*. URL: https://csrc.nist.gov/glossary/term/cryptography.

[35] NIST. *Secure Hash Standard (SHS)*. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[36] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf.

[37] Menezes Alfred J.; van Oorschot Paul C.; Vanstone Scott A. *Handbook of Applied Cryptography*. CRC Press, 2018.

[38] Pascal Paillier. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. 1999. URL: https://doi.org/10.1007/3-540-48910-X_16.

[39] Ahmad Al Badawi; Yuriy Polyakov. *Demystifying Bootstrapping in Fully Homomorphic Encryption*. URL: https://eprint.iacr.org/2023/149.pdf.

[40] Hans Dobbertin; Antoon Bosselaers; Bart Preneel. *RIPEMD-160:A Strengthened Version of RIPEMD*. URL: https://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf.

[41] Ronald L. Rivest. *A Description of the RC2(r) Encryption Algorithm*. RFC 2268. 1998. DOI: 10.17487/RFC2268. URL: https://www.rfc-editor.org/info/rfc2268.

[42] Ronald L. Rivest. *The MD4 Message Digest Algorithm*. URL: https://www.rfc-editor.org/rfc/rfc1186.

[43] Ronald L. Rivest. *The MD5 Message-Digest Algorithm*. URL: https://www.rfc-editor.org/rfc/rfc1321.

[44] Adleman L. Rivest R. Shamir A. «A Method for Obtaining Digital Signatures and Public-Key Cryptosystems». In: *Communications of the ACM*. 21 (1978), pp. 120–126.

[45] M. L. Dertouzos Rivest R. Adleman L. «On data banks and privacy homomorphisms». In: *Foundations of secure computation* 4,11 (1978), pp. 169–180.

[46] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 2015.

[47] Elisa Giurgea; Tudor Hutu; Emil Simion. *Some Practical Applications of Fully Homomorphic Encryption*. 2023.

[48] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1990.

[49] ZAMA. *Announcing Concrete v1.0.0*. URL: https://www.zama.ai/post/announcing-concrete-v1-0-0.

[50] ZAMA. *Deploy Concrete v1.0*. URL: https://docs.zama.ai/concrete/v/1.0-1/how-to/deploy.

[51] ZAMA. *Documentation Concrete v1.0*. URL: https://docs.zama.ai/concrete/v/1.0-1/.

[52] ZAMA. *ZAMA Website*. URL: https://www.zama.ai/.

[53] Zama. *Concrete: TFHE Compiler that converts python programs into FHE equivalent*. https://github.com/zama-ai/concrete. 2022.

[54] Craig Gentry Zvika Brakerski and Vinod Vaikuntanathan. *(Leveled) Fully Homomorphic Encryption without Bootstrapping*. ACM Transactions on Computation Theory. 2014.