

POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Fully Homomorphic Encryption

Supervisor
prof. Antonio Lioy

Candidate
Maddalena Cintioni

September 2026

† *To my Family and my Friends*

Summary

This thesis was developed by my colleague and me as a single project. We started from the same problem: a drone with limited computing power and battery life is not able to perform complex image processing directly on board. At the same time, the images collected by the drone may contain sensitive information, such as the location or identity of a target. For this reason, this information should not be exposed to an external and untrusted server. Together, we analyzed why traditional encryption or TLS can protect data during transmission, but not while the data is being processed in the Cloud. This led us to the main idea of our thesis: keeping the drone data encrypted not only during transmission, but also while it is processed in the Cloud. To do this, we used Fully Homomorphic Encryption (FHE) with CKKS scheme. We selected the CKKS scheme, implemented with OpenFHE, because it is suitable for approximate calculations on real and complex values, which are the type of data used in our application. We also defined the security model and the general architecture. The drone extracts a numerical representation of the image and encrypts it locally. The encrypted data is then sent to the Cloud. The Cloud performs the required computation directly on the encrypted values and sends an encrypted result back to the drone. Only the drone can decrypt this result. After defining the architecture, we divided the implementation according to different technical areas. My colleague mainly focused on the cryptographic part and on the integration of the different system components, while I mainly focused on image and signal processing. My work started from an important problem: images captured by a drone cannot simply be compared pixel by pixel due to the fact that position, orientation and size of the target can change between two images. Traditional techniques such as SIFT or ORB are difficult to use with CKKS because they require conditional operations that are not directly supported by this type of homomorphic encryption. For this reason, we needed a different approach. I moved the image comparison from the spatial to the frequency domain. I used the Fast Fourier Transform (FFT), to extract frequency information from the images. One important advantage is that the amplitude of the Fourier spectrum is not affected by image translations. To also reduce the effect of rotation and scale changes, I implemented the Fourier-Mellin Transform. In particular, I used a log-polar transformation, which converts rotation and scale changes into simpler shifts. In this way, we obtain a spectral representation that is more robust to changes in the position, rotation and size of the target. I then adapted this representation to the requirements of CKKS. The spectrum was converted into a compact vector with a fixed size, so that it could fit inside the available CKKS SIMD slots. I was also responsible for designing and running the experimental scenarios. We tested a nominal case, a negative case without the target, an urban scenario with background interference, and a scenario where the target was rotated and scaled. For each experiment, I analyzed the resulting matching distances to check whether the system could correctly identify the target and how robust the method was to geometric changes. Overall, my work created the connection between the original images and the numerical representation required by CKKS. This made it possible to perform the matching computation homomorphically in the Cloud and allowed us to evaluate both the advantages and the current limitations of the proposed recognition system.

Acknowledgements

This work of thesis was developed as a joint project starting from a topic proposed by the Army, which gave us the opportunity to study how the Fully Homomorphic Encryption could be implemented in some technologies. Although our academic backgrounds are different, in fact we are students coming from Electronic Engineering and Computer Science Engineering, we considered this project as an opportunity to combine our different skills within the same research. In particular, the work allowed us to connect aspects related to signal and image processing, UAV constraints, Cloud computing, and Fully Homomorphic Encryption. The design of the overall architecture, the implementation of the prototype, and the experimental evaluation were therefore developed through continuous collaboration and discussion between both authors.

The work presented in the thesis was divided as follows:

- Chapter 1 was jointly developed as the general introduction to the project. Within this chapter, the sections concerning the motivation behind FHE and the UAV technological context were mainly developed by Vita, while the sections concerning Cloud security, the Zero-Trust paradigm, and the objectives of the project were mainly developed by Cintioni.
- Chapter 2 was also divided between the two authors: Cintioni mainly developed the sections concerning the history and evolution of cryptography, the main cryptographic methodologies and Homomorphic Encryption, and Fully Homomorphic Encryption generations and applications, while Vita mainly focused on FHE libraries, compilers, and hardware accelerators. The final classification of FHE schemes and the analysis of the CKKS scheme were developed jointly.
- Chapter 3, concerning signal processing through DFT, FFT and the Fourier-Mellin based matching procedure, was mainly developed by Cintioni.
- Chapter 4, concerning the Drone-Cloud architecture, the security model, and the implementation of the system with data flow, was mainly developed by Vita.
- Finally, Chapters 5 and 6, including the experimental evaluation, the comparison between Plaintext, TLS, and CKKS, the discussion of the results, the conclusions and future developments, were developed jointly.

Contents

1 Introduction of the technological context 8
- 1.1 From data protection to data processing: the motivation behind FHE . . . 8
- 1.2 UAVs as a case study: the technological context . . . 9
- 1.3 Security threats and the Zero-Trust paradigm in Cloud Computing . . . 10
- 1.4 Objectives of the project and structure of the thesis . . . 12

2 The homomorphic encryption 14
- 2.1 History and evolution of cryptography . . . 14
  - 2.1.1 The Two Main Cryptographic Methodologies and Their Related Challenges 15
- 2.2 Homomorphic Encryption . . . 20
- 2.3 FHE - Fully Homomorphic Encryption . . . 24
  - 2.3.1 Bootstrapping . . . 25
  - 2.3.2 The Four Generations of FHE . . . 27
  - 2.3.3 FHE for Machine Learning . . . 34
  - 2.3.4 HE in Cloud Computing . . . 36
- 2.4 FHE Libraries . . . 40
  - 2.4.1 The Concrete Library . . . 44
  - 2.4.2 The OpenFHE Library (Open Source Fully Homomorphic Encryption) . . . 48
- 2.5 FHE Compilers . . . 49
- 2.6 FHE Accelerators . . . 55
  - 2.6.1 The Evolution Towards Homo-Processors and New Homomorphic Computing Platforms . . . 57
- 2.7 Classification of FHE Schemes . . . 59
  - 2.7.1 The CKKS Scheme . . . 60

3 Signal Processing 63
- 3.1 Limitations of Spatial Domain Matching and Algorithmic Incompatibility . . . 63
- 3.2 Moving to the Frequency Domain: The Discrete Fourier Transform (DFT) . . . 65
  - 3.2.1 The Fourier-Mellin Transform for Scale and Rotation Invariance . . . 67
  - 3.2.2 Practical Fourier–Mellin Processing Pipeline . . . 68
- 3.3 Extraction of the Invariant Spectral Footprint and Homomorphic Matching . . . 70
- 3.4 Dimensional Optimization and Adaptation to CKKS Scheme Constraints . . . 70

4 Drone-Cloud Implementation and System Design 72
- 4.1 Overall System Architecture . . . 72
- 4.2 Security Model and Trust Assumptions . . . 75
- 4.3 Data Flow . . . 76
  - 4.3.1 Initialization of the cryptographic context and exchange of keys . . . 76
  - 4.3.2 Image acquisition and extraction of local features (RGB Fourier-Mellin) . . . 77
  - 4.3.3 Optimization and local encryption of the difference vector . . . 78
  - 4.3.4 Homomorphic computation on the Cloud and local final decision . . . 78
- 4.4 Cloud Server Implementation (Flask/Docker) . . . 79
- 4.5 Local Decryption and Decision Logic . . . 80
- 4.6 Software Constraints and CKKS Integration . . . 81
  - 4.6.1 Data Representation and Compatibility with CKKS . . . 81
  - 4.6.2 Approximation and Homomorphic Depth Management . . . 81

4.6.3 Transport Encoding and Ciphertext Serialization . . . 82

5 Experimental Analysis and Discussion of Results 83
5.1 Test Environment Setup . . . 83
5.1.1 Hardware and Software Environment . . . 83
5.1.2 Containers Architecture . . . 84
5.1.3 Cryptographic Parameter Tuning (OpenFHE) . . . 85
5.2 Dataset and Validation Scenarios . . . 86
5.2.1 Nominal Case (Auto-Matching) . . . 86
5.2.2 Negative Scenario (Desert Environment) . . . 87
5.2.3 Geometric Variation Scenario . . . 87
5.2.4 Complex Urban Background . . . 87
5.2.5 Final Considerations on the Four Experiments . . . 87
5.3 Performance Evaluation . . . 88
5.3.1 Analysis of CKKS Setup and Client-Side Computational Times . . . 88
5.3.2 Analysis of Cloud and Comparison Times . . . 90
5.3.3 Performance Comparison between Plaintext and TLS . . . 91
5.3.4 Final Comparison between Plaintext, TLS and CKKS . . . 93
5.3.5 Security-Performance Trade-offs and Practical Implications . . . 95

6 Conclusions and Future Developments 98
6.1 Goals Achieved . . . 98
6.2 Future Developments . . . 99
6.2.1 Better accuracy for recognition . . . 99
6.2.2 Improving Computational Performance . . . 100
6.2.3 Expanding the system . . . 101
6.2.4 Final Considerations . . . 101

Bibliography 103

Chapter 1

Introduction of the technological context

1.1 From data protection to data processing: the motivation behind FHE

Modern cryptography has always been good at protecting data both when it is stored somewhere in memory, known with the name data-at-rest and when it travels across a network, known under the name of data-in-transit. If we consider symmetric algorithms like AES (Advanced Encryption Standard) or TLS (Transport Layer Security), the first is nothing more than a block cipher that operates on data blocks with fixed size of 128 bit, it uses the same secret key for both encryption and decryption and it is able to protect a file on a disk. If instead, we consider the TLS cryptographic protocol, it was designed to ensure secure communications, confidentiality and data integrity between a client and a server over an unsecured network like the Internet. In particular, both of them are capable of maintaining and guaranteeing a secure connection between two endpoints. It is for these characteristics that, for a long time, this was considered enough especially because computation itself used to happen locally that means on a machine controlled directly by the data owner. This assumption was overcome once cloud computing became the standard way of running any demanding task we have to work with. In fact, today it is completely normal to send data to a remote server only for the simple reasons that the server has more computing power, more memory or it is always available for any demanding tasks, while if we think to the local device, they does not have these resources. This shift introduced a third state of data that traditional cryptography was never built to protect, this state is called under the name of data-in-use, meaning data while it is actually being processed. In this way, as soon as a server has to run a computation, it must first decrypt the data and load it into memory in plaintext. This is exactly the point where confidentiality breaks down, because a compromised server, a curious system administrator or even a memory level attack can now reach the original information.

The question of whether this gap could be closed is actually not new. Already in 1978, Rivest, Adleman and Dertouzos asked whether a third party could perform useful computations on encrypted data without ever needing to decrypt it, this new idea started to developed under the name of privacy homomorphism. However, for more than thirty years nobody managed to build a scheme able to support both addition and multiplication at the same time, for an unlimited number of operations and without taking into account that ciphertext may be corrupted by the mathematical noise that these operations accumulate. This gap between plaintext protection and processed data protection was closed only in 2009, when Craig Gentry proposed, in his doctoral thesis, the first working construction of what he named **Fully Homomorphic Encryption (FHE)**, built on the geometry of ideal lattices [1]. Ideal lattices are mathematical structures used in cryptography, especially in post-quantum cryptography, that can create extremely secure and computationally efficient algorithms to provide a hard-to-solve problem on which security is based.

What we want to discuss in this introductory chapter is not the mathematical detail of how

this problem was solved, since the full evolution of the different families of schemes is presented in Chapter 2, but the practical consequence of Gentry's breakthrough. In fact, for the first time taking into account this assumption, an external and untrusted infrastructure could be used as a pure computational resource, without ever being trusted with the real content of what it processes. So in other words, we can say that FHE was not developed for any specific application, but it is a general answer to a general and long standing cryptographic problem. This general property becomes relevant and of great interest especially when applied to a real, resource constrained scenario, where a device is physically unable to process its own data, but still needs to keep it confidential and undecipherable to anyone who helps it process it.

Unmanned Aerial Vehicles(UAV), presented in the next section, are exactly this kind of scenario and represent the application case we focus on for this work.

1.2 UAVs as a case study: the technological context

UAV also known as: Remotely Piloted Aircraft; UAS (Unmanned Aerial System); RPV (Remotely Piloted Vehicle); ROA (Re- motely Operated Aircraft); UVS (Unmanned Vehicle System) or commonly named today as drones. They are a good example of a device that needs the property of a system having the ability to perform computations on encrypted data without revealing their actual content. They are equipped with cameras and other sensors able to collect large amounts of visual data, even if they operate under strict weight, power and battery limits. So, as a consequence of that, they are not able to run heavy computation on board. At the same time, if we send data to a remote server for processing, this would mean exposing sensitive operational information to an infrastructure we are not sure we can fully trust. However, before explaining this trade-off in detail, it can be useful to look at how unmanned flight reached this point.

The idea of flying without a human pilot is much older than you might think. In fact, already in 1898 in New York, Nikola Tesla presented a model of a radio controlled boat, demonstrating how a vehicle could in principle be remotely controlled. The first real and documented use of an unmanned vehicle in warfare dates back to 1849, when Austrian forces launched incendiary balloons against Venice. However, the real turning point, came during the First World War, together with the development of gyroscopic flight stabilization by Elmer and Lawrence Sperry. In fact, it was between 1916 and 1918 that Archibald Low tested the radio controlled Aerial Target, while at the same time American engineers built the Hewitt-Sperry Automatic Airplane and the Kettering Bug. None of these early machines actually was used for combat purpose, but they are considered to be the technical foundation for everything that followed.

It was during the interwar years, that French engineers Percheron and Boucher, in 1932, managed to take off and land a radio controlled Voisin biplane, while in 1935 Britain produced the DH.82B Queen Bee target aircraft that can be generally considered as the origin of the word drone itself. In fact, in the United States, Reginald Denny's company mass produced around 15,000 radio controlled target planes for the Second World War.

If instead, we focus on the Italian history, we can say that we followed a similar path. We started with the Radio-Piloted Aircraft (ARP) during the Second World War and later we adopted the Canadair CL-89 as our first operational UAV, in service until 2000, followed over the decades by the Mirach family and eventually the General Atomics RQ-1 Predator. The same period also saw the first armed and hazard facing uses of unmanned aircraft, from the American TDR-1 to the unmanned Hellcats and B-17s flown into the radioactive clouds of Bikini Atoll in 1946, and later the AQM-34 Firebee reconnaissance missions during the Vietnam War. But it was during more recent years, that drones became essential in disaster response for example during the 2011 Tohoku earthquake and the Fukushima nuclear crisis, where high altitude platforms could fly over areas that instead could be too dangerous for any crewed aircraft. Beyond military and emergency use, the technological progress of the 2000s opened the door to civilian applications such as modern drones equipped with: visible; infrared; multi-spectral and LiDAR sensors, now used for precision agriculture; environmental monitoring; infrastructure inspection and commercial filming; but also for many other tasks.[2, 3]

What all of these applications have in common and what makes UAVs particularly relevant for our project of thesis?

It is the same physical constraint: a drone must remain light and energy efficient in order to fly and this directly limits how much computation it can perform on its own. This constraint

is generally summarized through the engineering principle known as SWaP (Size, Weight and Power). But, due to the fact that battery is mostly consumed by the propulsion system needed to keep the aircraft on flight, the onboard processor cannot be too large and too powerful, so it has to rely instead on low power embedded platforms. Platforms such as those in the Nvidia Jetson family cannot perform demanding cryptographic or mathematical operations without noticeably reducing flight time.

For this reason, purely theoretical architectures are not able to work. In fact, if we think to a Cloud-Only approach where the drone continuously streams raw high resolution video to a remote server, this would require a bandwidth that is simply not available in the remote, rural areas or war zone environment where drones typically operate. On the other hand, an Edge-Only approach is limited by the onboard hardware itself, which usually cannot run complex deep learning models without a significant drop in detection accuracy.

Today, modern system design solves this conflict through what is known as the **Split Inference** paradigm which means that the computation is divided so that the lighter, initial layers run locally on the drone, while the remaining, more demanding layers are completed on a remote edge or cloud server. In practice, this workload distribution can be described in two stages:

- **Edge Stage:** During flight, the drone processes each frame with lightweight, optimized algorithms that act as a first filter, by keeping only the relevant information and discarding the rest. This also gives the system a useful degree of fault tolerance, since the drone can keep operating and storing data locally even if the connection to the ground station is temporarily lost.

- **Cloud Stage:** The remote infrastructure receives only the data already filtered by the drone, which significantly reduces the required bandwidth and allows the system to keep working even over degraded radio or satellite links.

This architecture can be considered to be very useful in several domains, from tactical reconnaissance and public safety, where it supports real time situational awareness without relying on fixed infrastructure, to the inspection of critical infrastructure such as pipelines and power lines, but also to precision agriculture, where it enables real time spectral monitoring of crops and land.

However, the Split Inference model, despite is able to solve the bandwidth and computation problem, it introduces exactly the privacy weakness discussed in the previous section which means that the intermediate data and image frames are transmitted to the external infrastructure in plaintext, exposing operational information, geographic coordinates and even the identity of the monitored target. This is the starting point of the work presented in this thesis, whose objective is to extend the conventional Edge-to-Cloud architecture by integrating a secure processing pipeline based on the **CKKS (Cheon, Kim, Kim e Song) Scheme** of Fully Homomorphic Encryption, a scheme that allows you to perform mathematical calculations such as addition and multiplication, directly on encrypted data, without the need to decrypt it first. In this way the cloud server can perform the required matching computation while remaining completely blind to the data it is processing and without ever known the real context of data.[4]

1.3 Security threats and the Zero-Trust paradigm in Cloud Computing

If the Edge-to-Cloud architecture described in the previous section is able to address the physical and computational limitations of UAVs through the use of computation offloading, outsourcing data processing to external server infrastructures instead introduces significant security and privacy challenges.

When talking about **computation offloading**, meaning transferring computationally intensive tasks to a separate processing unit like an hardware accelerator, or to an external platform such as a computing cluster or a Cloud infrastructure. Offloading to a coprocessor can accelerate applications such as image rendering and complex mathematical computations. At the same time, offloading computation to a remote platform over a network makes possible to overcome the hardware limitations of resource constrained devices, including limited processing power, storage capacity and battery life. If we focused on operational scenarios as for example reconnaissance,

tactical missions or environmental monitoring, the information exchanged between the UAV and the remote station may contain highly sensitive strategic data. This can include GPS coordinates that are able to reveal the exact location of the aircraft or the monitored area, but also high resolution video streams able to describe the geometry of the environment, and most importantly, the identity of the detected targets. Therefore, the compromise of even a single piece of this information could bring the entire mission to fail or expose critical infrastructures to serious security threats.

In this context, conventional cryptographic techniques based on symmetric and asymmetric algorithms, such as AES and TLS/HTTPS (Hypertext Transfer Protocol Secure), a protocol suite that secures HTTP communications through TLS encryption, commonly used to secure communication channels, are no longer sufficient. In fact, traditional cryptographic systems are designed to protect data only during two specific stages of its life cycle: when data are transmitted over a communication channel (*data-in-transit*) and when data are stored on a physical or digital medium (*data-at-rest*). However, these encryption schemes do not allow computations to be performed directly on encrypted data. As a result, before executing any computer vision algorithm or image matching operation, the Cloud server must first decrypt the received information and process it in plaintext within its main memory. It is exactly when data are decrypted and loaded into volatile memory, that the entire system becomes vulnerable to cyber attacks, platform compromises or even malicious system administrators. For example, if an attacker gains privileged access to a compromised Cloud server, they may perform memory scraping techniques to inspect the server's RAM and recover sensitive image data or intermediate processing results while they are temporarily stored in plaintext. Similarly, a malicious or compromised administrator with sufficient privileges could potentially access these data during processing, despite the use of encryption for both storage and communication [5]. All of these factors could compromise the confidentiality of the processed information. The way we have to address the security risks introduced by outsourced computation is adopting an architecture able to implement the **Zero-Trust** paradigm as a core design principle.[6]

Unlike traditional security models, which rely on a perimeter based approach and assume that any user or device operating inside the organization's network can be trusted once protected by a firewall, the **Zero Trust Architecture (ZTA)** is built around the fundamental principle of *"Never trust, always verify"*.

This security framework redefines access management by treating every request needed by the system, as if it is originated from an untrusted network. As a consequence of that, ZTA requires continuous and context aware verification through authentication mechanisms, real time monitoring and end-to-end encryption to prevent cyber attacks and to limit the movement of potential threats.

Within this model, the external Cloud server is assumed to follow the threat model, which is known in literature as **Honest-but-Curious**. Under this assumption, the Cloud provider is considered operationally trustworthy, meaning that it correctly performs all requested computations, but at the same time, it should be able to inspect or retain any information if it is available in plaintext. For this reason, the proposed architecture follows the principle of never trust, this is always verify because we start by assuming that the external infrastructure may already be compromised and therefore denying any implicit trust to the Cloud.

Focusing on the development of this thesis, **Fully Homomorphic Encryption** represents a suitable cryptographic mechanism for reducing the trust that must be placed in the external infrastructure when sensitive data are processed remotely. By exploiting the properties of the CKKS encryption scheme, the Cloud server is no longer required to access plaintext data or confidential credentials. Instead, it acts only as a computational engine capable of performing all required operations directly on encrypted information. More specifically, the server receives only encrypted numerical values, namely *ciphertexts* and executes the spectral matching algorithm using native homomorphic operations such as addition, multiplication and squaring, without ever accessing the underlying plaintext. The resulting output remains encrypted and can be decrypted only by the key owner. In this way, the proposed architecture supports the principles of least privilege access and data confidentiality by reducing the exposure of plaintext information to the remote Cloud. However, this protection introduces additional computational overhead that we will evaluate in Chapter 5. However, FHE alone does not provide all the mechanisms required by a complete Zero-Trust Architecture, such as authentication, message integrity, replay protection,

and continuous access verification.

1.4 Objectives of the project and structure of the thesis

The objective of this thesis is to design, implement and experimentally validate an Edge-to-Cloud architecture for UAV based tactical reconnaissance and environmental monitoring, following the principles of the Zero-Trust paradigm. The big challenge here is the trade off between the fact that on one hand, drones have very limited battery and processing power, so they must send heavy math tasks to a strong remote server. While on the other hand, we cannot let that server see the private data because we need to preserve the confidentiality of sensitive information throughout the computation process.

To solve this, we created a new way to process and encrypt images that changes how standard Split Inference works. Usually, Split Inference divides the work between the drone and the Cloud quite well, but in this way the remote server still has to read intermediate data in plain text, which leaves sensitive information wide open.

In this work, we maintain the same workload distribution while changing the role of the Cloud. In fact, instead of transmitting raw data for processing, all computations are performed directly on encrypted data, ensuring that the Cloud never gains access to the underlying information. With this approach we can distribute the computational workload that is well protected, in different steps:

1. **Feature extraction at the Edge:** The onboard processing module acquires image frames and performs an initial low cost local processing stage. This means that each image is projected into the frequency domain using the Fourier-Mellin Transform, computed through the Fast Fourier Transform (FFT) algorithm to reduce computational complexity and execution time on the embedded platform. This mathematical approach combines spectral analysis with log-polar sampling, allowing the extraction of the main geometric characteristics of the target while providing robustness against image rotations and moderate scale variations.

2. **Local cryptographic protection:** Before any data transmission takes place, the extracted spectral feature vectors are locally encrypted by using the CKKS homomorphic encryption scheme implemented through the OpenFHE cryptography library. The resulting data are converted into encrypted numerical representations, namely *ciphertexts*, that are unreadable to any external entity that does not possess the private decryption key.

3. **Homomorphic computation on the Cloud:** the Cloud receives only the encrypted difference vector and performs its homomorphic squaring through ciphertext multiplication. The plaintext feature vectors and their difference are never exposed to the remote server.

4. **Local output generation:** the resulting ciphertext is returned to the UAV, where it is decrypted. The Client aggregates the decrypted squared components into the final spectral score and compares it with the selected tolerance threshold in order to obtain the recognition decision.

Thanks to this architecture, the Cloud infrastructure can be seen as a blind computational executor that performs the required operations without accessing the processed data in plaintext. In the proposed project, the data we work with consist of spectral feature vectors extracted from frequency domain representations of images used for target identification. To develop the following idea, we decided to organize this thesis work into six different chapters:

- **Chapter 1** used to introduce the application context, highlighting the technological challenges and security requirements that motivated the design of the proposed architecture.

- **Chapter 2** where we present the origin of the homomorphic encryption, starting from its historical evolution, going on with the main cryptographic schemes, the available software libraries, dedicated compilers and hardware accelerators.

- **Chapter 3** used to cover the mathematical rules of signal processing, a focus on the Discrete Fourier Transform (DFT/FFT) and the Fourier-Mellin Transform, explaining how these techniques are integrated with the CKKS encryption scheme.

- **Chapter 4** used to describe the overall software architecture of the proposed system, including the communication interface between the Edge node and the Cloud infrastructure, along with the corresponding secure data flows.
- **Chapter 5** where we present and discuss the tests we performed and the results we obtained by focusing on execution times and the accuracy of the proposed spectral matching algorithm.
- **Chapter 6** last chapter in which we summarize the main contributions of this work and outline possible future research directions and potential applications of homomorphic encryption.

Chapter 2

The homomorphic encryption

2.1 History and evolution of cryptography

Today, when talking about technology, cybersecurity and data encryption play a fundamental role in ensuring the confidentiality, authenticity and integrity of information. Their applications cover lots of range from protecting everyday personal communications or validating digital signatures to secure financial transactions and government communication channels that are classified as top secret.

From an etymological perspective, the term **cryptography** [7] comes from the Greek words *kryptós* (hidden) and *gráphein* (to write), referring to the discipline that studies techniques for making information unintelligible to anyone that is not the final recipient. Formally, cryptography can be defined as the process of transforming readable information, known as *plaintext*, into an unreadable encrypted form, known as *ciphertext*, with the purpose of protecting sensitive data from unauthorized access.

Although the exchange of secret messages dates back to ancient civilizations with some of the earliest examples found in Egyptian and Mesopotamian societies, cryptography and cryptanalysis are considered modern scientific disciplines whose greatest advances have taken place over the last century. Historically, cryptography was closely linked to espionage and military communications and it is commonly associated with devices such as the Enigma machine, used by the German armed forces during the World War II. It is only with the beginning of the digital era and with the birth of informatics, that we can underline the transition from classical cryptographic techniques to modern ones, enabling the development of increasingly sophisticated encryption algorithms.

If we think to the modern cryptographic systems, these are based on mathematical algorithms able to transform plaintext into an unreadable representation using one or more cryptographic keys. The process begins by identifying the information that requires protection, which may include text messages, files, images or even video streams. After that, the encryption algorithm converts the original data into a ciphertext, generating an encrypted sequence that can only be interpreted with the appropriate decryption key. In general, increasing the key length significantly leads to the growth of computational effort required to recover the original information through unauthorized attacks. Once it is encrypted, the ciphertext is transmitted over the communication channel. At this point, even if data are intercepted by an unauthorized author, the strength of the encryption algorithm makes impossible the recovering and so the use of the original data. The legitimate recipient, on the other hand, can restore the original information by applying the corresponding decryption process with the correct key. Although the encryption and decryption keys are mathematically related, this relationship ensures that the operation performed with one key can be reversed only by using the corresponding key. In many practical systems, the decryption phase is also accompanied by authentication mechanisms that verify the integrity and authenticity of the recovered data.

Modern cryptographic standards not only are able to hide information, but they also support the three fundamental principles of information security that are:

- **Integrity:** Protection against improper information modification or destruction, including ensuring information authenticity and non repudiation [8].

- **Authentication:** Verifies the identity of the entity responsible for generating or transmitting the information.

- **Non repudiation:** Prevents any entity within the system, whether a user, software application or device, from denying responsibility for a performed action or completed transaction.

Today, cryptographic technologies are able to protect data both when they are stored (*data-at-rest*), when they are transmitted (*data-in-transit*), and through modern approaches such as homomorphic encryption, even when they are processed (*data-in-use*), regardless of whether the data resides in local infrastructures or Cloud computing environments. However, these protection mechanisms do not necessarily have to be adopted all together, as their use depends on the specific requirements and threats associated with a given scenario. For instance, encrypting data at rest may be sufficient when protecting sensitive information stored on a local device, while data transmitted between a drone and a Cloud server may require both *data-at-rest* and *data-in-transit* protection. In scenarios where sensitive data must also be processed by an untrusted external platform, such as a Cloud server, *data-in-use* protection can be added through homomorphic encryption, allowing computations to be performed without exposing the underlying plaintext.

2.1.1 The Two Main Cryptographic Methodologies and Their Related Challenges

Focusing on the main cryptographic methodologies and techniques, these can be divided into two major categories according to the nature of the keys involved: **symmetric cryptography and asymmetric cryptography**.

Symmetric cryptography

It performs both encryption and decryption operations using a single secret key that is shared among all authorized authors. This approach is very fast and computationally efficient however, it requires careful key management since the same key is used for both encryption and decryption operations. For this reason, symmetric cryptography is mainly used in scenarios where performance and execution speed are critical, such as the protection of large volumes of data or the securing of communication channels within closed systems.

The main symmetric cryptographic algorithms both block based and stream include:

- **Data Encryption Standard (DES):** Introduced by IBM in the 1970s as a global encryption standard. It is based on a relatively short key length, which made it vulnerable to brute force attacks and brought to its complete replacement to safer algorithms.

- **Triple DES (3DES):** Developed as a countermeasure to the weaknesses of DES, it applies the original algorithm three consecutive times to each block of data. Although it increases the effective key length, it is now considered obsolete and has been largely replaced by modern solutions.

- **Advanced Encryption Standard (AES):** It represents the globally adopted symmetric encryption algorithm used globally by governments and organizations. It provides strong mathematical security and supports key lengths of 128, 192 and 256 bits.

- **Twofish:** An open-source symmetric block cipher known for its high execution speed and resistance to cryptanalysis. It operates on fixed-size data blocks of 128 bits and supports symmetric keys of 128, 192 and 256 bits. It is mainly used in scenarios where both performance and security are required.

- **Salsa20:** A symmetric stream cipher designed by Daniel J. Bernstein as an alternative to traditional block ciphers. It relies on simple arithmetic and logical operations, enabling efficient software implementations while providing a high level of security. It supports 128 and 256 bit keys and is particularly suitable for applications where efficient encryption based on software is required.

- **ChaCha20:** A symmetric stream cipher derived from Salsa20 and designed to provide both high performance and strong security. It uses a 256 bit key and is particularly well suited to software implementations, including systems where dedicated hardware acceleration is unavailable. It is widely used in modern security protocols and is commonly combined with the Poly1305 message authentication code.

Asymmetric cryptography

It is very used due to the fact that it is able to overcome the problem of key exchange by using a pair of mathematically related but distinct keys. In fact, it relies on a public key which can be distributed and used by anyone who want to encrypt data and a private key, which is securely stored by the owner and represents the only way for decrypting the corresponding ciphertext. Although this paradigm is slower and computationally more demanding due to the complexity of the underlying mathematical operations, it provides stronger security by eliminating the need to exchange secret keys over insecure communication channels. Its main applications are based on online transactions and digital signatures.

Before introducing the different types of asymmetric cryptography, it is good to mention the concept of **Enveloped Data**, which combines symmetric and asymmetric cryptographic techniques to protect information efficiently. In this approach, the actual data are encrypted using a randomly generated symmetric key, while the symmetric key is then encrypted using the recipient's public key. The recipient can recover the symmetric key using the corresponding private key and subsequently decrypt the original data. This approach combines the efficiency of symmetric encryption for large amounts of data with the secure key distribution provided by asymmetric cryptography.

The main asymmetric cryptographic algorithms include:

- **RSA (Rivest-Shamir-Adleman):** One of the most widely known asymmetric cryptographic algorithms, based on the computational difficulty of factoring large integers. It has traditionally been used for secure key exchange, digital signatures, and authentication mechanisms, although more efficient alternatives are commonly preferred for modern applications.

- **Diffie-Hellman (DH):** A key agreement protocol that allows two parties to establish a shared secret over an insecure communication channel without directly transmitting the secret itself. Its security relies on the computational difficulty of the discrete logarithm problem and it is mainly used to establish symmetric session keys.

- **Elliptic Curve Diffie-Hellman (ECDH):** An elliptic curve variant of the Diffie-Hellman key agreement protocol. It provides the same basic functionality as traditional Diffie-Hellman while offering comparable security with significantly shorter keys. This makes it particularly suitable for systems with limited computational and hardware resources.

- **Digital Signature Algorithm (DSA):** A public key cryptographic algorithm specifically designed for generating and verifying digital signatures. It allows a recipient to verify the authenticity and integrity of a message and to confirm that it was signed by the holder of the corresponding private key.

- **Elliptic Curve Digital Signature Algorithm (ECDSA):** An elliptic curve variant of DSA used to generate and verify digital signatures. It provides strong security with shorter key sizes than traditional DSA and RSA, making it suitable for applications where computational efficiency and reduced storage requirements are important.

- **Elliptic Curve Cryptography (ECC):** A family of public key cryptographic techniques based on the algebraic properties of elliptic curves defined over finite fields. Its main advantage is the ability to provide security levels comparable to RSA and other traditional public key algorithms using significantly shorter keys. This makes ECC particularly suitable for devices with limited computational and hardware resources, such as smartphones and IoT nodes.

To manage asymmetric cryptography on a large scale, organizations implement a **Public Key Infrastructure (PKI)**. A PKI provides a complete architectural framework for the generation, distribution, revocation and validation of key pairs and their associated digital certificates, enabling secure web applications, e-mail communication and TLS/SSL(Secure Sockets Layer) sessions. A simple example of PKI can be seen when a user connects to an HTTPS website. The server has a public/private key pair and obtains a digital certificate from a trusted Certificate Authority (CA), which binds the server's identity to its public key. When the user connects, the server sends its certificate to the browser, which verifies the CA's signature, the certificate's validity period, the domain name and its revocation status. If these checks are successful, the certificate can be trusted and the TLS handshake can proceed, establishing a secure connection.

Certificate revocation is important because a certificate may become untrusted before its expiration date, for example if its private key is compromised. One mechanism used by PKI is the **Certificate Revocation List (CRL)**, a digitally signed list published by a CA containing the serial numbers of revoked certificates. The client can download the CRL and check whether the certificate appears on the list. Another mechanism is **Online Certificate Status Protocol (OCSP)**, which allows a client to query an OCSP responder for the status of a specific certificate instead of downloading the entire CRL. The response indicates whether the certificate is good,revoked or unknown. [9]

Despite their rigorous mathematical bases, modern cryptographic technologies still present specific weaknesses and systemic vulnerabilities when exposed to targeted attacks. Focusing on the main problems we can deal with:

Quantum Computing

It represents one of the most critical threats to asymmetric cryptography. It is known from the principles of quantum mechanics that quantum computers are theoretically able to execute Shor's algorithm in polynomial time. This algorithm enables the efficient solution of complex mathematical problems, such as large integer factorization and discrete logarithms. Since the security of traditional asymmetric cryptographic systems, including RSA and ECC, relies on the computational infeasibility of solving these problems within a reasonable time frame, the development of large scale quantum computers could potentially compromise current security infrastructures. For this reason, organizations should begin migrating to **Post-Quantum Cryptography (PQC)** solutions as early as possible, rather than waiting until large scale quantum computers become available. This is particularly important because of the so called "harvest now, decrypt later" threat: an attacker can collect and store encrypted data today and attempt to decrypt it in the future once sufficiently powerful quantum computers are available. Sensitive information with a long term value, such as confidential communications or personal and financial data, could therefore be exposed even if it is securely encrypted today. Consequently, the transition towards quantum resistant cryptographic algorithms is becoming an important part of long-term cybersecurity strategies. Today, Artificial Intelligence techniques can also support the development and optimization of cryptographic solutions, contributing to the effort to prepare current security infrastructures for future quantum threats. Unlike asymmetric cryptography, symmetric cryptographic algorithms and cryptographic hash functions are not fundamentally broken by quantum computing, although their security is still affected by quantum algorithms. In particular, **Grover's algorithm** can theoretically reduce the complexity of a brute force search from approximately 2^n to $2^{n/2}$ operations. Therefore, its impact can be mitigated by increasing the key size of symmetric algorithms and, where appropriate, using hash functions with sufficiently large size of the output. As a result, while asymmetric cryptography requires a transition to specifically designed PQC algorithms, symmetric cryptography and cryptographic hash functions can generally remain secure with appropriate security margins and larger parameters.

Brute force attacks

These attacks consist of systematic and automated attempts to test all possible combinations of cryptographic keys until the correct one is identified. Although brute force attacks against modern encryption algorithms require computational times that are not practically feasible even for current supercomputers, continuous improvements in parallel and distributed computing capabilities may progressively reduce the security margins provided by the used key. The effectiveness of

a brute force attack mainly depends on the key length and the computational resources available to the attacker. For example, a short or weak cryptographic key can potentially be discovered by testing a relatively limited number of possible combinations, while sufficiently long keys make exhaustive search computationally impractical.

Side-Channel attacks:

Unlike theoretical cryptanalysis, side-channel attacks do not attempt to break the mathematical foundations of an algorithm. Instead, they analyze the physical characteristics of the hardware device while performing cryptographic operations. In this way, anyone interested in hacking the system, can be able to measure execution time variations, power consumption patterns or electromagnetic emissions, all parameters that can be used to reconstruct secret keys or recover plaintext information. For example, a device performing an encryption operation may consume slightly different amounts of power depending on the intermediate values being processed. By collecting and statistically analyzing a large number of measurements, an attacker may infer information about the secret key. Common countermeasures include constant time implementations, masking sensitive intermediate values and reducing unintended information leakage from the hardware.

Due to these new threats, research is moving towards new solutions in terms of security. On one hand, Artificial Intelligence (AI) techniques are being introduced to optimize predictive key management and to develop quantum resistant algorithms, such as post-quantum cryptography, capable of adapting before quantum infrastructures can compromise the system. On the other hand, we can find quantum cryptography for the key distribution mechanisms based on quantum principles.

Today, organizations have access to multiple cryptographic solutions to implement multilayered security strategies, selecting the most appropriate technologies according to data type, requirements and operational constraints. The main solutions we can deal with are:

- **Encryption Software:** These are software applications that provide users and organizations with tools to encrypt and decrypt data without requiring them to implement cryptographic algorithms themselves. They can be used to protect individual files, folders, or larger collections of information and may also provide functions for generating, storing, rotating, and managing cryptographic keys. In enterprise environments, encryption software can be integrated with databases, cloud platforms, and other security systems, allowing encryption to be applied as part of normal data management processes.

- **Virtual Private Networks (VPNs):** A VPN creates an encrypted communication tunnel between a user's device and a VPN server, protecting the traffic that travels across an untrusted network such as a public Wi-Fi connection. This makes it more difficult for third parties connected to the same network to intercept or read the transmitted information. VPNs are therefore commonly used by organizations to provide employees with secure remote access to internal resources and services.

- **Cloud Encryption:** Cloud encryption solutions protect data that is stored or processed in cloud environments by converting readable information into ciphertext. This can be important when sensitive data is stored on infrastructure managed by a third party provider. Depending on the solution, encryption can be applied before the data is uploaded to the cloud or directly within the cloud environment. Proper key management is essential because anyone who obtains both the encrypted data and the corresponding encryption keys may potentially access the original information.

- **Network Encryption:** Network encryption protects information while it is being transmitted between two or more systems. A common example is TLS, which is used by HTTPS to secure communication between a web browser and a server. During the establishment of a TLS connection, cryptographic mechanisms are used to authenticate the server and establish session keys, which are then used to encrypt the exchanged data. This prevents attackers from easily reading or modifying sensitive information such as login credentials or payment details while it travels across the network.

- **Database Encryption:** Database encryption is used to protect structured information stored in database systems, such as customer records, financial information or authentication

data. Encryption can be applied to an entire database or only to some sensitive fields, such as credit card numbers or personal identification information. It provides protection if an attacker gains unauthorized access to the underlying database files or storage, although it does not, by itself, prevent an authorized application or user from accessing the decrypted data.

- **Full Disk Encryption (FDE):** Full Disk Encryption encrypts the entire contents of a storage device, including the operating system, applications and user files. The data is automatically decrypted when an authorized user successfully authenticates, while the information remains unreadable when the device is powered off or the storage is accessed without the appropriate credentials or encryption key. For example, FDE can protect a company laptop if it is lost or stolen, preventing someone from simply removing the storage drive and reading its contents.

- **Hardware-Based Encryption:** Hardware-based encryption uses dedicated components, such as security chips or Hardware Security Modules (HSMs), to perform cryptographic operations and protect sensitive keys. Because cryptographic keys can be stored and processed within a controlled hardware environment, they are less exposed to attacks targeting the operating system or application software. This approach is commonly used in high security environments where the protection of cryptographic keys is particularly important, such as financial institutions and large enterprise infrastructures.

- **File and Folder Encryption:** File and folder encryption allows users or administrators to protect selected files or directories rather than encrypting an entire storage device. The encrypted files remain unreadable to anyone who does not possess the required key or credentials. This approach is useful when only specific information needs additional protection, for example when storing confidential documents on a shared computer or transferring sensitive files through an external storage device.

- **Email Encryption:** Email encryption protects the content of electronic messages and their attachments from unauthorized access. Depending on the technology used, encryption can protect the message while it is being transmitted between mail servers or provide stronger protection in which only the intended recipient can decrypt the content. This is particularly useful when emails contain confidential information, such as business documents, personal data or financial records.

- **End-to-End Encryption (E2EE):** End-to-End Encryption is a communication model in which the message is encrypted on the sender's device and remains encrypted while traveling through the network, being decrypted only on the intended recipient's device. Intermediate systems, such as servers or communication providers, only handle the encrypted data and should not have access to the plaintext or the encryption keys. E2EE is commonly associated with secure messaging applications and is useful for protecting the confidentiality of private conversations even when the communication infrastructure cannot be fully trusted.

While the solutions described above provide protection for data when they are stored or transmitted, another important challenge to face is when sensitive information needs to be processed. In traditional encryption systems, data generally need to be decrypted before computations can be performed. This is where **Homomorphic Encryption (HE)** becomes particularly relevant, as it addresses the protection of data while they are being processed.

Homomorphic Encryption is a class of encryption techniques that allows computations to be performed directly on encrypted data without requiring the plaintext to be revealed. The encrypted result can then be decrypted by the authorized party to obtain the corresponding result of the computation. This property is particularly useful in cloud computing and outsourced environments, where organizations may want to take advantage of external computational resources without exposing sensitive information to the infrastructure performing the computation.

In this context, Homomorphic Encryption can support the principles of the Zero-Trust security model by reducing the need to trust the infrastructure that processes sensitive data. Instead of assuming that external systems are completely trustworthy, data can remain encrypted throughout the computation, limiting the exposure of plaintext information. Therefore, Homomorphic

Encryption extends the protection provided by conventional encryption from data at rest and data in transit to the more challenging scenario of **data in use**.

2.2 Homomorphic Encryption

The roots of homomorphic encryption do not originate from recent technological developments, but its origins can be traced back to the early stages of modern computer science. The first theoretical intuition dates back to 1978, when researchers Ron Rivest, Leonard Adleman and Michael Dertouzos published a pioneering work introducing, for the first time, the concept of **Privacy Homomorphism**. The original objective of this research was to enable the processing of sensitive information within external centralized databases, without exposing the underlying plaintext data and without compromising its confidentiality. At the same time, Rivest and Adleman formalized the RSA asymmetric encryption algorithm, which immediately revealed a native multiplicative homomorphic property.

However, for more than three decades, the scientific community was unable to develop a mathematical framework able to simultaneously allow both addition and multiplication operations for an unlimited number of computational cycles. It is at this point that a breakthrough arrived in 2009 with the doctoral thesis of **Craig Gentry**. Gentry introduced the first scheme of **Fully Homomorphic Encryption (FHE)** based on ideal lattice structures. The main innovation introduced by Gentry was the **Bootstrapping** technique, a process able to reduce and reset the mathematical noise accumulated within ciphertexts during computations. By solving this noise problem, it became possible to run complex mathematical and logical tasks directly on encrypted data for the very first time.

The importance of this innovation becomes evident when compared with the limitations of conventional cryptographic environments. As we known, in traditional systems, the protection of data is confined only to storage phases and network transmission phases. So, whenever computational processing is required, the system architecture forces the data to be decrypted and loaded into memory, creating a continuous and potentially vulnerable sequence of encryption and decryption operations. To overcome this limitation and extend protection to the computation phase, the **Homomorphic Encryption (HE)** was introduced.

The concept of **Homomorphism** in computer science and cryptography comes from the corresponding principle in abstract algebra. From a mathematical point of view, homomorphism can be defined as a function between two algebraic structures of the same type, such as two groups or two rings, which preserves the internal operations defined within them. A group is an algebraic structure with a single operation, while a ring has two operations, typically addition and multiplication, that satisfy specific properties.

In fact, given two groups $(G, \cdot)$ and $(H, *)$, a group homomorphism is a function $\phi : G \to H$ such that, for every pair of elements $x, y \in G$, is true the following relation:

$$\phi(x \cdot y) = \phi(x) * \phi(y)$$

This equation represents the concept of preservation of the operation. In fact, performing an operation first within the original group G and then mapping the result into H through the function ϕ is equivalent to individually mapping the two elements into H and subsequently applying the internal operation defined in H.

In the field of homomorphic encryption, the original group G represents the plaintext message, while the target group H represents the ciphertext. The function ϕ corresponds to the encryption algorithm *Enc*.

This means that the encryption function behaves as a mapping that preserves the relevant mathematical operations. Furthermore, homomorphic encryption schemes are designed to preserve these operations even when they are applied directly to encrypted data. As a consequence, applying a mathematical operator directly to ciphertexts produces a result that, for exact homomorphic schemes, corresponds to the exact plaintext computation, while approximate schemes such as CKKS preserve the result up to a controlled numerical approximation.

To describe in a formal way this behavior, an encryption system can be considered properly homomorphic if it satisfies the following **two fundamental criteria**:[10]

1. An encryption system is homomorphic with respect to an operation "$\star$" if it satisfies the following equation:

$$\text{Enc}(m_1) \star \text{Enc}(m_2) = \text{Enc}(m_1 \star m_2), \quad \forall m_1, m_2 \in \mathcal{M}$$

   where $\text{Enc}(\cdot)$ represents the encryption algorithm and $\mathcal{M}$ is the set of all possible messages.

2. Given two ciphertexts $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$, where m_1 and m_2 are the corresponding plaintext messages, it is possible to obtain:

$$\text{Dec}\left(\text{Enc}(m_1) \star \text{Enc}(m_2)\right) = m_1 \star m_2$$

   where "$\star$" represents either addition or multiplication and $\text{Dec}(\cdot)$ defines the decryption algorithm.

If we consider two generic plaintext values a and b, this fundamental property can be expressed through the two main algebraic operations that are addition and multiplication, by taking into account that circled operators $\oplus$ and $\otimes$ are used to identify the specific evaluation functions executed over ciphertext values. In fact, while $\oplus$ often represents a bitwise XOR operation in general computer science, in homomorphic cryptography these symbols serve to denote algebraic operations when applied to encrypted data to perform addition or multiplication on the plaintexts. The operations are:

- **Additive Homomorphism:** It occurs when the sum of two encrypted messages corresponds to the encryption of the sum of the original plaintext values. So, it is true the following relationship:

$$\text{Enc}(a) \oplus \text{Enc}(b) = \text{Enc}(a + b) \quad [11]$$

  By applying the decryption function (Dec) to both sides, we can obtain the following identity:

$$a + b = \text{Dec}\left(\text{Enc}(a) \oplus \text{Enc}(b)\right)$$

- **Multiplicative Homomorphism:** It occurs when the product of two encrypted messages corresponds to the encryption of the product of the original plaintext values. So it is true that:

$$\text{Enc}(a) \otimes \text{Enc}(b) = \text{Enc}(a \times b) \quad [11]$$

  By applying the decryption function, the multiplication performed in the encrypted domain becomes:

$$a \times b = \text{Dec}\left(\text{Enc}(a) \otimes \text{Enc}(b)\right)$$

Thanks to these fundamental operations, the remote computational infrastructure, that is represented by the Cloud server, is able to perform algebraic operations supported by the homomorphic encryption scheme directly on encrypted data. By combining these operations, more complex computations can be performed over encrypted data while the secret key remains exclusively with the data owner. As a result, computational tasks can be delegated to third-party servers without requiring them to access the plaintext information during processing.

From a practical point of view, a homomorphic cryptosystem is different from a traditional cryptographic systems because, while the traditional ones relies on a standard structure composed of three functions: **Key Generation, Encryption and Decryption**, the homomorphic ones introduces a completely new element that is:

the Evaluation algorithm.

Therefore, a homomorphic cryptosystem can be described by taking into account the four fundamental Probabilistic Polynomial-Time (PPT) algorithms:

$$\mathcal{H} = \langle \text{KG}, \text{Enc}, \text{Dec}, \text{Eval} \rangle \quad [11]$$

Here, $\mathcal{H}$ identify the Homomorphic Encryption Scheme. It is used to represent the complete cryptographic framework as a formal tuple, as already described.

1. **Key Generation (KG):** It represents the first function responsible of the generation of the security keys of the system. It uses as input security parameter λ, while as outputs it provides the secret key (sk), the public key (pk) used for encryption operations and the evaluation key (evk), which is required to enable and perform homomorphic computations directly on ciphertexts.

2. **Encryption (Enc):** It represents the algorithm that receives the public key pk and the original plaintext message m belonging to the message space and that is able to transform it into an encrypted representation, that is the ciphertext:

$$c \leftarrow \text{Enc}_{pk}(m)[11]$$

$c \leftarrow$ Ciphertext generated using homomorphic encryption

$c^* \leftarrow$ Ciphertext after performing homomorphic evaluation

$f \leftarrow$ Function used for the computation of the ciphertext

$pk \leftarrow$ Public key

which is unintelligible to anyone who want to read it without having the decryption key.

3. **Decryption (Dec):** It represents the inverse algorithm of Encryption. In fact, it receives as input the private key sk and ciphertext c. Through the private key, it reconstructs the original plaintext message in the following way:

$$m \leftarrow \text{Dec}_{sk}(c)[11]$$

If at this point the algorithm is unable to recover the original message, for example due to data corruption or excessive noise accumulation, it will return as output a formal error symbol $\perp$.

4. **Evaluation (Eval):** It represents the fundamental and unique function introduced by homomorphic encryption that is generally performed by a remote server or Cloud infrastructure. It receives as input, the evaluation key evk, a mathematical or logical function f and a tuple of ciphertexts $(c_1, \ldots, c_t)$ corresponding to plaintext messages $(m_1, \ldots, m_t)$. At this point, without accessing the plaintext context, the algorithm is able to perform the requested operations and will generate a new evaluated ciphertext in the following way:

$$c_f = \text{Eval}_{evk}(f, (c_1, \ldots, c_t))$$

We should also have to take into consideration the principle of correctness. This property ensures that the result obtained by performing an operation on encrypted data corresponds to the result that would have been obtained by performing the same operation directly on the original plaintext data. In other words, after the evaluated ciphertext is decrypted, the result must satisfy the following correctness condition:

$$\text{Dec}_{sk}(\text{Eval}_{evk}(f, (c_1, \ldots, c_t))) = f(m_1, \ldots, m_t)[12]$$

For example, if the function f represents an addition, the server can compute the addition directly on the ciphertexts without knowing the real plaintext values. After decryption, the result must correspond to the sum of the original plaintexts. Therefore, correctness does not require the evaluated ciphertext to be identical to a ciphertext obtained by directly encrypting the final plaintext result:

$$c \leftarrow \text{Enc}_{pk}(f(m_1, \ldots, m_t))$$

The two ciphertexts are considered equivalent with respect to decryption because both produce the same plaintext result. However, they can differ in their internal construction and, in particular, in their accumulated noise level. This distinction is important because homomorphic operations can progressively increase the noise contained in ciphertexts and, if the noise becomes too large, correct decryption may no longer be possible.

Moreover, we can measure how good and how big is the system through two main properties that are:

- **The maximum degree of the function** f: This refers to the maximum degree or complexity of the function f that can be correctly evaluated on encrypted data. It determines how many and how complex homomorphic operations can be performed before the accumulated noise prevents correct decryption.

- **Ciphertext expansion**: This refers to the increase in the size of a ciphertext, measured in bits, as a result of homomorphic operations. If the size growth stays independent of how complex function f is, the scheme is called compact. This property is important because it limits the communication and storage overhead introduced by homomorphic computation, especially in cloud and distributed environments.

Moreover, as the scientific literature shows, a homomorphic encryption model is efficient if the ciphertext size stays within a polynomial limit compared to the original security parameter, even after running several computations.

Homomorphic encryption is not a single cryptographic algorithm, but rather a class of encryption schemes that differ in the type and number of operations they can perform on encrypted data. An **encryption scheme** defines the set of algorithms and keys required to encrypt, decrypt, and, in the case of homomorphic encryption, evaluate computations over ciphertexts. Based on the mathematical operations they support and the amount of computation they can perform, homomorphic encryption schemes can be divided into three main groups:

PHE - Partially Homomorphic Encryption

PHE schemes support only one type of homomorphic algebraic operation, typically either addition or multiplication, which can be applied repeatedly according to the properties of the specific scheme. The classification as partially homomorphic concerns the set of supported operations and does not imply that the underlying encryption algorithm is deterministic or that ciphertexts are free from cryptographic randomness. From a security point of view, homomorphic encryption schemes aim to achieve the **IND-CPA (Indistinguishability under Chosen Plaintext Attack)** security standard. IND-CPA means that an attacker, even when allowed to choose plaintext messages and obtain their corresponding ciphertexts, should not be able to determine which of two selected plaintexts was used to generate a given ciphertext. In simple terms, the encryption must not reveal enough information to distinguish between encrypted messages. To provide this property, encryption must be probabilistic, meaning that encrypting the same plaintext multiple times should produce different ciphertexts due to the use of randomness. Moreover, if we focus on the multiplicative family, standard RSA fails to hit the IND-CPA target because of its deterministic nature. On the other hand, the ElGamal algorithm is a probabilistic public key encryption scheme that provides multiplicative homomorphism, meaning that multiplication between ciphertexts corresponds to multiplication of the associated plaintexts after decryption. Its security is based on the computational difficulty of the discrete logarithm problem. Focusing on additive systems, the Goldwasser-Micali cryptosystem is a probabilistic public key encryption scheme that provides a limited form of additive homomorphism, particularly for XOR operations. It was one of the first schemes able to provide semantic security, although its ciphertext expansion makes it inefficient for encrypting large amounts of data. The Paillier cryptosystem is another probabilistic public key encryption scheme that provides additive homomorphism. In particular, it allows encrypted values to be combined so that, after decryption, the result corresponds to the sum of the original plaintexts. This property makes Paillier useful in applications such as electronic voting, where individual encrypted votes can be combined without revealing each vote separately.

SHE - Somewhat Homomorphic Encryption

SHE schemes are the middle step in the evolution of homomorphic encryption. This type of schemes can handle both addition and multiplication within the same encrypted space, but we can only run a limited number of consecutive operations.In particular, each homomorphic operation contributes to the accumulation of mathematical noise within the ciphertext, with multiplication generally causing a more significant increase than addition. Once the accumulated noise exceeds a certain threshold, the ciphertext can no longer be correctly decrypted. This noise is required to ensure security under the IND-CPA model because without it, the encrypted

data could be simplified into a basic system of equations and easily solved, breaking IND-CPA security. The security of these schemes is generally based on computationally hard mathematical problems, including lattice based problems that are believed to remain resistant to both classical and quantum attacks, such as:

- **The LWE (Learning With Errors) problem:** Introduced by Oded Regev in 2005, LWE is based on the difficulty of recovering a secret vector from a, set of linear equations whose results are affected by small, intentional random errors. Without knowing this error, finding the secret vectors in polynomial time is impossible. Moreover, in homomorphic encryption schemes based on LWE, the error term contributes to the security of the ciphertext, while its accumulation during homomorphic computations progressively limits the number of operations that can be performed before decryption becomes unreliable.

- **The Approximate Greatest Common Divisor (Approximate GCD) problem:** Especially used for schemes that work with integers, this problem is used to make safety the ciphertext by making it a large random multiple of the secret key p, altered by adding a little factor of error $2r$. In this case, the encryption of a message bit m (0 or 1) looks like:

  $$c = q \times p + 2 \times r + m[11]$$

  The introduction of this noise is able to stop an attacker from factoring or finding the greatest common divisor among different ciphertexts. So, during the decryption, the real user can extract the message by calculating $c \pmod p$ and by extracting then the least significant bit.

As we said, when performing computations, if the noise overcomes a certain threshold, the decryption algorithm Dec fails, corrupting the data and making it impossible to recover the original plaintext. Despite this limit, SHE schemes are much faster than fully homomorphic systems. This makes them optimal for operational scenarios that are controlled like financial or medical data analysis, where you know at the beginning the exact number of calculations required.

FHE - Fully Homomorphic Encryption:

The FHE schemes represents the highest development of computational privacy. It allows us to run any type of mathematical or logical operation on encrypted data as many times as we want. To overcome the problem related to the mathematical noise barrier, that limits SHE frameworks, FHE paradigm natively integrates the procedure called **Bootstrapping**. This means that when the accumulated noise gets close to the threshold that would corrupt the data, the system runs a self evaluation. This evaluation consists of an internal decryption and re-encryption of the ciphertext underneath the protective layer using the public key. This process is able to clean and regenerate the data by cutting down the noise level and allowing the Eval algorithm to going on with its execution.

Because of the computational overload related to these constant bootstrapping routines, standard FHE systems are much slower than PHE and SHE. To fix this performance issue, modern engineers have focused on building optimized schemes based on the LWE problem. A first example of this is represented by the **CKKS** scheme, an algorithm born to work with vectors of real and complex numbers in the frequency domain. We are going to describe its mathematical implementation and use in the next sections.

2.3 FHE - Fully Homomorphic Encryption

As already introduced in the previous sections, before the turning point achieved in 2009, all attempts to implement a Fully Homomorphic Encryption scheme based on classical cryptographic approaches had failed. Every operation able to combine both addition and multiplication operations, led either to an uncontrolled expansion of the ciphertext size or to the corruption of the encrypted data. Is at that point that the key to solve these problems was introduced by Gentry through the development of **lattice-based cryptography**.

The great understanding behind this change of paradigm relies on the translation of a cryptographic problem into a multidimensional geometric problem, where information is no longer represented as an isolated numerical values, but rather as spatial coordinates.

To understand the effectiveness of this approach, it is possible to geometrically define an algebraic lattice $\mathcal{L}$ as the set of all linear combinations with integer coefficients of a collection of n linearly independent vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{R}^m$, which represent the basis vectors of the lattice. This can be expressed through the following relation:

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \;\middle|\; x_i \in \mathbb{Z} \right\} [12]$$

Within this framework, the private key is represented by a basis composed of short and almost orthogonal vectors, which allows efficient navigation inside the lattice in order to recover the original information. On the other hand, the public key is generated from a basis made of long and highly distorted vectors, making the inverse computation computationally impossible. As a consequence of that, the encryption process consists of mapping the plaintext message onto an exact point of the lattice and subsequently hiding its precise position by intentionally adding a small random error vector to which we referred as the noise. In this way, for anyone who do not have the private key, it is impossible to solve the geometrical problem and to identify the original lattice point starting from a perturbed position in a space with thousands of dimensions. This infeasible challenge is known in the literature under the name of **Closest Vector Problem (CVP)**, it will require an exponential computational time and is able to make the system mathematically resistant both against classical computers and future quantum machines, providing the system the property of **Quantum-resistance**.

This geometric framework is formally described from an algebraic perspective through the variants of the **Learning With Errors (LWE)** and **Ring-LWE (RLWE)** problems. These two approaches differ significantly in fact, while the standard LWE problem operates through traditional matrix algebra, requiring extremely large matrices that introduce considerable computational overhead and significant ciphertext expansion, its variant **Ring-LWE (RLWE)** moves the computation into the domain of polynomial rings. From a structural point of view, in RLWE both ciphertexts and cryptographic keys are no longer represented by matrices, but as high degree polynomials. This algebraic transition enables polynomial multiplications to be performed more efficiently compared to large matrix products and brings to a reduction of both execution time and the overall ciphertext size. At the same time, when representing ciphertexts as polynomials, this allows us to exploit the ring properties in order to encode and pack an entire data vector inside a single cryptographic structure. This allows the Cloud infrastructure to execute a single homomorphic operation over multiple data simultaneously, exploiting the SIMD (Single Instruction, Multiple Data) execution model.

Therefore, although lattice geometry is able to solve the fundamental challenge of supporting both additions and multiplications, it introduced the most complex engineering challenge in homomorphic encryption that is known as: **the propagation and management of mathematical noise**.

2.3.1 Bootstrapping

It is known that every ciphertext is generated with a small amount of initial noise, which is fundamental in order to mask the encrypted data and guarantee IND-CPA security. However, when the evaluation algorithm Eval performs algebraic operations between ciphertexts, the magnitude of this error undergoes significant variations, in fact:

- **The execution of a homomorphic addition** ($\oplus$): produces a linear addition of the corresponding error vectors. The noise growth remains limited and can be easily tolerated by the system.
- **The execution of a homomorphic multiplication** ($\otimes$): generates cross-products between error components. This operation causes an exponential increase in the overall noise level inside the ciphertext.

Therefore, if the Cloud server continues with the evaluation of the circuit without any noise mechanism of control, the accumulated error would exceed the maximum threshold allowed by the

cryptographic scheme to which we can refer with the term **Noise Budget**. As we known, from a geometric point of view, a ciphertext can be viewed as a vector located near a target lattice node which represent the plaintext. At this point, when homomorphic operations accumulate noise, the ciphertext vector is pushed further away from its origin node. If the noise grows too much, the vector shifts beyond the decoding radius of the lattice cell, preventing the decryption algorithm Dec from mapping it back to the correct lattice node. This brings to the corruption of the encrypted data and make the restoration of the original plaintext infeasible. To overcome this limitation and to enable computations with arbitrary circuit depth, Gentry introduced the concept of **Bootstrapping/Recryption**.

Now, if we consider a somewhat homomorphic encryption scheme (SHE), two independent key pairs $(\mathrm{sk}_1, \mathrm{pk}_1)$ and $(\mathrm{sk}_2, \mathrm{pk}_2)$ and an initial ciphertext $c = \mathrm{Enc}_{\mathrm{pk}_1}(m)$, which protects the message m under the first public key, the ciphertext refresh procedure Bootstrapping, can be described through three sequential phases:

1. **Secret key encryption:** The original private key sk_1 is encrypted and protected using the second public key pk_2:

$$\mathrm{Enc}_{\mathrm{pk}_2}(\mathrm{sk}_1) \rightarrow \overline{\mathrm{sk}_1} \quad [12]$$

   Since the algorithm requires the decomposition of the secret key into binary components, this operation generates a set of intermediate ciphertexts which, once aggregated, represent the evaluation key (evk) required by the Cloud server to perform secure homomorphic computations.

2. **Encryption of the original ciphertext:** The noisy ciphertext c is converted and brought into the domain of the second public key pk_2:

$$\mathrm{Enc}_{\mathrm{pk}_2}(c) = \overline{c} \quad [12]$$

   In most of cryptographic schemes, this operation does not require a real second encryption process due to the fact that $\overline{c}$ is obtained by applying a zero-randomness component, that means that the original ciphertext is treated as a new encrypted object through suitable padding and scaling algorithms.

3. **Homomorphic circuit decryption:** By exploiting the evaluation key evk, the Cloud server performs the homomorphic decryption of the new ciphertext $\overline{c}$ using the protected secret key $\overline{\mathrm{sk}_1}$ and evaluating the function $\mathrm{Dec}_{\overline{\mathrm{sk}_1}}(\overline{c})$. In this way, the remote infrastructure executes the decryption procedure in a blind way where the old noise is mathematically reduced and the original message m is reconstructed and encapsulated inside the second encryption layer, obtaining the encrypted expression: $\hat{\mathrm{Enc}}_{\mathrm{pk}_2}(m)$ This operation can be formalized through the homomorphic evaluation of the decryption circuit using the evaluation key evk:

$$\mathrm{Eval}_{\mathrm{evk}}(\mathrm{Dec}, \overline{c}, \overline{\mathrm{sk}_1}) = \mathrm{Eval}_{\mathrm{evk}}(\mathrm{Dec}, \mathrm{Enc}_{\mathrm{pk}_2}(c), \mathrm{Enc}_{\mathrm{pk}_2}(\mathrm{sk}_1)) = \hat{\mathrm{Enc}}_{\mathrm{pk}_2}(\mathrm{Dec}_{\mathrm{sk}_1}(c)) = \hat{\mathrm{Enc}}_{\mathrm{pk}_2}(m)$$

   The final value $\hat{\mathrm{Enc}}_{\mathrm{pk}_2}(m)$ is mathematically equivalent to a native encryption $\mathrm{Enc}_{\mathrm{pk}_2}(m)$, since both expressions converge to the same plaintext m during the final decryption phase. This allows the user to correctly recover the original data:

$$\mathrm{Dec}_{\mathrm{sk}_2}(\hat{\mathrm{Enc}}_{\mathrm{pk}_2}(m)) = \mathrm{Dec}_{\mathrm{sk}_2}(\mathrm{Enc}_{\mathrm{pk}_2}(m)) = m \quad [12]$$

From an analytical point of view, bootstrapping applies at the same time both the decryption function and a second encryption operation. As shown in Fig.2.1, the accumulation of error generated by successive homomorphic evaluations (Eval$(\cdot)$) may exceed the critical level beyond which decryption becomes impossible.

During this phase, the entire block composed of the previous encrypted message and the corresponding accumulated noise, represented in yellow, is treated as a new plaintext value that

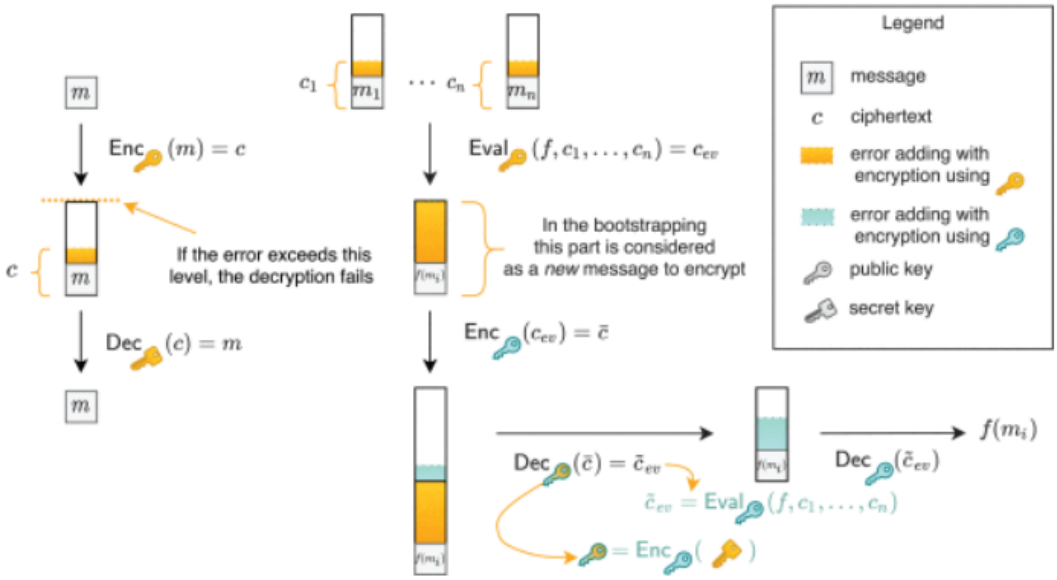


Figure 2.1. Bootstrapping process

must be protected through a second encryption process (Enc(·)). The resulting ciphertext $\bar{c}$ contains a residual error given by the arithmetic sum between the intrinsic noise introduced by the new encryption operation and the additional noise generated by the homomorphic evaluation of the decryption circuit itself. As a consequence of that, even if the error level of the resulting ciphertext is higher compared to a freshly generated ciphertext produced by the user, it remains significantly lower than the noise accumulated after the homomorphic evaluation of functions with a logical depth greater than that of the decryption circuit. This reset mechanism, despite it is computationally complex and expensive, due to the fact that it requires lots of memory resources, it represents the only available approach to transform a limited homomorphic scheme (SHE) into a Fully Homomorphic Encryption system (FHE), able to work with circuits with arbitrary computational depth.

2.3.2 The Four Generations of FHE

The transition from Gentry's theoretical idea to a sustainable and efficient systems has passed through four distinct generations of FHE schemes. Each generation differs both in the fundamental mathematical assumptions and the algebraic structure of ciphertexts, as well as in the methodology adopted for controlling or eliminating the intrinsic noise. A timeline and a schematic representation of the evolution of FHE schemes, together with their respective advantages and disadvantages, are shown in Fig. 2.2 and Fig. 2.3:

First Generation - FHE Based on Ideal Lattices (2009)

In the first generation we can find the original scheme proposed by Craig Gentry, who was able to demonstrate that it was possible to build FHE schemes. To achieve this, Gentry relied on the algebraic framework of ideal lattices introduced by Alice Silverberg, who demonstrated how to map algebraic ideals into geometric lattice spaces. This construction operates over an integer sub-lattice $\mathcal{L}(I) \subseteq \mathbb{Z}^m$ defined by an ideal I.

The ideal I does not represent a simple geometric grid, but rather a special subset of a commutative ring characterized by the absorption property where the product between any element of the ring for an element of the ideal lattice, always will produce another element belonging to the same ideal ones. The introduction of this algebraic structure was the main point of the fundamental idea of Gentry's construction. But what differs is the fact that, while in the traditional lattices the homomorphic addition is immediate, the multiplication between ciphertexts is more complex and tend to move the evaluated data outside the original lattice space. By configuring



Figure 2.2. Timeline of FHE schemes

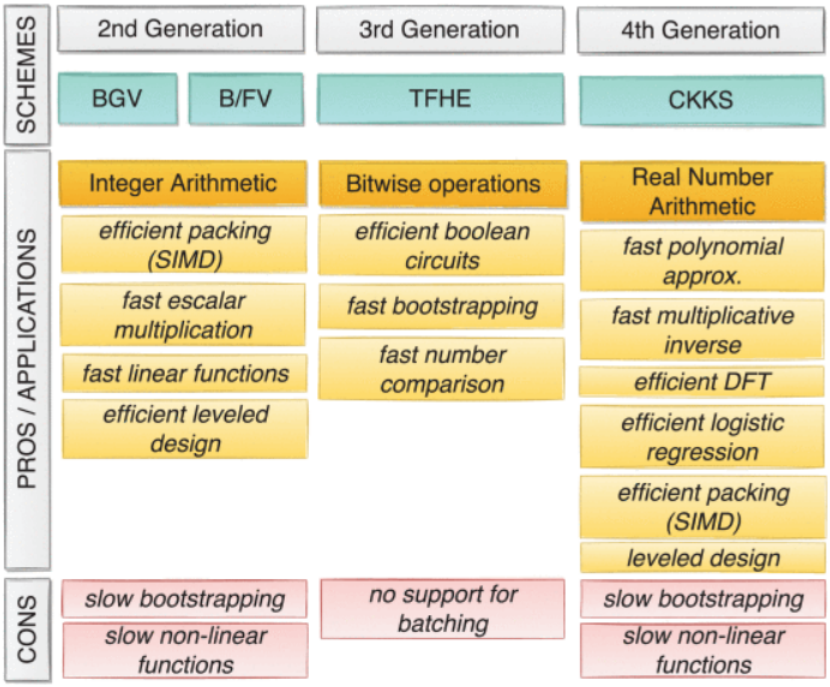


Figure 2.3. Generations of FHE

the lattice such that its points coincide with the elements of the algebraic ideal I, the scheme maintain the absorption property with respect to multiplication. This enables to define a geometrically consistent homomorphic multiplication that is the one able to ensure that the result of a computation, even if able to increase the intrinsic noise level, will remain confined within the same vector space and therefore able to be correctly decryptable.

Within ideal lattice based cryptosystems, encryption and decryption algorithms operate in the following way:

- **Encryption:** The plaintext message, represented by a single bit $m \in \{0,1\}$ is, at the beginning, encoded into a geometric point:

$$a = m + 2e$$

where e represents a small error vector. The final ciphertext c is obtained by translating the point a inside the fundamental parallelepiped $\mathcal{P}(B_{\text{pk}})$ associated to the public key B_{pk}. From a mathematical point of view, the encryption operation can be expressed as:

$$c = a - \left(\lceil a B_{\text{pk}}^{-1} \rfloor B_{\text{pk}} \right) \text{ [12]}$$

where the operator $\lceil \cdot \rfloor$ represents the rounding to the nearest integer.

- **Decryption:** The decryption algorithm receives the ciphertext c and reconstructs the original point by projecting it onto the fundamental parallelepiped $\mathcal{P}(B_{\text{sk}})$ associated with the secret key B_{sk}:

$$a' = c - \left(\lceil c B_{\text{sk}}^{-1} \rfloor B_{\text{sk}} \right) \text{ [12]}$$

The original plaintext is then recovered by applying a modulo-two operation to the obtained value, which removes the error component:

$$m = a' \pmod 2$$

From a geometric point of view, both the public key B_{pk} and the secret key B_{sk} represent two different bases of the same ideal lattice $\mathcal{L}(I)$. However, the public key can be seen as what is defined a bad basis due to the fact that is composed of long, distorted and highly non orthogonal vectors. On the other hand, the private key can be seen as the good basis due to the fact that it presents short and almost orthogonal vectors that enable efficient geometric projection operations, as we can see in Fig. 2.4.

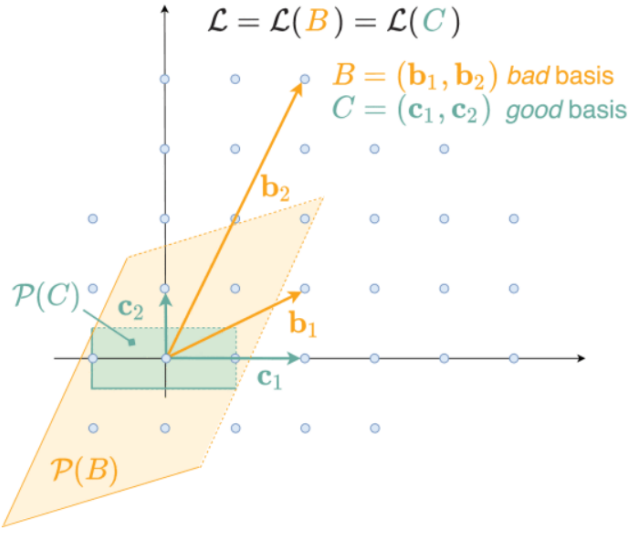


Figure 2.4. Good and bad bases for an ideal lattice

Unfortunately, ideal lattice based schemes do not naturally support bootstrapping due to the excessive complexity of their decryption circuit, which does not allow an homomorphic evaluation in such a short time. To overcome this limitation, Gentry introduced the technique known as **Squashing**.

Through this technique, we are able to transform the original SHE scheme into a new one that presents the same homomorphic capabilities, but characterized by a significantly simplified decryption function. To reduce the complexity of the decryption circuit, additional information about the secret key is added into the evaluation key. This information is represented by a set of vectors defined as:

$$S = \{ s_i \mid i = 1, \ldots, S \}$$

From this set we can extract a sparse subset T, such that the secret key sk can be reconstructed as the sum of the elements belonging to T. The security of this type of construction relies on the complexity of the mathematical problem known as **SSSP (Sparse Subset Sum Problem)**, which consists of determining if exists a sparse subset of elements whose sum is equal to zero.

The first practical implementations of this model were introduced by Smart and Vercauteren. They used principal ideal lattices together with a technique based on the Chinese Remainder Theorem (CRT), an algebraic principle that decomposes complex structures into independent parallel slots. By applying the CRT, the scheme can pack multiple plaintext values into a single ciphertext, allowing a server to compute, on an entire vector of data, simultaneously. Despite several optimization attempts, the main limitation of first generation schemes remained the complexity of their mathematical structures, which made efficient implementations extremely challenging and introduced a prohibitive computational overhead. Furthermore, their security can be vulnerable to quantum attacks. Although the first ideal-lattice constructions were affected by important efficiency limitations and some specific constructions were later subject to structural cryptanalytic attacks, lattice-based cryptography is generally considered a promising foundation for post-quantum security. No general polynomial-time quantum algorithm analogous to Shor's algorithm is known for the lattice problems underlying modern homomorphic encryption schemes.

Second Generation - Leveled FHE Schemes and the RLWE Problem (2011-2012)

In order to overcome the performance limitations of the first generation, the schemes of the second one began to be developed in 2011 with the Brakerski-Vaikuntanathan (BV) scheme and later formalized both through BGV (Brakerski-Gentry-Vaikuntanathan) and BFV (Brakerski-Fan-Vercauteren). They shifted the security topic toward the standard Learning With Errors (LWE) and Ring Learning With Errors (RLWE) problems. The RLWE problem is important due to the fact that it improved computational efficiency by expressing plaintexts, secret keys and ciphertexts as elements of a polynomial ring: $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$

Focusing firstly on the BV scheme, it introduces two fundamental concepts in terms of working with data:

- **Key Switching:** It reduces the ciphertext dimension, which tends to increase after every homomorphic multiplication, by transforming the ciphertext into a new representation associated with a different secret key. This operation allows the scheme to continue performing computations without letting the ciphertext size grow. Key switching is based on the circular security assumption in which information related to a secret key, is encrypted under the same key or under a key that is closely related to it. The assumption is that exposing such an encrypted form of the secret key does not allow an attacker to recover the secret key or break the security of the scheme. So it is relevant in homomorphic encryption because key switching and related techniques may require information derived from a secret key to be encrypted and made available during the evaluation process.

- **Leveled FHE:** This approach allows homomorphic computations to be performed up to a predefined multiplicative depth without requiring the expensive bootstrapping procedure. The multiplicative depth represents the maximum number of consecutive levels of multiplication that the circuit requires. By knowing this depth in advance, the cryptographic parameters can be selected to support the required computation while keeping the accumulated noise within acceptable limits. Therefore, Leveled FHE is suitable for applications where the complexity of the computation is known beforehand.

Furthermore, in order to efficiently manage noise increase, the second generation schemes can be divided into two different mathematical branches:

1. **Modulus Switching (implemented in BGV):** This technique is used because is able to reduce numerical precision in order to proportionally reduce the magnitude of the ciphertext noise. The operation takes a ciphertext $\mathbf{c}$ defined over an initial modulus q_0, scales it down toward a smaller modulus q_1 ($q_1 < q_0$) and applies a rounding operation to the nearest integer. The final equation that defines the new ciphertext in $\mathbb{Z}_{q_1}^{n+1}$ can be expressed as:

$$\mathbf{c}' = \left\lfloor \mathbf{c} \cdot \frac{q_1}{q_0} \right\rceil \pmod{q_1}$$

   When using a small secret key, this scaling operation reduces the noise magnitude proportionally to the modulus reduction:

$$e' \approx e \cdot \frac{q_1}{q_0}$$

By alternating noise accumulation caused by homomorphic operations with progressive modulus reductions at each level of the circuit, the dimension of the parameter is no longer growing in an exponential way, but scales in approximately linear way with respect to the computational depth.

2. **Scale-Invariant Technique (implemented in BFV/FV):**

   This technique rather than reducing the cryptographic modulus level by level as in modulus switching, achieves the same linear noise growth while preserving a constant modulus q throughout evaluation. Here plaintext is embedded directly into the most significant bits (MSB) of the ciphertext through the scaling factor:

$$\Delta = \left\lfloor \frac{q}{p} \right\rfloor$$

   Where p represents the plaintext modulus. The final plaintext reconstruction is performed by isolating the residual error through a division by the ciphertext modulus and a rounding to the nearest integer in the following way:

$$\left\lfloor \frac{p \cdot (c_0 + c_1 \cdot s \pmod q)}{q} \right\rceil \pmod p = m[12]$$

   During homomorphic multiplication, the resulting ciphertext corresponds to a quadratic polynomial:

$$c_1(s)c_2(s) = \alpha_0 + \alpha_1 s + \alpha_2 s^2$$

   Which is then linearized and re-scaled without without altering the modulus q. We can also assert that the FV scheme represents one of the fundamental components of Microsoft's SEAL library, widely adopted for exact arithmetic operations over encrypted integer data.

These techniques enable efficient data parallelization by inserting multiple plaintext values into a single ciphertext thanks to the Chinese Remainder Theorem. Therefore, we can assert that second generation schemes transformed FHE from a theoretical construction into a technology that can be effectively applied to practical circuit implementations with predefined computational depth.

Second Generation - NTRU-Based FHE Schemes

Within the second generation, another research direction is based on NTRU (Number Theory Research Unit), a family of lattice based cryptosystems whose security relies on the difficulty of solving specific problems involving polynomial rings and lattices. The main innovation introduced by this family was the introduction of Multikey FHE (MKFHE), a framework able to perform homomorphic computations over ciphertexts encrypted by different secret keys. This scheme exploits the bootstrapping and modulus switching techniques, in particular it is based on a polynomial secret key f characterized by the property $f \equiv 1 \pmod 2$ and on a public key h. While encryption processes combine the plaintext message with the public key and a noise component, the decryption alghoritm remains linear because it only requires multiplication between the ciphertext c and the secret key f in the following way:

$$m = (f \cdot c \pmod q) \pmod 2[12]$$

The security of the original scheme was based on the DSPR (Decisional Small Polynomial Ratio) problem, characterized by the difficulty of distinguishing the public key from a random polynomial. After that, the YASHE (Yet Another Somewhat Homomorphic Encryption) scheme removed this assumption by adopting a scale invariant approach, although at the cost of larger and more complex evaluation keys. The decline of this family was caused by the discovery of

subfield lattice attacks that were able to compromise the security of the original parameter sets based on DSPR. By applying lattice reduction techniques to this smaller structure, the attacker may recover information about the secret that would be much harder to obtain from the original high dimensional lattice. This can be seen as a reduction of a complex problem into a smaller one that retains the most useful information, making the recovery of the secret computationally more manageable.

If instead, we analyze from a performance point of view, it can be demonstrate that the use of FV and BGV schemes were more efficient and able to generate a lower noise growth compared to NTRU-based approaches. To protect us from the newly discovered attacks, security parameters had to be significantly increased. However, this correction brought to a substantial decrease in terms of performance, making NTRU-based schemes less competitive compared with BGV and FV. As a consequence of that, these schemes are no longer adopted or supported by modern cryptographic libraries.

Third Generation - The GSW Scheme, FHEW and the TFHE Framework (2013-2014)

Developed by Gentry, Sahai and Waters (GSW), the third generation introduced a new algebraic structure of ciphertexts thanks to the introduction of the approximate eigenvector method. The approximate eigenvector method is based on a fundamental concept from linear algebra. An eigenvector is a non-zero vector whose direction remains unchanged when it is multiplied by a matrix, while only its magnitude is modified by a scalar factor called the eigenvalue. In the GSW construction, this concept is adapted to the encrypted domain where the ciphertext is represented by a matrix C, while the secret key vector s behaves as an approximate eigenvector. Ideally, the product $C \cdot s$ would be equal to $m \cdot s$, where the plaintext message m acts as the corresponding eigenvalue. However, due to the noise introduced during encryption, this relation is only approximately satisfied, hence the term approximate eigenvector. This structure allows homomorphic operations to be performed directly on ciphertext matrices while preserving the information required for decryption. This has removed the need to use complex technique of key switching and modulus switching. The decryption operation can therefore be expressed as:

$$C \cdot s = m \cdot I_n \cdot s + R \cdot A \cdot s \approx m \cdot s[12]$$

Since the error introduced by the random matrix R and the public matrix A is sufficiently small, the resulting product converges toward the eigenvalue representation. To recover and extract the plaintext, the system analyzes the first component of the resulting vector, exploiting the fact that is the first component of the secret key to be normalize to one $s_1 = 1$.

The main advantage of this generation is that the homomorphic multiplication of two ciphertexts corresponds to a standard matrix multiplication operation. In this way, the resulting noise accumulation no longer grows in an exponentially way as observed in BGV or FV schemes, but it is reduced to a smaller linear polynomial factor. However, the main disadvantages of the GSW schemes are related to its computational complexity and communication overhead due to the large size of the ciphertext matrices compared with the plaintext representation. To overcome the overhead, researches focused the attention on optimizing the bootstrapping procedure, leading to two important variants:

1. **FHEW:** It introduced a more efficient ring-based construction specifically designed for the homomorphic evaluation of Boolean circuits and a more efficient bootstrapping procedure able to evaluate a function directly on an encrypted value by simultaneously reducing the accumulated noise. In particular, FHEW introduced the concept of **Programmable Bootstrapping (PBS)**, a procedure that can be programmed to evaluate a function represented as a look-up table. In this way, instead of simply refreshing the ciphertext, the bootstrapping can transform an encrypted input into an encryption of the corresponding function output. This allows Boolean gates and other functions to be evaluated efficiently while refreshing the ciphertext and preventing noise accumulation from limiting the computation.

2. **TFHE - Fast Fully Homomorphic Encryption over the Torus:** TFHE can be considered an evolution of the FHEW variants, with the main goal of making bootstrapping faster and more efficient. While FHEW introduced Programmable Bootstrapping (PBS) to evaluate a function during the noise refresh operation, TFHE optimized this mechanism and

made it suitable for efficient evaluation of Boolean circuits. The scheme is based on three cryptographic primitives defined over the Torus:

- TLWE (Torus Learning With Errors);
- TRLWE (Torus Ring Learning With Errors);
- TRGSW (Torus Ring GSW).

The combination of these primitives allow TFHE, with optimized PBS procedure, to simultaneous evaluate an univariate function together with the noise reduction operation. This makes possible to perform a large number of Boolean operations without requiring a separate noise management procedure after each operation. Moreover, a key aspect of this optimization is the use of polynomial operations that can be efficiently implemented through the Fast Fourier Transform (FFT). By transforming convolution type of operations into point wise multiplications in the Fourier domain, techniques based on FFT allow to reduce the computational cost of the operations involved in bootstrapping. This is important because bootstrapping is performed repeatedly during homomorphic evaluation and would otherwise represent a major computational bottleneck.

By using these optimizations and different libraries for the Fast Fourier Transform, third-generation systems turned bootstrapping from a slow, heavy task into something that happens almost instantly. This change meant they could run fast Boolean circuits, making FHE practical for real time application where encrypted math needs to in real time.

Fourth Generation - The CKKS Cryptosystem and Approximate Computation (2017-Present)

Despite the computational efficiency achieved by the previous three generations, these schemes were designed for operating computations over binary values or discrete integers. As a consequence of that, every time we want to perform operations with real values or floating point numbers that require the homomorphic simulation of complex hardware level circuits, this will result in a dramatic performance degradation. In the fourth generation we can overcome this limitation, achievable thanks to CKKS (Cheon-Kim-Kim-Song) scheme which introduces the paradigm of approximate homomorphic computation.

The fundamental innovation of CKKS is that when we consider the intrinsic cryptographic noise, we evaluate it not as an element of disturbance, but as a numerical approximation error similar to the rounding errors that we deal with when working with floating point arithmetic on modern processors. Here, the scheme maps the plaintext space into a complex vector space $\mathbb{C}$ by exploiting primitive roots of unity to encode real and complex values through dense complex packing techniques. The encryption process is constructed by exploiting the Ring Learning With Errors (RLWE) problem and produces a ciphertext of level ℓ defined over a scaled modulus. In this way, the decryption formula is characterized by a linear structure based on the product between the ciphertext and the secret key $sk = (1, s)$, this means:

$$m = \langle c, sk \rangle \pmod{q_\ell} = \beta + \alpha \cdot s \pmod{q_\ell}$$

Unlike previous generations, homomorphic multiplication requires a dynamic management of the ciphertext level. In fact, when two ciphertexts of different levels are multiplied, the system performs a rescaling operation to align their respective levels of encrypted moduli. The rescaling method transforms a ciphertext at level ℓ into a lower level ℓ' by reducing the modulus from q_ℓ to $q_{\ell'}$:

$$c' = \left\lfloor \frac{q_{\ell'}}{q_\ell} \cdot c \right\rceil \pmod{q_{\ell'}}[12]$$

This operation reduces the ciphertext by eliminating the least significant bits. In doing this the accumulated noise is deleted during the computation, while preserving the required decimal precision. Furthermore, compared with previous schemes such as BGV and BFV, CKKS introduces a highly efficient data packing mechanism (batching). This parallelization capability allows

thousands of values to be cleaned from noise at the same time through a single bootstrapping process and to implement faster computation times.

However, approximate cryptographic schemes introduce a problem in terms of security. Indeed, if an attacker is able to access in a passive way both the ciphertexts and the corresponding decrypted plaintexts, he may be able to reconstruct the secret key information using linear algebra, since the approximation error is linearly related to the key components. These attacks can be mitigated by preventing the public sharing of decrypted messages or by modifying the decryption procedure by introducing a small amount of artificial noise at the end of the process.

Today, modern hybrid solutions such as the certificate of CHIMERA and its optimized successor PEGASUS, are able to allow dynamic ciphertext conversion among TFHE (for Boolean gate evaluation), BFV/BGV (for exact integer arithmetic) and CKKS (for approximate numerical computation). In this way, by maintaining arithmetic precision throughout the complete computational pipeline, we can assert that CKKS represents the most suitable FHE scheme for large scale matrix operations, complex signal processing and for privacy preserving implementations of the Fourier-Mellin Transform, which we will analyze and use in the next chapters of this thesis.

2.3.3 FHE for Machine Learning

The bond between homomorphic encryption and machine learning has led to the birth of what is known under the name of **Privacy-Preserving Machine Learning (PPML)**. Nowadays, conventional Machine Learning models require access to massive volumes of plaintext data in order to achieve high levels of accuracy. However, this requirement introduces significant challenges about the confidentiality, privacy and security of sensitive information and data. It is thanks to the introduction of FHE schemes and their ability to support complex arithmetic evaluations directly over encrypted data, that now is possible to both train models and perform predictions directly on encrypted data. However, machine learning algorithms cannot be directly deployed within an FHE environment. They must be completely redesigned and optimized according to the mathematical constraints imposed by the selected cryptographic scheme. In fact, the fundamental architectural trade off we have to respect is: There is no single cryptographic library or FHE scheme that is suitable for every type of computational workload. The most appropriate choice depends on how and where the computation needs to be performed, as different applications may require different mathematical structures and computational properties. Therefore, the characteristics of the workload help to determine the most suitable mathematical structure and, consequently, the FHE scheme to use. From this trade off, two opposite computational strategies arise:

The Cloud-Oriented approach (CKKS Library)

The CKKS scheme, thanks to its support for floating point arithmetic and efficient ciphertext packing techniques, represents the most suitable solution for large scale and centralized Cloud computations. It is particularly suitable for the training and solutions of complex neural networks, which are computational models inspired by the structure of the human brain and composed of interconnected layers of artificial neurons that learn patterns from data. It can also be applied to linear and logistic regression models, as well as Support Vector Machines (SVMs). SVMs are supervised learning models used for classification tasks, where the goal is to identify the optimal decision boundary that separates data belonging to different classes. However, leveled CKKS schemes are not able to work with complex and nonlinear operators without taking into account the approximation through polynomials, which may degrade the accuracy of the machine learning model. Furthermore, the huge memory overhead introduced by CKKS cryptographic parameters makes it unsuitable for application on individual edge devices with limited computational and storage resources.

The Edge-Oriented approach (TFHE Library)

The TFHE-based approach represents the best solution for addressing the accuracy limitations of CKKS while enabling exact computations on devices with limited computational and memory resources. TFHE enables the evaluation of Boolean functions, requiring lower memory consumption compared with CKKS based solutions. This makes it suitable for implementing sensitive algorithms such as K-Nearest Neighbors (KNN) directly on edge infrastructures. KNN is a

supervised learning algorithm used for both classification for predicting a category, and regression tasks where the goal is to predict a continuous numerical value. Its basic idea is simple: when a new sample needs to be classified, the algorithm compares it with the samples already present in the training dataset and identifies the k most similar ones according to a distance metric. The new sample is then assigned to the class that is most common among these nearest neighbors. Since the algorithm relies on comparisons between data points rather than on a complex model, it can be particularly useful for privacy-preserving machine learning on edge devices. Nevertheless, due to the absence of dense ciphertext packing compared to the CKKS scheme, TFHE introduces higher latency for individual operations, leading to slower execution times.

In order to mitigate the computational limitations of individual edge nodes and improve the overall efficiency of the system, modern architectures start to be oriented toward distributed homomorphic computation across multiple edge devices. Using this approach means of partitioning encrypted data among multiple edge nodes and transmitting ciphertexts through dedicated APIs (Application Programming Interface) within a distributed cluster of peripheral nodes. This allows different nodes to process separate parts of the homomorphic computation in parallel, reducing the computational load on individual devices and making privacy-preserving models more feasible on resource constrained hardware.

The resulting architecture is based on a complementary use of CKKS and TFHE rather than on the assumption that a single FHE scheme can efficiently handle every type of computation. CKKS is more suitable for dense numerical and linear operations, particularly in Cloud based environments with greater computational and storage resources, while TFHE is better suited for bit level Boolean operations that require exact and non approximated results at the Edge. The trade off therefore concerns the choice between computational efficiency, supported operations, accuracy and available resources: CKKS provides efficient packed numerical computation but relies on approximations and requires significant resources, whereas TFHE provides exact computation and is better suited to resource constrained devices, at the cost of higher latency for individual operations.

The practical impact of choosing CKKS or TFHE becomes evident when comparing the execution performance of different Machine Learning algorithms. Since each algorithm relies on different types and combinations of operations, the computational cost of evaluating them over encrypted data can vary significantly depending on the selected FHE scheme and execution environment. So the comparison between Cloud-based CKKS and distributed Edge-based TFHE provides a useful way to evaluate how these differences affect the practical performance of privacy-preserving Machine Learning. In particular, experimental results reported by [13], demonstrate the following key outcomes:

- **Support Vector Machine (SVM):** Homomorphic inference requires approximately 18 seconds in a Cloud environment using CKKS compared to 4.15 minutes when working in a distributed Edge environment based on TFHE. [13]
- **Logistic Regression (LR):** The computation requires approximately 17 seconds in the Cloud using CKKS, whereas the distributed Edge implementation based on TFHE requires approximately 7.82 minutes. [13]
- **Neural Networks (NN):** Neural network inference achieves the best performance within a Cloud-based CKKS environment, requiring approximately 57 seconds. [13]
- **K-Nearest Neighbors (KNN):** KNN represents the most critical computational bottleneck in encrypted machine learning environments. In fact, while the execution in the Cloud using CKKS requires approximately 248 seconds, the distributed Edge implementation based on TFHE needs up to 37 minutes. This result highlights that specific non linear algorithms require dedicated mathematical optimizations to be used within fully encrypted environments. [13]

Today, several companies and research organizations are working to make FHE more practical and suitable for real world applications. Among them, Zama [14] is one of the companies focusing on the development of open source FHE technologies and tools for integrating homomorphic encryption into existing software and machine learning workflows. In fact it has developed compilers that can automatically and transparently translate standard Machine Learning models

into homomorphic executable code, without requiring significant changes to the original model or its development workflow, enabling blind inference over encrypted data. At the same time, major technology companies such as Intel and Ant Group are studying a way to enable the integration of FHE with the differential privacy technique and with protected hardware execution environments. Furthermore, organizations such as Duality Technologies are developing dedicated hardware accelerators, such as ASIC (Application Specific Integrated Circuit) and FPGA (Field Programmable Gate Array) architectures of which we will discuss later. They are used with the aim of reducing computational bottlenecks and enabling real time homomorphic machine learning.

2.3.4 HE in Cloud Computing

Cloud Computing refers to a system in which computing resources are provided by a service provider to an end users through the Internet. Services such as data storage, computing power or communication services, all of which are made possible by a distributed infrastructure made up of existing, configurable and remotely accessible resources.

Within the cloud computing framework, hardware and software resources are not configured by the provider exclusively for a single user, but are dynamically allocated through automated procedures from a shared pool of resources used by multiple users. At this point is the final user to be responsible for the configuration of them. After that, once the resource is released by the user, it is automatically reconfigured and returned to the shared pool for future allocations.

Nowadays, cloud computing has become the standard choice for managing large scale datasets, for sending data and for executing intensive tasks through external and shared servers. However, this model introduces a fundamental security trade off in which users must choose between the convenience of relying on highly powerful remote machines and the security of their private data.

Homomorphic Encryption provides a possible solution to this problem in fact, it enables cloud servers to process data even if it remains encrypted in the form of ciphertexts. In this way, the server performs the requested computations directly on encrypted information and returns an encrypted output which can only be decrypted by the user who owns the corresponding private key. Nevertheless, moving from theory to practice is not so simple. In fact, performing computations over encrypted data requires lots of computational resources. This leads us to have to deal with the continuous management of the mathematical noise accumulated during homomorphic operations, but also with the fact that the user is forced to blindly trust the cloud provider that is defined to be **"Honest-But-Curious"** from a security perspective. This means that we assume the server to be correctly executing the computations, but it may be interested in extracting information from the processed data.

Today, modern secure computing architectures do not rely on just one cryptographic scheme, but they combine different encryption schemes depending on the specific computational workload. To reduce computational costs and ensure the latency requirements of cloud applications demanded by users, the cloud infrastructure optimizes the execution pipeline. Especially by organizing operations according to their multiplicative depth; that means depending on the number of sequential multiplications we want.

Furthermore, to avoid slowing down the system, large scale encrypted computations can be divided into smaller workloads and parallelized across high performance GPUs (Graphics Processing Units), specialized processors designed to execute a large number of operations simultaneously. These computations can also exploit algorithms such as the NTT (Number Theoretic Transform), a mathematical transform similar in concept to the Fast Fourier Transform (FFT) but defined over finite modular arithmetic, able to accelerate polynomial multiplication and other operations commonly used in homomorphic encryption. [15]

However, deploying this homomorphic cloud computing model in real world and multi-user environments introduces additional requirements that traditional HE schemes alone cannot efficiently address, such as: [12]

Homomorphic Proxy Re-Encryption - HPRE

When a cloud service process has to encrypt information belonging to different users in parallel sessions, standard HE schemes are no longer useful because they generally support operations only between ciphertexts encrypted under the same public key. Therefore, Proxy Re-Encryption (PRE) techniques become necessary.

Proxy Re-Encryption (PRE) is a technique that allows a proxy entity, such as a cloud server, to transfer encrypted data from one authorized user to another. This means that the proxy can transform a ciphertext encrypted under the public key of one user into a ciphertext that can be decrypted using the public key of another authorized user. The proxy performs this transformation without accessing the plaintext or the secret keys of either user. In this way, data can be securely shared with another authorized party without requiring the original ciphertext to be decrypted and re-encrypted. The process is shown in Fig. 2.5.

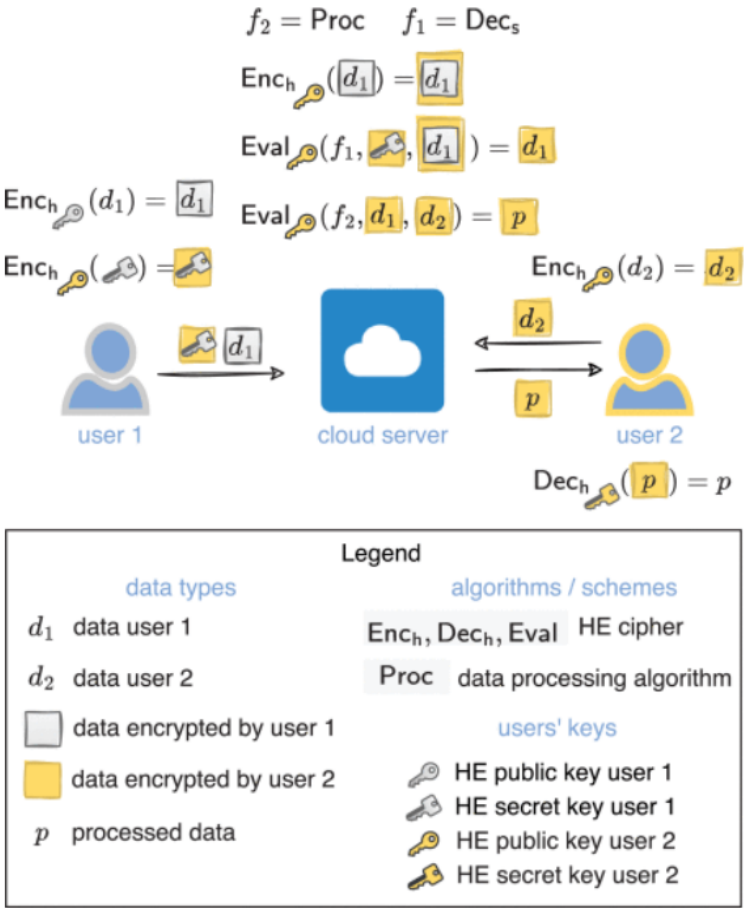


Figure 2.5. Proxy Re-Encryption

The first approach to construct Homomorphic Proxy Re-Encryption can be traced back to Gentry's FHE construction, where re-encryption can be achieved by evaluating the decryption circuit homomorphically. This was a technique in which the cloud evaluates the decryption algorithm directly over encrypted data by using an encrypted representation of the owner of the secret key. In this way, data can be transferred from one encryption domain to another without ever being revealed in plaintext. However, the solution proposed by Gentry suffered from collusion attacks, a situation in which the cloud and the delegated user could combine their information to violate the privacy of the original user and to try to calculate his private key. In order to overcome this problem, key switching techniques can be used to transform ciphertexts between different encryption keys while providing stronger resistance to collusion attacks.

Homomorphic Authenticated Encryption - HAE

In sensitive or commercial cloud applications, the user must verify that the server has executed the requested computations in the correct way. In fact, a malicious cloud provider could return an incorrect result or claim to have performed the computation without actually executing it. By doing so, the provider could avoid the computational resources required for homomorphic operations and reduce its costs, while the user would have no immediate way to determine whether

the result is correct. This problem is solved by Homomorphic Authenticated Encryption, as illustrated in Fig. 2.6, which combines homomorphic encryption with authentication mechanisms such as digital signatures or homomorphic message authentication codes (MACs).

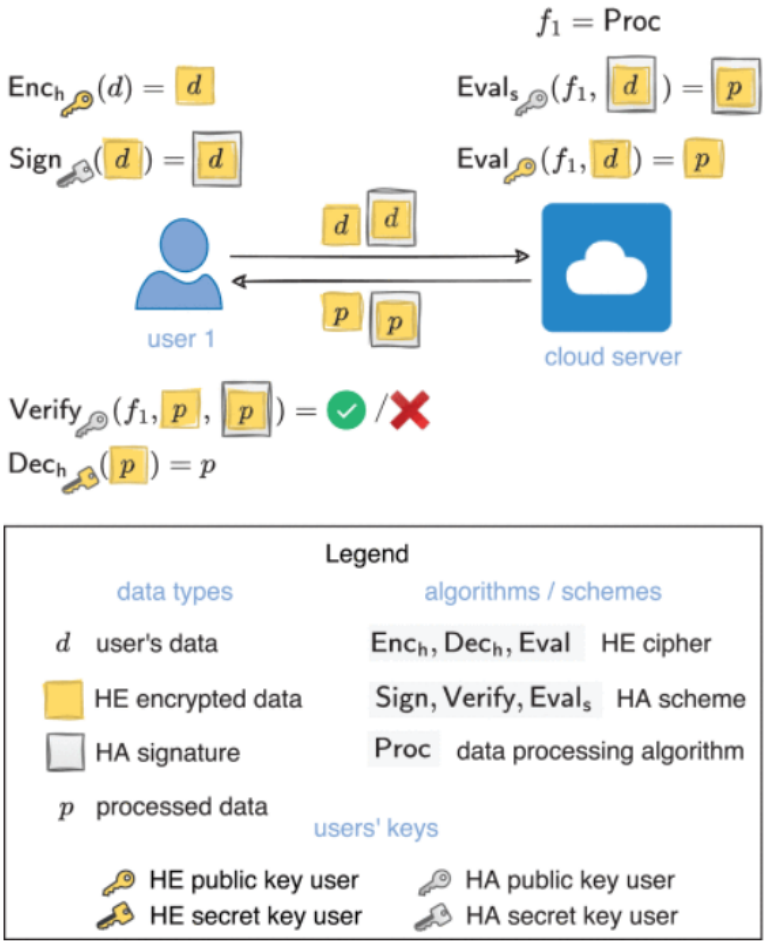


Figure 2.6. Authenticated encryption

As we can see when the user sends data, he associates to each encrypted element an authentication signature. In this way, while performing computations over encrypted data, the cloud simultaneously updates these signatures according to the evaluated function. At the end of the computation, the server returns both the encrypted result and the updated authentication signature. If the authentication signature is valid, the user has a mathematical assurance that the computation has been correctly executed. This mechanism can strengthen the security of the system against stronger attack models, such as CCA1 (Chosen-Ciphertext Attack). In this security model, an attacker can ask the system to decrypt selected ciphertexts and observe the corresponding plaintexts. The attacker then receives a challenge ciphertext and must determine which of two possible plaintexts was used to generate it. The challenge ciphertext is therefore the encrypted message that the attacker is trying to distinguish. A CCA1 secure scheme ensures that the information obtained from the previous decryption requests does not help the attacker identify the plaintext contained in the challenge ciphertext. Alternatively, the same guarantees can be achieved through external verification mechanisms or ZKP (Zero Knowledge Proof) systems. These techniques allow the Cloud provider to generate a valid proof that the required computation has been correctly performed, without revealing the underlying encrypted data or other sensitive information. In this way, the user can verify the correctness of the computation without directly accessing or trusting the internal operations performed by the Cloud.

Multi-Party Computation - MPC

When multiple users want to merge their private datasets for a common analysis, but without disclosing the underlying information to each other, secure Multi-Party Computation (MPC) protocols become necessary as in Fig. 2.7).

In this context, Homomorphic Encryption plays a key role during the initial pre-processing phase, as this step generates the cryptographic material required for subsequent computations. If we are able to do this in advance, HE-based MPC protocols can significantly reduce the online computational overhead. This approach, based on HE, is often more efficient than traditional cryptographic techniques such as Oblivious Transfer (OT), a protocol that allows a user to take a specific piece of information from a server without revealing which element was selected and preventing the user from accessing the remaining information.

However, OT-based approaches may require a large number of communication rounds and therefore can introduce considerable network overhead. Finally, advanced HE extensions, such as Multi-Key Fully Homomorphic Encryption (MKFHE) and Threshold Multi-Key Fully Homomorphic Encryption (TMFHE), allow several users to collaborate while reducing the need for repeated interaction during the computation.

- **MKFHE:** Here each user independently encrypts their data under their own secret key. The resulting ciphertexts can then be sent to a common Cloud server, which can perform homomorphic operations directly on ciphertexts associated with different keys. The computation therefore produces a ciphertext that depends on the secret keys of all participating users, without requiring the Cloud to access any of the corresponding plaintexts or secret keys. At the end of the computation, the users cooperate to decrypt the final result.

- **TMFHE:** It extends the MKFHE approach by introducing a threshold decryption mechanism. Instead of requiring every user to provide their complete secret key, the decryption capability is distributed among the participants through partial decryption shares. When the Cloud completes the homomorphic computation, each authorized user generates a partial decryption share using their secret key. A minimum number of valid shares, defined by the chosen threshold, must then be combined to recover the final plaintext. As a result, no single participant is sufficient to decrypt the result independently, while the Cloud can perform the entire computation without accessing the underlying data. In this way, data remains protected throughout the computation, while the final result can only be recovered through the collaboration of a sufficient number of authorized users.

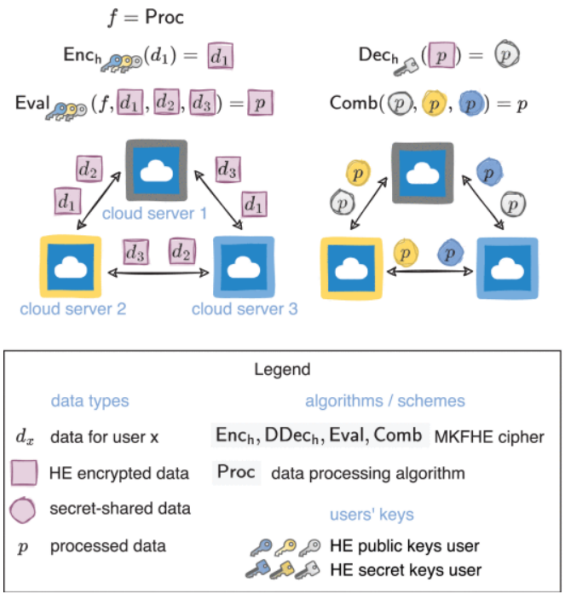


Figure 2.7. Multi-Party Computation

2.4 FHE Libraries

The main goal of software libraries is to provide developers with reusable implementations of complex functionalities (ready-to-use tools). Allowing them to integrate advanced features into an application without implementing every underlying mechanism from scratch leading to a speed up in the implementation. This abstraction is particularly valuable in the context of FHE, where the implementation of cryptographic schemes involves complex mathematical operations and careful management of ciphertexts. FHE libraries expose these operations through structured APIs, making the main cryptographic primitives accessible to application developers. Besides providing the core functions like key generation (KeyGen), encryption (Enc), decryption (Dec) and homomorphic evaluation (Eval), almost all modern libraries include complex logic to manage ciphertexts, controlling cryptographic noise, parameter selection, ciphertext transformations, and, depending on the scheme, operations such as bootstrapping and key switching. This allows developers to use FHE without directly implementing the underlying cryptographic primitives and mathematical procedures. However, the use of a library does not eliminate the need for a proper understanding of the underlying cryptographic scheme. In fact, choosing the right library and using it correctly is something under the control of the developer. The developer remains responsible for selecting appropriate parameters, using the available APIs correctly, and understanding the security and computational implications of each operation. Incorrect parameter choices or inappropriate use of the library can lead to performance degradation, decryption errors or security vulnerabilities.

The growing adoption of open-source FHE software has also led to the development of different libraries with distinct design choices in terms of programming languages, supported encryption schemes, optimization techniques, and hardware acceleration. Some libraries are designed to provide general-purpose support for multiple FHE schemes, while others focus on specific schemes or applications. The main FHE libraries considered in this work are summarized in Table. 2.1:

Library	Language	Scheme					Latest Release/Status
		BGV	B/FV	FHEW	TFHE	CKKS	
HElib [16]	C++	●	○	○	○	●	v2.3.0 - 18/07/2023
SEAL [17]	C++/C#(.NET)	●	●	○	○	●	v4.4.4 - 24/08/2026
PALISADE [18]	C++	●	●	●	●	●	v1.11.9 - Legacy
Lattigo [19]	Go	●	●	○	○	●	v6.2.0 - 02/02/2026
FHEW [20]	C++	○	○	●	○	○	v2.0beta - Legacy
TFHE [22]	C++/C	○	○	○	●	○	v1.1 - Legacy
Concrete [24]	Python/Rust	○	○	○	●	○	v2.10.0 - 10/04/2025
HEAAN [25]	C++	○	○	○	○	●	v2.1 - Legacy
RNS-HEAAN [28]	C++	○	○	○	○	●	2018 - Legacy
FV-NFLlib [30]	C++	○	●	○	○	○	2016 - Legacy
CuFHE [31]	Cuda/C++	○	○	○	●	○	v1.0beta(2018) - Legacy
NuFHE [32]	Python	○	○	○	●	○	v0.0.3(2019) - Legacy
OpenFHE [33]	C++	●	●	●	●	●	v1.5.1 - 10/04/2026
TenSEAL [35]	Python/C++	○	●	○	○	●	v0.3.17 - 04/08/2026

Table 2.1. Comparison of FHE libraries and supported schemes (information refers to the status of the libraries as of September 2026).

HElib

It is one of the oldest and most important open-source libraries for Fully Homomorphic Encryption. It is written in C++ and relies on the NTL (Number Theory Library) math library. From a cryptographic point of view, HElib naturally supports two main schemes: BGV, which uses bootstrapping to extend the supported computational depth for exact arithmetic on integers, and the CKKS, designed for fast, approximate arithmetic on real and complex numbers. One of the main advantage of HElib is its large set of algorithmic optimizations designed to cut down the heavy processing costs of encrypted computations. In particular, this library focuses both on advanced ciphertext packing techniques, which allow to pack multiple messages to be stored into a single ciphertext so they can be processed in parallel according to SIMD paradigm and with

several mathematical optimizations. Today, the library offers a low-level assembly language for homomorphic encryption that means it provides native routines for basic steps like setting up parameters, adding, multiplying and shifting data. This low-level approach gives developers greater control over the execution of homomorphic computations and allows them to take advantage of the optimizations provided by the library. In addiction, it includes an automatic system to track and downscale mathematical noise, supports multi-threading and it introduces features plaintext objects that act just like encrypted data, simplifying code development. [16]

SEAL

Microsoft SEAL is one of the most widely used open-source libraries for homomorphic encryption. Developed primaly in C++, it is known for its reliability and native support for two of the main homomorphic encryption schemes: the exact BFV scheme, which is well suited for computations on integers and the CKKS. Microsoft SEAL presents a robust and optimized API that allows raw data to be encoded into plaintext objects through the use of `BatchEncoder` or the `CKKSEncoder` and then encrypted them into ciphertexts using the `Encryptor`. In addition to that, through the `Evaluator` object, it supports not only basic arithmetic operations such as addition and multiplication, but also advanced ciphertext management functions, including cyclic vector rotations, relinearization to control and reduce the size of ciphertexts after multiplication and modulus switching for managing the ciphertext modulus and computational levels. Unlike fully bootstrappable FHE libraries, SEAL does not provide native bootstrapping, so the supported computation is limited by the selected encryption parameters and available multiplicative depth. Therefore, Microsoft SEAL is widely used for the development of real and privacy-preserving applications, although it requires careful selection of the initial encryption parameters to ensure correct configurations and sufficient computational depth. [17]

PALISADE

It is one of the most complete and well-structured open-source libraries for lattice-based cryptography. Developed in C++, it is a cross-platform library that natively supports Linux, Windows and macOS without requiring external dependencies. The main characteristic of PALISADE is its cryptographic flexibility in fact it supports homomorphic encryption schemes for both integer and real number arithmetic (BFV, BGV, and CKKS), but also for schemes designed for the evaluation of Boolean circuits (FHEW and TFHE). In addition, this library integrates advanced multi-user extensions for advanced multi-user features such as threshold homomorphic encryption and Proxy Re-Encryption protocols for the BGV, BFV and CKKS schemes. From a software engineering point of view, PALISADE presesnts a modular architecture organized into four main layered that are: low-level mathematical operations, lattice operations, the cryptographic layer and the encoding layer. Another important feature is its use of runtime polymorphism, which provides a common and consistent API across all supported encryption schemes and makes the library easier to use. PALISADE also served as the foundation for OpenFHE, its successor, which is currently the main project for the continued development of the library. [18]

Lattigo

It is an open-source library developed in Go designed for the implementation of homomorphic encryption and lattice-based cryptographic primitives. It is mainly based on the RLWE mathematical problem and provides efficient implementations of schemes such as BFV, BGV, and CKKS, with extensive use of the Full-RNS (Residue Number System) representation, to improve the efficiency of homomorphic operations. The library also provides support for multi-user and distributed cryptographic protocols, making it suitable for applications in which encrypted computations need to be performed across different parties. The choice of using Go as the programming language provides native support for concurrency and facilitates the development of networked and distributed applications, which can be useful in Cloud and microservice-based architectures. Lattigo also supports WebAssembly, allowing parts of its cryptographic functionality to be executed in web environments. From an architectural point of view, the library follows a layered structure in which higher-level cryptographic functionalities are built on top of lower-level mathematical and polynomial operations. This modular organization allows developers to

work at different abstraction levels while maintaining control over the underlying homomorphic operations. [19]

FHEW

It is a revolutionary library for Fully Homomorphic Encryption, mainly designed for efficient computation on encrypted bits. Its innovation is the introduction of fast bootstrapping mechanism which refreshes a ciphertext by reducing the noise accumulated during homomorphic computations on encrypted data allowing further operations to be performed on the same ciphertext. FHEW is suited for evaluating Boolean functions and supports basic logical operations such as AND, OR, NAND, and NOR directly on encrypted data. This makes it possible to build complex Boolean circuits in the encrypted domain, where a Cloud server is able to make logical decisions on a user's data without having the access to its real content. FHEW is based on the Learning With Errors problem and introduced techniques that significantly contributed to the development of later bootstrapping-based homomorphic encryption schemes. Although originally was developed primarily as a research prototype and is no longer actively maintained, its bootstrapping approach had played an important role on the evolution of Fully Homomorphic Encryption. [21] [20]

TFHE (Fast Fully Homomorphic Encryption over the Torus)

It is an open-source library developed in C++ and born as the optimized evolution of FHEW, with a particular focus on efficient bootstrapping and Boolean computation. For this reason, it is considered to be one of the most efficient solutions for evaluating Boolean circuits over encrypted data. Its main advantage is related to the high speed of the gate-by-gate bootstrapping mechanism. In TFHE, bootstrapping is integrated into the evaluation of individual logic gates, allowing ciphertexts to be refreshed and their accumulated noise to be reduced after each operation. This makes it possible to evaluate arbitrarily complex Boolean circuits without requiring the original data to be decrypted. From a mathematical point of view, TFHE is based on the LWE and RLWE problems and uses a variant of the GSW encryption technique together with arithmetic over the torus. To further reduce computation time, the library uses the Fast Fourier Transform together with hardware-level acceleration instructions, including SIMD instructions such as AVX, AVX2, and FMA. Moreover, unlike many other cryptographic libraries, TFHE does not present a predefined limit on the number of Boolean gates that can be evaluated sequentially, nor the depth of the Boolean circuit. This makes TFHE particularly suitable for applications requiring frequent evaluation of Boolean functions on encrypted data. Finally, even if the original C++ implementation is no longer actively maintained, its underlying techniques have strongly influenced more recent homomorphic encryption projects, including Concrete by Zama and the Rust-based TFHE-rs library. [22] [23]

HEAAN

It is an open-source C++ library that was born as the implementation of the CKKS scheme in order to support fixed-point arithmetic and approximate computations on real and complex numbers. Unlike cryptosystems based on exact arithmetic, HEAAN allows a small and controlled approximation error, similar to the rounding errors that occur in conventional floating-point computations. This library provides the main operations required for homomorphic computation, including key generation, encryption, decryption, and arithmetic operations on ciphertexts. Although it does not include the bootstrapping mechanism for removing the accumulated noise; however, later versions introduced bootstrapping techniques to extend the computation that can be performed on encrypted data. The introduction of HEAAN and the CKKS scheme represented an important step towards practical homomorphic computation on numerical data, as they made it possible to efficiently represent and process real and complex values in the encrypted domain. This characteristic makes CKKS-based approaches particularly suitable for applications involving machine learning, statistical analysis, and other computations on numerical data that can tolerate a limited approximation error. [26] [25]

RNS-HEAAN

It represents an important evolution of the original HEAAN library, it was developed in order to overcome the performance limitations caused by the manipulation of large integers in the CKKS scheme. The main idea is to use the Residue Number System technique, which expresses large integers as a set of residues with respect to several smaller, pairwise co-prime moduli. These smaller values can be processed independently and are typically chosen to fit within the native word size of modern processors, such as 64-bit registers. This decomposition significantly reduces the cost of large-integer arithmetic and allows the underlying operations to be efficiently parallelized. RNS-HEAAN also makes extensive use of the NTT transformation to efficiently perform polynomial multiplication in modular arithmetic. By combining RNS decomposition with NTT-based polynomial operations, the library can significantly improve the efficiency of the homomorphic operations required by CKKS. These optimizations contributed to make RNS-based implementations an important approach for practical CKKS computation and later became widely adopted in modern homomorphic encryption libraries, particularly for applications involving large numerical workloads such as machine learning and data analysis. [27] [28]

FV-NFLlib

It is an open-source and header-only C++ library, meaning that it is composed of header files which makes it easy to integrate because does not require a separately compiled library during the linking stage, making it relatively lightweight to integrate during the C++ compilation process. It is mainly used because it provides a pure implementation of the FV scheme for integer arithmetic. A distinctive feature of this FV-NFLlib is that it does not implement mathematical functions starting from zero knowledge, but it starts its implementation thanks to NFLlib, a low-level library providing optimized arithmetic operations over finite fields and polynomial rings. Instead of implementing all the underlying mathematical primitives independently, FV-NFLlib builds its homomorphic encryption functionality on top of the optimized components provided by NFLlib. FV-NFLlib includes also advanced testing and benchmarking tools designed to evaluate the performance of homomorphic operations and monitor the evolution of ciphertext noise during computations. These tools were used to demonstrate the feasibility of relatively complex computations over encrypted data, including the homomorphic evaluation of elliptic-curve point addition. Although FV-NFLlib is no longer actively maintained, it represents an early example of how optimized low-level arithmetic libraries can be combined with homomorphic encryption schemes to improve the efficiency of encrypted computations. [29] [30]

CuFHE

It is an open-source library developed in CUDA and C++ with the aim of reducing the computational cost of homomorphic encryption by exploiting the parallel processing capabilities of NVIDIA GPUs. The library implements the TFHE scheme at hardware level and is designed for gate-by-gate evaluation of Boolean circuits, supporting native operations such as AND, OR, NAND, XOR and NOT. Its main advantage is the improved performance of the gate bootstrapping mechanism, which is significantly faster by exploiting GPU parallelism compared with a CPU-only implementation of TFHE. This improvement is achieved through a highly optimized CUDA implementation of the NTT. From a software point of view, CuFHE provides two separate components, `libcufhe_cpu` and `libcufhe_gpu`, together with a Python wrapper that facilitates its integration into higher-level applications. The GPU implementation allows multiple independent operations involved in the evaluation of Boolean circuits to be processed in parallel, making CuFHE particularly suitable for workloads containing a large number of homomorphic operations. As a result, CuFHE represents an early example of how GPU acceleration can significantly improve the performance of homomorphic encryption and reduce the latency associated with Boolean computations on encrypted data. [22] [31]

NuFHE

It is an open-source library that provides a Python interface for the TFHE homomorphic encryption scheme and is designed to accelerate its execution by exploiting the computational capabilities of GPU devices. Unlike CuFHE, which is specifically designed for NVIDIA GPUs using

CUDA, NuFHE supports both CUDA and OpenCL, providing greater hardware flexibility and allowing it to run on a wider range of GPU devices. From a mathematical point of view, it allows the user to choose if we want to perform a multiplication using classical FFT or using integer-based NTT methods derived from cuFHE. This flexibility allows the implementation to adapt the computation to different hardware configurations and workloads. The main performance advantage of NuFHE comes from the massive parallelism provided by GPUs, which can significantly accelerate the computationally intensive operations involved in TFHE bootstrapping. Experimental evaluations reported by the project demonstrate substantial speedups compared with CPU-based execution, particularly for workloads involving a large number of homomorphic Boolean operations. NuFHE therefore represents an important example of how GPU acceleration and hardware portability can be combined to improve the practical performance of homomorphic encryption. [31]

TenSEAL

It is an open-source library developed by OpenMined that simplifies the use of homomorphic encryption through a Python interface built on top of Microsoft SEAL. It is mainly designed for privacy-preserving machine learning applications and numerical computations, allowing operations to be performed directly on encrypted data that are represented as vectors and matrices, hiding the complexity of the underlying cryptographic computations. The library supports both BFV and CKKS schemes, enabling the operations such as addition, multiplication adot products, and other vector and matrices computations on encrypted data. Thanks to its C++ core and Python interface, TenSEAL presents the computational efficiency of the underlying Microsoft SEAL implementation with a high-level and accessible programming interface. Its main advantages is therefore the reduction of the programming effort required to integrate homomorphic encryption into machine learning and data-processing applications. However, this higher level of abstraction also introduces some limitations. TenSEAL is primarily focused on BFV and CKKS, depending on the performance of Microsoft SEAL, and does not provide native support for advanced features such as bootstrapping or GPU acceleration. For these reasons TenSEAL is still used in academic environments, or applications that require an accessible interface for encrypted numerical computation, while more general-purpose libraries such as OpenFHE may be preferable when broader scheme support and more advanced cryptographic features are required. [34] [35]

2.4.1 The Concrete Library

In the field of Fully Homomorphic Encryption, one of the most difficult obstacles to overcome has always been the high-level of mathematical knowledge required for its implementation. To reduce the gap between theoretical research and practical software development, the cryptographic company Zama developed Concrete. Concrete is an open-source framework designed to simplify and automate the adoption and use of FHE technology. The main goal of this framework is to enable complex computations to be performed directly on encrypted data without the need to decrypt it. This guarantees the user complete privacy and provides strong protection against cyber attacks. In fact, since all operations performed by the server are executed only on encrypted data, sensitive information remains protected and inaccessible even if the hosting cloud infrastructure is compromised. This automatic compilation mechanism reduces development time and makes the ecosystem highly flexible. To address different application requirements, the Zama ecosystem has expanded through four main extensions and libraries: Concrete (the core compilation engine), Concrete-ML (an extension built on top of the main framework to integrate FHE encryption into machine learning models), TFHE-rs (a pure Rust implementation of the TFHE scheme designed to provide high performance and memory safety) and finally fhEVM (an extension focused on the blockchain environment for the execution of private Smart Contracts). [36]

Concrete

The main innovation introduced by Concrete is its ability to significantly reduce the complexity of homomorphic encryption, allowing developers to create privacy-preserving applications without requiring advanced mathematical knowledge [24]. The core of this framework is a cryptographic compiler based on the LLVM infrastructure, which is able to automatically translate

standard high-level code, written using Python APIs, into optimized homomorphic circuits based on the TFHE scheme. From a software point of view, Concrete is able to abstract the underlying mathematical complexity through a simple workflow organized into three main phases:

1. **Defining the computation:** The developer writes the desired algorithmic logic or the desired computation using standard Python code.

2. **Compiling the computation:** the compiler analyzes the code and automatically generates an optimized computational graph, handling the cryptographic parameters and constraints required for homomorphic evaluation.

3. **Executing the computation:** The resulting circuit is executed directly on encrypted data by exploiting, if available, hardware acceleration on GPUs to reduce computation time.

Concrete-ML

It is an important open-source library that brings FHE into the field of Data Science, allowing the creation of Machine Learning models able to operate directly on encrypted data. The main advantage of this library is its accessibility in fact, data scientists can develop privacy-preserving applications from the starting point without requiring advanced cryptographic knowledge. By using familiar and widely adopted machine learning tools such as scikit-learn, a Python machine learning library, and PyTorch, an open-source deep learning framework, Concrete-ML is able to automatically convert traditional models into their homomorphic equivalents. Moreover, it supports data pre-processing and even allows the training or fine-tuning of Large Language Models (LLMs) while keeping the information encrypted. If we focus on the development and implementation of a model in Concrete-ML, the process can be divided into three main phases:

1. **Training the model:** Initially the model is trained on clear data using standard libraries. Since the underlying TFHE scheme operates on integers, Concrete-ML performs a quantization process on the model weights, converting floating-point values into integer representations.

2. **Compiling the model:** At this point, the quantized model is passed to the core compiler, which first translates it into a Concrete Python program and then into an equivalent homomorphic circuit. During this process it is able to automatically determine the cryptographic parameters required for the homomorphic evaluation.

3. **Performing inference:** During deployment, the application uses a client/server architecture in which the client encrypts its data locally, the cloud server performs predictions using FHE without the need of decrypting the inputs and finally the client after having received the encrypted result will decrypt it.

The library also supports Federated Learning scenarios, in which multiple clients collaboratively train a shared machine learning model while keeping their local training data on their own devices. Instead of transferring the raw data to a central server, each client performs the training locally and shares only the resulting model updates. These updates are then aggregated to obtain a global model. This approach can be combined with Concrete-ML to train models in a distributed way and subsequently perform privacy-preserving inference using FHE, avoiding the high computational cost associated with performing the entire training process homomorphically.

Although Concrete-ML is a fundamental library for Confidential Computing applied to artificial intelligence, it also presents some structural limitations. In particular, the quantized representation of models imposes constraints on the supported precision, which can lead to a reduction in accuracy compared to the corresponding model operating on clear data. Moreover, Concrete-ML supports training on encrypted data only for a limited set of models and it leaves to the client the control of the pre-processing and post-processing phases. [37]

TFHE-rs

It is a Rust-based library designed to perform Boolean and integer arithmetic operations on encrypted data. It is based on the Learning With Errors problem and it is considered promising

candidates for post-quantum cryptography. The main feature of TFHE-rs is its ease of use for developers in fact, the library is able to automatically manage all the mathematical parameters required by low level cryptographic operations. In this way, software developers can focus only on the functionality of their applications without the need to become cryptography experts. Moreover, to integrate with different software architectures, TFHE-rs provides a flexible multi-platform ecosystem composed of three main programming interfaces: a native Rust-API, a C-API designed for the integration with legacy systems, and a WebAssembly(WASM)-API on the client side, developed to include encryption routines directly inside web applications.

From a computational point of view, TFHE-rs implements the homomorphic computation workflow through four sequential steps that separate the role of the user on the client side, from the cloud on the server side. The flow can be summarized in the following way:

1. **Key generation:** Thanks to predefined security parameters, the system generates a pair of separate keys. The Client key must remain secret and is used for encryption and decryption operations, while the Server key also referred to as the evaluation key, is provided to the computing infrastructure to perform homomorphic operations.

2. **Encryption:** The client encodes plaintext messages into ciphertexts using its private key.

3. **Homomorphic operation execution:** The server receives the ciphertexts and performs the requested computations by applying the evaluation key. During this phase, to prevent the corruption of data caused by the accumulation of mathematical noise, the library automatically manages circuit stability through Programmable Bootstrapping by periodically restoring the security limits of the data.

4. **Decryption:** The client receives the results processed by the server and by using its Client key can decrypt the ciphertexts in order to recover the computed plaintext results.

Finally, the internal architecture of TFHE-rs includes also a flexible and optimized backend, able of exploiting not only the parallel computing power of GPUs, but also the hardware acceleration by the use of dedicated HPUs (Hardware Processing Units). [38]

fhEVM (Fully Homomorphic Encryption Virtual Machine)

It represents the underlying technology of the confidential blockchain protocol developed by Zama. It is the extension of the Ethereum Virtual Machine (EVM) that introduces the possibility of performing computations directly on encrypted data within smart contracts, which are computer programs that are automatically executed on a blockchain when specific predefined conditions are satisfied. A blockchain is a distributed system that records and shares transactions and data across a network of participants without relying on a central authority. Thanks to fhEVM, developers can create decentralized applications (dApps) able to process transactions and data on a public blockchain without exposing sensitive information, which instead remains encrypted and protected. The architecture and the transmission of encrypted data within fhEVM are based on the cooperation of six key components, both on-chain and off-chain:

- **FHEVM Solidity Library:** allows developers to write confidential smart contracts using the standard Solidity language and introducing native encrypted data types and homomorphic operations. Solidity is a high-level programming language specifically designed for developing smart contracts on blockchain platforms such as Ethereum.

- **Host Contracts:** are trusted contracts deployed directly on blockchains that are compatible with EVM. Their goal is to manage Access Control List (ACL), which define the permissions for accessing specific resources, and to start homomorphic computation requests outside the main blockchain.

- **Coprocessors:** are external decentralized services responsible for the verification of the validity of encrypted inputs. They have to perform the FHE computations and record the results back on the blockchain.

- **Gateway:** acts as the central component of the entire protocol. It validates encrypted input data, manages ACLs, moves ciphertexts between different chains and coordinates the work of coprocessors and the key management service.

- **Key Management Service (KMS):** is a decentralized network based on threshold Multi-Party Computation (MPC). This module generates and rotates global FHE keys and performs secure and verifiable decryptions only when authorized.
- **Relayer:** is a lightweight off-chain service that facilitates the interaction between users and the Gateway by automatically sending encryption or decryption requests coming from the dApp frontend.

Thanks to this complex infrastructure, fhEVM is able to overcome the traditional limitation of transparency public networks, by introducing a new paradigm of scalability and privacy where smart contracts can process sensitive data in a completely secure and decentralized way. [39]

To summarize and have a better understanding of the ecosystem developed by Zama, it can be described as a layered technological stack. Each layer abstracts the complexity of the one below it by progressively moving from low level mathematical concepts to practical applications for end users. In this way, the architecture can be represented through three main levels:

The Base Infrastructure: TFHE-rs (The Cryptographic Engine)

It represents the initial and deepest layer of the stack. At this level we do not handle high-level logic such as Machine Learning or Blockchain protocols, but we work on the pure mathematical management of cryptographic primitives. In particular, it includes:

- Atomic operations on single bits and basic arithmetic operations such as addition, subtraction and multiplication, on encrypted integers.
- Noise management in fact, it directly implements Programmable Bootstrapping algorithms to reset the mathematical noise introduced by the LWE scheme.
- Hardware efficiency is entirely written in Rust to ensure maximum memory safety and supports hardware acceleration through GPU and HPU backends. In this way it acts as a computational engine.

The Intermediate Layer: Concrete (The Compiler)

Due to the fact that programming by directly working with Boolean circuits of TFHE-rs is a complex task and it is easy to deal with errors, Zama introduced Concrete. In this layer we can find:

- Code abstraction in fact, Concrete acts as a software compiler. It is able to take a program or function that is written in high-level Python code and automatically converts it into an optimized computational graph compatible with TFHE-rs primitives.
- Cryptographic automation reduces the need for developers to manually configure the underlying cryptographic parameters. The compiler generates the parameters required by the computation and manages the operations needed to obtain an efficient homomorphic circuit, including the use of bootstrapping when required.

High-Level Application Libraries

They represent the top layer of the stack and are divided into two main solutions:

- Concrete-ML: Dedicated to the Data Science field. It takes predictive models and neural networks developed with traditional frameworks such as Scikit-learn, an open source Python library for classical machine learning algorithms, and PyTorch, a widely used deep learning framework for training flexible neural networks. It performs their quantization and then delegates the compilation process to Concrete in order to transform them into models capable of performing inference directly on encrypted data.
- fhEVM: Dedicated to Web3, the decentralized iteration of the World Wide Web built on blockchain technology and smart contracts, and the broader Blockchain space. It integrates the computational capabilities of Concrete and TFHE-rs inside the Ethereum Virtual Machine by introducing new data types in the Solidity language that are natively protected by encryption.

In conclusion, the architectural logic of the Zama stack allows software project requirements to be mapped onto specific components. In this way, if the main requirement is the computational speed or hardware support, the attention shifts on the performance of TFHE-rs. For example, if the main goal is the simplicity of development, the reference interface is the automation provided by Concrete. Finally, if we analyze from the application point of view, the main solutions are Concrete-ML and fhEVM.

2.4.2 The OpenFHE Library (Open Source Fully Homomorphic Encryption)

In the field of privacy-preserving technologies, OpenFHE represents one of the most robust, flexible and modern software libraries used for the execution of encrypted computations. Originally developed as the successor to the PALISADE library, OpenFHE incorporates contributions from several research projects and institutions and was developed through collaboration between industry and academia, following the security recommendations and standardization efforts of the "HomomorphicEncryption.org" consortium. The library is developed in C++ and is cross-platform, this means that it is compatible with Linux, Windows and macOS through GNU C++ compilers version 9 or Clang C++ version 10 or higher version. It differs from the other ones because it presents a design architecture focused on the maximum usability, meaning that all cryptographic schemes supporting data packing share the same common API and are implemented using runtime polymorphism. In this way we allow developers to abstract the low level complexity of the cryptographic operations.

From an architectural point of view, OpenFHE presents a modular structure organized into four fundamental layers:

- **Primitive Math Layer:** is the lowest layer and is used for managing basic modular arithmetic, Number Theoretic Transforms, and integer sampling. This layer is designed to run on different hardware platforms.
- **Lattice/Polynomial Operations Layer:** is responsible for managing algebraic operations on polynomial rings, lattice operations and lattice trapdoor sampling. In cryptography, a trapdoor is a secret mathematical shortcut that makes extremely hard lattice problems easy to solve. Trapdoor sampling uses this secret shortcut to generate random keys or values from a specific probability distribution, ensuring the system remains secure and impossible for an attacker to break.
- **Crypto Layer:** contains the optimized and concrete implementations of the FHE protocols.
- **Encoding Layer:** is used to handle the different plaintext encoding methods required by the various schemes.

Thanks to this flexible architecture, OpenFHE can be considered a **natively multi-scheme library**. In fact, it integrates efficient variants, often optimized through the RNS (Residue Number System) technique, for all the main homomorphic encryption schemes like BFV, BGV and CKKS.

One of the most innovative characteristics of OpenFHE is its native support for hybrid vector-based schemes, whose goal is to extend the ability to perform functional bootstrapping typical of FHEW/TFHE, also to vector-oriented schemes such as CKKS and BFV. In particular, the library allows performing advanced operations of switching schemes, such as:

- The dynamic switching between CKKS and FHEW/TFHE for the evaluation of non continuous or non regular functions, such as comparison operations, is achieved by applying the scalar functional bootstrapping of FHEW/TFHE.
- The switching between the RLWE scheme and the CKKS scheme enables the evaluation of arbitrary lookup tables on encrypted integer vectors by converting RLWE ciphertexts to CKKS and by using vectorized functional bootstrapping.

Finally, the OpenFHE infrastructure also includes powerful multi-party extensions for homomorphic encryption, which are essential for collaborative and distributed scenarios. Among these features, the main ones are threshold FHE, available for the BGV, BFV, and CKKS schemes,

interactive bootstrapping protocols designed for Threshold CKKS environments and Proxy Re-Encryption mechanisms extended to all major supported schemes (BGV, BFV, and CKKS). These mechanisms allow a ciphertext encrypted with a specific key to be transformed into a ciphertext encrypted with a different key without the need of decrypting the original data. [40]

In our project, we decided to use OpenFHE as the reference library for the implementation of the drone image monitoring and matching system. The choice of using this library was based on several important elements:

- **Native and efficient support for the CKKS scheme with RNS:** Since the Fourier-Mellin spectral algorithm and the transform computations, as the FFT, developed in this project require the processing of real-valued and complex-valued vectors and floating point calculations, the use of the CKKS scheme was fundamental. Although the FFT naturally generates data in complex form, the software architecture was designed to decompose these outputs, separate them and encrypt respectively the real and imaginary components into parallel streams. The RNS implementation in OpenFHE further improves the efficiency of the underlying modular arithmetic while supporting the approximate numerical representation required by CKKS.

- **Cryptographic parameter configuration (Leveled HE):** Unlike high level libraries that hide the configuration of security parameters, OpenFHE allows fine grained control over mathematical noise. In the context of this project, we decided to use a leveled homomorphic encryption approach, this was done by analytically configuring the multiplicative depth (`SetMultiplicativeDepth`) and the scaling modulus size (`SetScalingModSize`). This precise calibration made us possible to optimize the circuits used for squaring operations and spectral distance computation by ensuring stability to the system, but without introducing the high computational overhead typically associated with bootstrapping procedures.

- **Native serialization and distributed cryptographic architecture:** A key factor in selecting OpenFHE was the presence of a highly efficient native serialization module based on the Cereal libraries. In the developed code, ciphertexts and evaluation keys are directly serialized in binary format (`SerType::BINARY`) and are encapsulated into network transmittable strings through Base64 encoding. This made possible to have a clear separation of roles following a Confidential Computing approach in which the client module generates the keys and keeps the secret key private, while it sends to the Cloud server only the encrypted data and the multiplication evaluation key (`export_mult_key`). In this way, the remote server can perform homomorphic multiplication operations (`EvalMult`) without accessing and having a knowledge of the original information.

- **Architectural robustness, binding and modern code design:** The development of the project was performed through a hybrid software architecture in which we combined C++ with Python programming languages. The central core for the use of cryptography was developed using the native C++ backend of OpenFHE and ensuring, at the same time, the absence of memory leaks and optimized execution times. The high level application logic and image processing operations performed through OpenCV, were instead implemented in Python. We were able to implement this dual approach by developing a custom binding system using the pybind11 library and by creating a native extension (`ckks_context`) that exposes C++ classes directly to the Python interpreter. This solution allows us to maintain the efficiency of the cryptographic engine while significantly accelerating the application prototyping and debugging phases.

2.5 FHE Compilers

A compiler is a program that translates source code written in a programming language such as C, C++ or Java into machine code. This process converts high level instructions into an executable that can be directly understood and so used by the processor of the computer being used. In our work of thesis, FHE compilers are high level software tools designed to abstract the complex APIs provided by the underlying cryptographic libraries. The introduction of a compilation layer is fundamental to allow developers, who are not necessarily cryptography experts, to implement

privacy-preserving mechanisms in a secure and optimal way. FHE compilers are used because, thanks to them, we are able to automatically handle some of the most challenging aspects of the developing homomorphic applications like:

- **The selection of cryptographic parameters for the chosen FHE scheme:** This is done in order to guarantee both post-quantum security and computational efficiency. In fact, to achieve this, compilers usually present modules that automatically generate the required parameters based on predefined application requirements.

- **Plaintext Encoding:** In homomorphic encryption, the representation of plaintext data depends on the type of computation that needs to be performed. Here compilers are very useful due to the fact that, many of them help developers to correctly map native software data types to the corresponding cryptographic structures.

- **Data-Independent Execution:** By design, FHE operations cannot depend on the actual value of encrypted data and so this makes traditional conditional branches unsuitable because they would compromise the confidentiality of data. In order to overcome this limitation, compilers will evaluate both branches of the computation in parallel and will select the correct result at the end of the execution by using homomorphic logical operations.

- **Packing and Batching:** FHE schemes that support packing multiple messages into a single ciphertext can take advantage of SIMD computation. In fact, FHE compilers optimize performance by automatically vectorizing execution circuits, allowing a single homomorphic operation to be performed on entire data vectors simultaneously.

- **Ciphertext Maintenance:** We know that managing and limiting the mathematical noise generated during homomorphic operations is one of the greatest challenges in FHE development. To solve this problem, compilers implement advanced static and dynamic optimization strategies to preserve the integrity of encrypted data.

Today, there are many FHE compilers, including both general purpose solutions and domain specific that are the ones designed for Machine Learning or other specialized fields. Focusing on main general purpose compilers, as shown in Table. 2.2, where in the Status column "A" stands for *Active*; "L" stands for *Legacy* and "D" stands for *Discontinued*; we can mention:

Compiler	Language	Library						Year/Status
		HElib	SEAL	PALISADE	FHEW	TFHE	HEAAN	
ALCHEMY [41]	Haskell	○	○	○	○	○	○	2018 - L
Cingulata [43]	C++	○	○	○	○	●	○	2020 - L
E^3 [45]	C++	●	●	●	●	●	○	2022 - L
SHEEP [46]	C++	●	●	●	○	●	○	2019 - L
EVA [47]	C++/Python	○	●	○	○	○	○	2021 - L
Marble [49]	C++	●	●	○	○	○	○	2018 - L
RAMPARTS [53]	Julia	○	○	●	○	○	○	2019 - L
Transpiler [54]	C++	○	○	●	○	●	○	2022 - L
CHET [55]	C++	○	●	○	○	○	●	2019 - L
nGraph-HE [56]	C++	○	●	○	○	○	○	2021 - D
SEALion [58]	C++	○	●	○	○	○	○	2019 - L
Sunscreen [59]	Rust	○	●	○	○	○	○	2024 - A
HEIR [60]	C++/Python	○	○	○	○	●	○	2026 - A
Fhelipe [61]	C++/Python	○	○	○	○	○	○	2024 - A
CHEHAB [63]	C++/Python/Rust	○	●	○	○	○	○	2026 - A
Oraqle [65]	Python/C++	●	○	○	○	○	○	2024 - L

Table 2.2. Comparison of FHE compilers and supported libraries (information refers to the status of the projects as of September 2026).

ALCHEMY

It is a Haskell-based compiler developed for Fully Homomorphic Encryption. It is built around embedded Domain-Specific Languages (eDSLs) and uses a typed tagless-final approach, which represents encrypted computations through typed interfaces and allows Haskell's type system to detect certain programming errors at compile time. This enables ALCHEMY to define specialized languages for describing encrypted computations while providing compile-time verification of their structural correctness. It implements the BGV encryption scheme using the $\Lambda \circ \lambda$ (L O L) library for lattice based cryptography. The compiler translates computations written for plaintexts into equivalent computations on ciphertexts and automatically manages several aspects of FHE evaluation, such as ciphertext parameters, key switching operations, and noise. This reduces the amount of FHE specific knowledge required from the programmer and simplifies the development of homomorphic circuits. [42] [41]

Cingulata

It was developed in C++, it is both a tool chain and a runtime environment used for Fully Homomorphic Encryption. It can compile C++ programs into Boolean circuits that can be evaluated using TFHE, while also providing support for arithmetic computations through its B/FV implementation. CINGULATA uses mathematical libraries such as FLINT and SageMath and applies heuristic optimization techniques to improve the efficiency of the generated circuits. In particular, it aims to reduce the multiplicative depth of Boolean circuits and automatically parallelize parts of the computation. This can improve the efficiency of homomorphic evaluation, since lower multiplicative depth can reduce the computational cost of FHE operations. CINGULATA also provides a parallel runtime environment that can exploit the parallelism present in the generated circuits. Overall, it makes it easier to run C++ programs on encrypted data while reducing some of the complexity associated with FHE. [44] [43]

E^3 (Encrypt-Everything-Everywhere)

It is a C++ based compiler mainly designed to work with programs that use encrypted data. One of its main features is that it is not related to a specific FHE library. This means that the same C++ code can be used with different libraries and encryption schemes such as Microsoft SEAL, HElib, TFHE, FHEW, and PALISADE. Not only, E3 provides a simple layer between the program and the underlying FHE libraries, hiding many of their technical details. It also allows developers to use custom encrypted data types and standard C++ operators. Another important feature is *bridging*, which allows arithmetic and Boolean computations to be combined within the same application. This makes it possible to use modular arithmetic, supported by schemes such as BFV and CKKS, together with bit-level Boolean operations supported by schemes such as TFHE. Moreover, E^3 natively supports SIMD batching and it is able to manage operations such as relinearization and rescaling. Overall, E3 makes it easier to develop C++ programs that work with encrypted data and to switch between different FHE libraries. [45]

SHEEP

It was developed in C++, it was born for the evaluation and benchmarking of HE systems. It is not designed as a traditional FHE compiler, but as a tool for testing and comparing different HE libraries and schemes. SHEEP uses a library independent language, in this way the computation we want to perform can be described only one time and then it can be evaluated using different FHE backends. Moreover, the platform provides predefined benchmarks and allows users to test their own programs. During the execution of tests, SHEEP is able to collects information about encryption, decryption, and evaluation times, of ciphertext size, but also it is able to checks whether the results are correct. [46]

EVA (Encrypted Vector Arithmetic)

It is a compiler developed by Microsoft for Fully Homomorphic Encryption. Its main library is written in C++ and it also provides official Python bindings, for example PyEVA. EVA provides an intermediate representation for describing encrypted vector computations and compiles these

programs into homomorphic operations supported by Microsoft SEAL. EVA focuses especially on the CKKS scheme, which is designed for computations on encrypted approximate real and complex numbers. One of its main goals is to hide much of the cryptographic complexity from the programmer by automatically selecting encryption parameters and inserting operations such as rescaling and relinearization when needed. It also applies several optimizations to improve the efficiency of homomorphic computations and provides support for parallel execution. EVA can also be used as an intermediate representation for higher level of FHE compilers, making it possible to build other specialized tools on top of it. [48] [47]

Marble

It is a C++ compiler and high-level FHE framework designed to simplify the development and optimization of applications based on Fully Homomorphic Encryption. Introduced in 2018, Marble aims to hide much of the cryptographic complexity from the programmer by automatically handling several aspects of FHE configuration, including the selection of the encryption scheme, plaintext encoding, and cryptographic parameters. It provides a familiar C++ programming environment and translates high-level computations into operations that can be executed on encrypted data. In its original implementation, Marble supports the BGV scheme through HElib and the BFV scheme through Microsoft SEAL, selecting the most appropriate backend according to the requirements of the computation. This allows developers to focus on the application logic rather than on the low-level details of FHE configuration and circuit construction [50] [49]. The same research line was later extended with **HECO**, a separate and more recent end-to-end FHE compiler developed by members of the same research community. Unlike Marble, which primarily focuses on simplifying the configuration and development of FHE applications, HECO focuses on automatically transforming high-level imperative programs into efficient implementations following the SIMD-oriented computational model required by FHE. HECO is built on the MLIR (Multi-Level Intermediate Representation) compiler infrastructure to progressively lower the input program from a high-level representation to FHE-specific operations and finally to scheme-specific operations. It supports Ring-LWE-based schemes such as BFV, BGV, and CKKS and targets Microsoft SEAL as its underlying FHE implementation. Therefore, HECO can be considered a subsequent development of the ideas introduced with Marble, extending them from a high-level FHE development framework toward a more general end-to-end optimizing compiler. However, the official HECO repository was archived in September 2026, and the project should therefore be considered a legacy compiler rather than an actively maintained solution. [51] [52]

RAMPARTS

It is a Julia-based system designed to make easier the development of Homomorphic Encryption applications. Julia is a high-level, high-performance programming language particularly suited to scientific and numerical computing, making it adequate for expressing mathematical computations involved in HE applications. It performs symbolic execution to analyze Julia programs and automatically generates arithmetic circuits for homomorphic evaluation. RAMPARTS can optimize the generated circuits and automatically select important FHE parameters and plaintext encodings, reducing the amount of cryptographic knowledge required from the programmer. It uses the PALISADE library as its FHE backend. Overall, RAMPARTS provides a more programmer-friendly way to develop and optimize homomorphic computations while abstracting several low-level details of the underlying FHE implementation. [53]

Transpiler

Developed by Google in C++ for FHE, it is built on the Google XLS (Accelerated HW Synthesis) compiler infrastructure. This hardware-oriented compilation approach makes the Transpiler particularly suitable for exploring efficient FHE implementations on hardware accelerators. As the name suggests, it translates high-level programs into optimized logic circuits that are compatible with the TFHE scheme. Google XLS is an open-source framework and toolchain developed by Google for High-Level Synthesis (HLS), which enables hardware circuits to be generated from high-level software descriptions such as C++ or XLS's own domain-specific language. The original implementation uses the TFHE library as its FHE backend and focuses mainly on Boolean

and bit-level computations. By using XLS, the transpiler can optimize the generated circuits and transform control structures such as loops into logic graphs suitable for efficient parallel execution. Another important feature is its modular design, which allows to use different input and output languages and FHE libraries. Overall, Transpiler aims to simplify the transformation of existing programs into FHE compatible versions while hiding many of the low level details of homomorphic evaluation. [54]

CHET

Developed by Microsoft for neural network inference on encrypted data, CHET is a domain specific compiler written in C++. It translates tensor programs into optimized homomorphic circuits and focuses mainly on the CKKS scheme, supporting both HEAAN and Microsoft SEAL libraries. One of its main features is the automatic optimization of tensor layouts, which is important for making efficient use of batching and vector operations in FHE. CHET also automatically selects the required encryption parameters and rotation keys based on the target computation. These optimizations reduce the amount of low level FHE knowledge required from the programmer and can improve the performance of neural network inference. The work on CHET was later followed by EVA, which provides a more general compiler framework for homomorphic computations. [55]

nGraph-HE

nGraph-HE is an Intel extension to the nGraph deep learning compiler that enables neural network inference on homomorphically encrypted data. It works as an HE backend for nGraph and uses Microsoft SEAL to implement the CKKS encryption scheme. It allows models developed with deep learning frameworks such as TensorFlow to be executed on encrypted data while applying HE-specific optimizations during compilation. These optimizations include techniques such as constant folding and SIMD packing, as well as optimized ciphertext operations, which help improve the efficiency of homomorphic evaluation. The original version of nGraph-HE was limited to neural networks using polynomial activation functions. To support the non-polynomial activation functions commonly used in deep learning, the subsequent nGraph-HE2 introduced a client-aided approach based on a two-party computation model, where the server temporarily sends the required non-linear computations to the client, which evaluates them and returns the re-encrypted result. Developed primarily as a research proof-of-concept, nGraph-HE is no longer maintained by Intel. Its official repository was archived in 2024, and the project is therefore considered discontinued. [57] [56]

SEALion

It is an open-source framework for privacy-preserving machine learning using Homomorphic Encryption. It presents an architecture based on two layers where the first one is built on TensorFlow and Microsoft SEAL and provides standard algebraic and deep learning operations, while the second ones offers a simpler Keras-like syntax for training and inference. This allows users to develop neural networks used for prediction on encrypted data. SEALion hides most of the low level details related to data encoding and encryption, making it easier to prototype privacy-preserving models without requiring extensive knowledge of Homomorphic Encryption. It also includes advanced optimizations, such as activity sparsifiers, which are used to improve the efficiency of neural networks, as well as support for remote loading through Variational Autoencoders (VAEs) operating on encrypted data. Overall, SEALion was designed to make the development and testing of privacy-preserving neural networks easier. [58]

Sunscreen

It is a Rust-based open-source ecosystem for developing privacy-preserving applications using Fully Homomorphic Encryption and, more recently, zero-knowledge proofs. The original Sunscreen compiler provides a high-level interface for writing FHE programs in Rust and uses Microsoft SEAL with the BFV scheme as its underlying FHE backend. The use of Rust provides strong type and memory safety guarantees, which can be particularly useful when developing cryptographic software. The project also provides a web-based Playground that allows users to experiment with FHE programs without installing the complete framework locally. However,

the original compiler is explicitly intended for experimentation, has not been externally audited, and is therefore not intended for production use. More recently, the Sunscreen ecosystem has evolved toward a new Secure Processing Framework (SPF), which is based on a variant of the Torus Fully Homomorphic Encryption (TFHE) scheme. SPF introduces the Parasol processing architecture, which enables arbitrary computations to be executed over encrypted data. Its architecture includes dedicated components for ciphertext operations, instruction scheduling, and execution, with the Parasol processor providing the underlying execution model. This represents a significant shift from the original BFV-based compiler and Microsoft SEAL backend toward a TFHE-based architecture specifically designed for executing computations directly on encrypted data. The SPF implementation is also explicitly described as unaudited, and therefore it should be considered experimental rather than production-ready. [59]

HEIR (Homomorphic Encryption Intermediate Representation)

It is a modular compiler tool chain for Fully Homomorphic Encryption based on the MLIR infrastructure. Its main goal is to provide a common compilation framework that can work with different FHE schemes, libraries and hardware platforms, reducing the dependency on a single backend. Moreover, HEIR can take high-level programs written in languages such as Python, including programs developed using frameworks such as PyTorch. The same can be done for different backends including OpenFHE, Lattigo, TFHE-rs, and Jaxite, depending on the supported scheme. It also includes advanced optimizations for algebraic operations and RNS polynomial arithmetic, in this way it generates optimized code not only for CPUs but also for dedicated hardware accelerators such as GPUs, FPGAs and ASICs. However, HEIR is still an actively developing ecosystem, so some parts are less developed than established FHE libraries such as Microsoft SEAL. In addition, the use of a general compiler framework may involve trade-offs between portability across different backends and the fine-grained control available when targeting a specific FHE library or hardware platform. [60]

Fhelipe

Fhelipe is a tensor compiler for Fully Homomorphic Encryption, mainly designed for tensor computations and applications such as neural network inference. It is based on the Go-based Lattigo library as its cryptographic backend, even if this is not shown in the Table 2.2. One of its main contributions is automatic data packing: the compiler automatically chooses how multidimensional tensors should be used and packed into ciphertexts in order to reduce the cost of homomorphic operations such as rotations and multiplications. This allows programmers to use a simple, tensor based programming interface instead of manually handling low level FHE operations. Fhelipe can reduce the amount of source code needed by around 10× to 48× compared with low level implementations, while its evaluations show that it can match or even outperform carefully hand optimized FHE applications. This makes Fhelipe suitable for applications such as logistic regression, neural networks, FFT and other tensor operations. However, Fhelipe is mainly focused on tensor computations rather than general purpose FHE programs, so its programming model is primarily suited to arithmetic tensor computations rather than applications based on Boolean logic or general-purpose logic circuits. Another limitation is its development setup, which requires several dependencies when installed manually, although the project provides a Docker setup to simplify the installation. [62] [61]

CHEHAB

It is a domain-specific compiler for FHE, designed to work with the Microsoft SEAL library and to support the BFV encryption scheme. It introduces a domain specific language embedded in C++, that allows developers to describe homomorphic computations in a simple way while delegating circuit management to the compiler. A key feature is an optimization engine built on a Term Rewriting System (TRS) and e-graphs. The TRS applies algebraic identity rules to simplify equations, while the e-graph efficiently stores and tracks thousands of mathematically equivalent program variations simultaneously. This allows the compiler to explore a large search space of mathematically equivalent program variations and select transformations that reduce both the number of multiplications and the circuit depth. To achieve this, the compiler uses

artificial intelligence techniques to determine the best transformations for the cryptographic graph that is the one able to reduce the mathematical noise of ciphertexts. CHEHAB also includes a reinforcement learning (RL) optimizer that can learn how to apply transformations based on the cost of the resulting FHE program. These optimizations aim to reduce the number and cost of FHE operations, as well as the circuit and multiplicative depth, which are important for both execution time and noise growth. Moreover, CHEHAB applies more traditional optimizations such as Constant Folding (CF), which pre-calculates constant expressions during compilation, and Common Subexpression Elimination (CSE), which eliminates redundant computations by reusing previously computed values. A limitation of CHEHAB is that it currently focuses on BFV and Microsoft SEAL, making it less flexible than multi-scheme compiler frameworks such as HEIR. Since BFV is designed for exact integer arithmetic, CHEHAB is more naturally suited to integer based computations, while applications that require approximate real or complex arithmetic, such as many machine learning workloads, are better suited to schemes such as CKKS. [64] [63]

Oraqle

It is a compiler designed to translate high level Python code into arithmetic circuits for HE. In this way, Oraqle makes HElib easier to use. One of its main features is depth aware arithmetization, which converts high level operations into arithmetic ones while considering both the number of multiplications and the multiplicative depth of the resulting circuit. Instead of generating only one circuit, Oraqle can produce multiple versions with different trade-offs between these two costs. Additionally, Oraqle uses semantic Common Subexpression Elimination (CSE), an advanced optimization that detects mathematically equivalent expressions even if they are written differently, preventing redundant FHE calculations and streamlining the final circuit. Finally, for the homomorphic evaluation, Oraqle uses HElib and relies on the BGV scheme, while fhegen is used for parameter selection. Since it is strictly related to HElib backend, the compiler lacks multi-library flexibility. Therefore, although Oraqle is particularly effective for building complex mathematical and algebraic circuits where reducing multiplicative depth and the number of multiplications is important, it is less flexible when applications require support for different FHE libraries. On the other hand, it It is therefore less suited to machine learning workflows than specialized frameworks such as Concrete-ML. [66][65]

All of these tools aim to simplify the development of FHE applications by translating high level programs into optimized homomorphic computations. Although they differ in their programming languages, supported schemes, backends, and target applications, they all try to hide part of the low level complexity of FHE and make encrypted computation easier to develop and optimize.

2.6 FHE Accelerators

Today, hardware acceleration for FHE plays a key role in overcoming the high computational overhead of fully homomorphic encryption. Although the evolution of software tools has significantly reduced the execution time of FHE schemes, the performance gap compared to plaintext computation is still one of the main engineering challenges that limits the large scale adoption of FHE. This problem becomes even more evident in complex applications such as privacy-preserving Machine Learning and neural network inference, where computations can involve large circuits and many sequential operations.

One of the main source responsible of this overhead is bootstrapping. In fact, as homomorphic computations continue, noise builds up in the ciphertext and may need to be reduced through a bootstrapping operation. However, bootstrapping is computationally expensive and can become a bottleneck in FHE applications. As a result, a computation that takes only a few seconds in plaintext may require much more time when performed homomorphically on a standard CPU. To address this problem, researchers and industry have developed dedicated hardware accelerators specifically designed for FHE. These accelerators take advantage of the high level of data and task parallelism found in FHE workloads and are designed to reduce both computation time and memory movement. The main hardware acceleration are:

GPU-based accelerators (Graphics Processing Unit)

Originally designed for graphics processing and massive parallel computation, GPUs have undergone significant technological evolution to satisfy the computational requirements of cryptography. Modern GPU architectures present dedicated computing units that can efficiently support modular arithmetic and Number Theoretic Transforms. The latter represents a fundamental algorithmic component of FHE, as it allows polynomial multiplication to be performed more efficiently by transforming the polynomials into a domain where the multiplication can be performed element by element. By taking advantage of their high memory bandwidth and massive parallelism, GPUs have become a flexible and powerful solution for managing large scale FHE workloads in Cloud infrastructures, like during AI model training. A concrete example is TensorFHE, which exploits the Tensor Core Units of an NVIDIA A100 GPU to accelerate NTT and homomorphic multiplication through extensive operation level parallelism. The reported results show that TensorFHE can achieve 913 KOPS (Kilo Operations Per Seconds) for NTT and 88 KOPS for homomorphic multiplication on an NVIDIA A100, demonstrating the potential of general purpose GPU architectures for high throughput FHE workloads [67]. The main drawback is their higher energy consumption and heat generation compared to ASIC based solutions.

FPGA-based accelerators (Field-Programmable Gate Array)

They represent one of the first hardware acceleration solutions for homomorphic encryption implemented on flexible and reprogrammable platforms. Early prototypes focused on optimizing operations such as the Chinese Remainder Theorem for the second generation of FHE schemes. These prototypes achieved performance improvements compared to CPU based software implementations. After that, hardware research moved towards the Number Theoretic Transform, which is one of the most computationally expensive operations in many ring-based lattice cryptographic schemes. The development of dedicated NTT coprocessors and specific offloading architectures allowed FPGA based platforms to improve the performance of traditional software libraries. Another step was achieved with the introduction of flexible hardware designs optimized for the CKKS scheme. These architectures enabled efficient processing of encrypted vectors and achieved higher performance than highly optimized software libraries such as Microsoft SEAL. Finally, the accessibility of these technologies was extended to modern infrastructure models by demonstrating the feasibility of deploying FPGA accelerators directly in Cloud environments to support real time industrial applications.

ASIC-based accelerators (Application-Specific Integrated Circuit)

ASICs are designed to overcome some of the main limitations of FPGA based accelerators. The main advantage of FPGAs is that they can be reprogrammed for different FHE schemes and workloads. However, FPGA resources are not as specialized as those with a dedicated circuit, which can lead to higher latency, greater energy consumption, and less efficient use of hardware resources for specific workloads. In addition, moving data between memory and processing units can become an important bottleneck, especially in FHE workloads that require large amounts of data movement. To overcome these limitations we have introduced ASICs circuits, specifically designed for the desired FHE operations. They allow the hardware to reduce unnecessary data movement and execute the most expensive operations of FHE, more efficiently. Although ASICs are less flexible than FPGAs, their specialized design can provide higher performance and better energy efficiency for specific FHE workloads. Among the main ASIC-based approaches, we can find:

- **Hybrid systems for neural networks:** These architectures combine homomorphic encryption with complex circuits bringing to a reduction of latency during neural network inference and further improvements in terms of efficiency, up to approximately 80 times compared to the first prototypes.

- **Wide-vector programmable co-processors:** These represent flexible and programmable ASIC accelerators based on wide vector processors and managed through static scheduling strategies. By minimizing data movement, this architecture achieves significant speed improvements and provides native support for multiple cryptographic schemes such as BGV,

HEAAN and GSW. This can be achieved by accelerating basic FHE operations such as modular arithmetic, NTT, and data permutations rather than being tied to a single scheme. This shows that FHE hardware can achieve very high performance while still remaining programmable and flexible.

In addition to the hardware accelerators described above, another important research direction is the development of processors and architectures specifically designed around the computational characteristics of FHE. These systems are often referred to as **Homomorphic Processors or Homo-Processors**, of which we will discuss in the following section.

Today, there are many different types of hardware platforms used for computation, such as traditional CPU processors, GPU graphics cards, and special FPGA or ASIC chips. For this reason, FHE software needs an abstraction layer that allows the same cryptographic operations to be executed using different underlying implementations. Without such a layer, developers would need to directly interact with the specific implementation used by each platform. This is one of the reasons why OpenFHE library becomes very important. OpenFHE includes a software layer called **HAL (Hardware Abstraction Layer)**, which provides an abstraction between the higher level FHE operations and the underlying mathematical backends. HAL is not a physical hardware component and does not represent a hardware accelerator. Instead, it allows OpenFHE to switch between different implementations of the mathematical operations used by the library. In this way, the higher level FHE code remain mostly independent from the specific backend used to perform the underlying computations. The OpenFHE documentation describes HAL as a mechanism for switching between different math backends. [68]

A practical example of this architecture is the native integration of the **Intel HEXL library inside OpenFHE**, which optimizes lattice arithmetic operations directly on compatible processors through the HAL interface. Thanks to this approach, advanced hardware manufacturers can develop their own acceleration modules and connect them directly to the standard HAL specifications. This architectural design provides two main advantages for the ecosystem; on one hand, it allows the exploitation of the highest levels of computational parallelism offered by hardware platforms, on the other hand, it reduces the need for software developers to have advanced cryptographic or hardware design expertise. [68]

However, hardware acceleration is only one part of the effort to make FHE a reliable and widely adopted technology. Standardization has also played an important role in the development of the FHE ecosystem. With this goal, in 2017 some companies such as Microsoft, IBM and Duality Technologies founded the "international consortium HomomorphicEncryption.org". This organization actively works to define standardized and mathematically certified security parameters. This work has accelerated both the development of ready-to-use solutions created by specialized companies such as Zama AI, Duality and Cryptolab, and the adoption of FHE by large technology companies within commercial privacy-oriented architectures. Some of the most important examples are the use of privacy-preserving ML with Intel SGX (Software Guard Extensions) and Google's Password Checkup system. SGX is a security technology that creates protected areas in the processor's memory where sensitive data and computations can be processed securely. Google's Password Checkup, on the other hand, uses Private Set Intersection (PSI) to check whether a user's password has been exposed without revealing the actual password to Google. In this way, the system can check the security of user credentials while keeping sensitive information private.

2.6.1 The Evolution Towards Homo-Processors and New Homomorphic Computing Platforms

The structural limitations found in traditional digital architectures such as CPUs and GPUs, including limited parallelism, high memory traffic, and high energy consumption for FHE operations, have pushed research and industry towards the development of a new class of unconventional hardware: **Homo-Processors**. A Homo-Processor is a computing architecture specifically designed to execute mathematical operations directly on encrypted data in FHE. Unlike general-purpose processors, which execute FHE operations through software and are not specifically optimized for the large polynomial structures involved, Homo-Processors integrate specialized hardware units, memory hierarchies, and data paths designed to accelerate homomorphic operations directly on ciphertexts. This approach focuses on reducing data movement and accelerating the

most expensive operations of homomorphic computation. The main architectural characteristics include:

Memory Hierarchy

In FHE, a single plaintext value can be represented by a ciphertext that is much larger than the original data, often reaching sizes of KB or even MB depending on the encryption scheme and parameters. Traditional cache memories are not designed to store and move such large amounts of data. As a result, data movement between the processor and main memory can become an important bottleneck. Homo-Processor is able to solve this issue by introducing an innovative memory hierarchy with very large on-chip registers, high bandwidth, and optimized data paths. This allows large cryptographic polynomials and ciphertexts to be processed while reducing the amount of data transferred between the processor and the external memory.

Native bootstrapping acceleration

Bootstrapping is one of the most computationally expensive operations in many FHE schemes and can become a major bottleneck when performed on general purpose processors. To be performed it requires billions of clock cycles when executed through software, making it extremely slow. Homo-Processor, thanks to the integration of dedicated hardware circuits specifically designed for this operation, is able to transform the computation into a single high speed hardware instruction. An important development in this area was the introduction of Gate Bootstrapping and Functional Bootstrapping. Gate Bootstrapping is mainly used to refresh a ciphertext after a Boolean operation, such as AND or XOR, so that it can be used again for further computations. Functional Bootstrapping goes a step further; while refreshing the ciphertext, it can also apply a function to the encrypted value. In simple terms, Gate Bootstrapping mainly refreshes the encrypted data, while Functional Bootstrapping can refresh it and perform a computation at the same time. This makes Functional Bootstrapping more flexible and useful for more complex FHE applications. These techniques have greatly reduced the time required for bootstrapping compared with early approaches. For example, while the bootstrapping operation in Gentry's original scheme could take up to approximately 30 minutes, later algorithmic and implementation improvements have reduced bootstrapping time by several orders of magnitude. In fact, with Gate and Functional Bootstrapping, execution times are on the order of milliseconds for specific workloads and parameter settings. [69]

Polynomial arithmetic units (NTT Units)

The algebraic engine of FHE is based on complex multiplications between high degree polynomials, which are computationally expensive when performed directly. NTT addresses this problem by converting polynomial multiplication into element wise multiplication in the transform domain, significantly reducing its computational complexity. Homo-Processors integrate hundreds of parallel NTT hardware units, allowing multiple polynomials and independent transform operations to be processed simultaneously. This massive parallelism reduces the latency of polynomial arithmetic and accelerates fundamental FHE operations such as ciphertext multiplication, key switching, and relinearization.

The development of these specialized architectures has led to several research and commercial directions for homomorphic computing. In this context, FPGA and ASIC should not be considered as new alternatives to the accelerators discussed in the previous section. Instead, they can be used as implementation technologies for specialized Homo-Processors. For example, **FPGA based Homo-Processors**, to maintain a certain level of reprogrammability, while an **ASIC based Homo-Processors** implementation can provide a more specialized design with higher efficiency for a specific FHE workload.

Other approaches explore completely different computing technologies, such as photonic computing and in-network processing:

Photonic Accelerators (Optical Computing)

This platform represents the post digital boundary and introduces the revolutionary concept of Computing in Transit that is promoted by companies such as Optalysys with the Etile accelerator.

Since the most computationally expensive mathematical operations, such as NTT and inverse NTT, can be modeled through light diffraction, in this case the computation is performed when the optical signal travels through the system at the speed of light. This approach allow to reduce computational latency from microseconds to nanoseconds, to eliminate the thermal and electrical delays associated with transistors and to significantly reduce energy consumption per bit. However, it is an emerging technology and so it presents challenges including integration with existing digital systems, precision, calibration, and the management of physical noise, without affecting the integrity of the homomorphic computation. [70]

DPU and SmartNIC (In-Network Computing)

Another emerging direction is the use of Data Processing Units (DPUs) and SmartNICs to move part of the computation closer to the network interface. Unlike traditional Network Interface Cards (NICs), which mainly handle communication between the server and the network, SmartNICs and DPUs include their own processing resources, memory, and hardware acceleration capabilities. This can reduce the amount of data that needs to be transferred between the network interface, the memory of the system, and the CPU. In a traditional NIC, FHE computations are mainly performed by the CPU, meaning that encrypted data received from the network must first be transferred to the host system for processing. SmartNICs introduce additional programmable or dedicated processing capabilities, allowing selected networking and computational tasks to be performed without involving the main CPU for every operation. DPUs represent a further evolution of this approach by integrating networking, programmable processing cores, dedicated memory, and hardware accelerators in a single device. Platforms such as the NVIDIA BlueField series can execute computational workloads independently of the host CPU and provide a high degree of programmability. For FHE workloads, this architecture could be useful for reducing data movement and offloading selected cryptographic or homomorphic operations from the main processor. However, current DPU platforms such as NVIDIA BlueField are not specifically designed as FHE processors and do not natively provide general-purpose FHE acceleration. Their use for FHE should therefore be considered an emerging research direction, where future systems could integrate FHE-specific operations, such as polynomial arithmetic and NTTs, closer to the network interface.

Overall, the evolution towards Homo-Processors represents a shift from simply accelerating individual FHE operations towards designing complete computing architectures around the requirements of homomorphic computation. FPGA-based implementations offer flexibility and reprogrammability, while ASIC-based designs can provide higher specialization and efficiency. At the same time, emerging approaches such as photonic computing and in-network processing explore new ways of reducing the computational and data movement costs of FHE.

2.7 Classification of FHE Schemes

As already described in the previous sections, the evolution of Fully Homomorphic Encryption from a purely theoretical concept to an applicable and implementable technology has been driven by the development of different families of cryptographic schemes, each one designed and optimized for specific computation.

As reported in the literature [12], the evolution of schemes based on the complexity of lattice problems, such as the Ring Learning With Errors (RLWE) problem, has led to the development of three main families of solutions. These families are classified according to the algebraic domain of the plaintext and the methods used to manage the noise. In particular, we can find:

- **Schemes for Exact Integer Arithmetic (BGV / BFV):** The Brakerski - Gentry - Vaikuntanathan (BGV) and Brakerski-Fan-Vercauteren (BFV) schemes were born for working on vectors of integer values inside a finite field. In these architectures, the noise introduced by encryption is separated from the plaintext message. The original value is maintained after every computation, as long as the global noise level remains below a critical threshold, which can be controlled through the bootstrapping process when required.

- **Schemes for Boolean Circuits and Look-Up Tables (TFHE / DM):** These schemes are optimal for the evaluation of binary logic circuits such as AND, XOR and NOT gates.

We can also find schemes as Torus FHE (TFHE) that is important since it is able to perform a very fast bootstrapping operation after each individual logical gate. This approach completely removes noise accumulation, but at the cost of a computational granularity mainly focused on Boolean logic and non-linear evaluations that are implemented through the use of look-up tables.

- **Schemes for Approximate Arithmetic on Real Numbers (CKKS):** This represents the third and most recent class of solutions, they are introduced with the aim of overcoming the limitations caused by data quantization in scientific and statistical computing scenarios.

The choice of the cryptographic scheme is therefore not a simple technical detail, but a decision that affects the overall data architecture and logical workflow of the application. In fact, when computational workloads involve continuous signals, biometric data, statistical functions, or floating-point matrices, such as those used in spectral vector processing or machine learning model inference, integer-based schemes such as BGV and BFV require data quantization. This can introduce additional overhead and approximation issues that may be problematic in real-time or resource-constrained embedded scenarios. For these reasons, CKKS is often preferred for such applications because it natively supports approximate arithmetic on real and complex numbers. By treating the noise introduced by encrypted computation similarly to numerical approximation errors in floating-point calculations, CKKS enables the processing of continuous-valued signals and vectors directly in the encrypted domain. This reduces the need for complex quantization procedures and makes CKKS particularly suitable for privacy-preserving scientific and machine learning applications.

2.7.1 The CKKS Scheme

As we know, before CKKS introduction, the FHE ecosystem was mainly based on discrete schemes designed for exact computations over modular integers. These schemes introduced significant computational costs and limited accuracy when applied to continuous computational domains. [71]

Mathematical structure and vector encoding

The architecture of the CKKS scheme exploits a batching technique based on the complex canonical embedding map. This optimization technique allows an entire data vector to be packed inside a single ciphertext instead of encrypting each individual number in a separate way. As a result of that, parallel computations can be performed on all elements of the vector using a single cryptographic operation that brings a significantly reduction of the execution times.

Through this isometric homomorphism, a dense plaintext vector belonging to $\mathbb{C}^{N/2}$ is mapped into the ring of cyclotomic polynomials. This geometric transformation preserves the norm of the elements, preventing the expansion and amplification of numerical errors during the encoding phase. In this way the process allows the data vector to be condensed into a single plaintext polynomial, divided into independent parallel channels called slots. As a consequence of that, every homomorphic operation performed on the ciphertext acts simultaneously to all slots according to the SIMD paradigm that is the one able to significantly reduce the linear computational overhead.

Cryptographic noise and error management in numerical computations

The main intuition behind the use and implementation of the CKKS scheme was the overcoming of the traditional concept of cryptographic noise in systems that are based on the Ring Learning With Errors (RLWE) problem. While in traditional protocols, such as BFV or BGV, noise represents a destructive component separated from the message, CKKS integrates noise by redefining it as part of the approximation error that is naturally present in numerical computations.

Since digital computers introduce a small error of rounding when operating with floating-point formats, the CKKS scheme allows cryptographic noise to overlap with the least significant digits of the plaintext. While discrete schemes require complex and expensive quantization process to convert decimal values into modular integers, by introducing non-linear discretization errors, CKKS is able to process continuous data while preserving its original nature. Moreover, to avoid

ciphertext expansion during multiplication operations, it is introduced the rescaling procedure. This operation consists of removing the least significant bits (LSB) affected by noise and reducing the ciphertext modulus through a scalar division by a predefined factor. In this way, the error is recalibrated while preserving the most significant bits (MSB) containing the actual information, allowing the decryption of the data with a predefined precision level.

Native homomorphic operations in the encrypted domain

If we move towards the encrypted domain, we know that CKKS scheme natively supports a complete set of algebraic primitives on ciphertext slots, following a homomorphic logic that works in the same way as the one of vector operations performed on plaintext data. In fact, if we consider two generic ciphertexts c_1 and c_2, which encrypt the complex message vectors m_1 and m_2 respectively under the same secret key, the four fundamental operations supported by the cryptographic architecture are:

- **Homomorphic Addition (Add):** The addition of two ciphertexts is a simple algebraic sum performed component by component between the coefficients of the corresponding polynomials. This primitive generates a new ciphertext that will contain the sum of the original elements slot-by-slot $m_1 + m_2$. Due to the fact that it is a linear operation, the growth of cryptographic noise can be considered negligible by making it a computationally inexpensive operation.

- **Homomorphic Multiplication (Mul):** This operation allows two generic ciphertexts to be multiplied, we will obtain a resulting ciphertext that represents the product of the vector elements slot-by-slot $m_1 \times m_2$. This operation performs the tensor product of the polynomials, which linearly increases the ciphertext dimension from two to three elements and exponentially increases both the plaintext scale and the noise level. As a consequence of what is performed, we need to do a relinearization phase in order to restore the ciphertext to its standard dimension, together with a rescaling operation.

- **Squaring Operation (Square):** This represents an optimized and specialized version of homomorphic multiplication, where a ciphertext is multiplied by itself to calculate the square of the original messages m_1^2. From a mathematical point of view, the square operation uses dedicated algorithms at both the algebraic and software levels, as implemented in the OpenFHE APIs. Since the tensor product decomposition is performed on a single identical element, this operation reduces computation time and lowers the overhead compared to a multiplication between two different ciphertexts.

- **Homomorphic Rotation (Rotate):** This is the fundamental primitive for the structural manipulation of encrypted vectors. It allows the cyclic shift of the elements contained inside the ciphertext slots by k positions without the need of decrypting the data or modifying its content. The operation is based on **Galois automorphisms**, which are algebraic transformations of the underlying cyclotomic polynomial ring that preserve its structure while permuting the encoded slots. This operation is essential in scientific computations to perform algorithmic shifts, permutations, or to implement scalar products and complex matrix computations directly in the Cloud environment.

Suitability of the CKKS scheme in the drone pipeline

The mathematical properties described above, together with the flexibility of the scheme, make CKKS the only suitable choice for the requirements and the goal we want to achieve in our project. In fact, the images that are captured at the beginning by the drone, undergo a phase of preparation directly during flight that we can define as embedded pre-processing. This step is performed through the theoretical application of the Discrete Fourier Transform (DFT) and the practical implementation of the Fast Fourier Transform (FFT), which takes the image pixels and converts them into a frequency domain representation. The output of this computation is a list of high precision decimal and complex numbers that represent the actual spectral footprint of the image. After that, through the commands provided by the OpenFHE library, which offers an effective software abstraction layer, this spectral footprint is encoded. All decimal values are

inserted into parallel slots contained inside a single ciphertext before being transmitted through the network.

But we know that, even if CKKS scheme theoretically supports all four operations previously described, in the software developed for this thesis we will use only the native Homomorphic Addition and Squaring operations. This design choice is done because it allows the Cloud server, that is considered in our model as an external and untrusted infrastructure, to evaluate complex comparison and vector matching algorithms. In fact, the server can execute a single operation that acts simultaneously and in parallel on all the data that are present inside the vector. In this way we are implementing SIMD homomorphic computation. By doing this, our project not only will take benefits from the higher computational efficiency of the squaring operation compared to a standard multiplication, but also completely avoids the need to convert decimal values into integers. This allows us to maintain high precision during real-time detections, without ever exposing the original images or the spectral data extracted from the drone to third parties.

In the next chapters, we will present and describe in detail the **Edge-to-Cloud system pipeline architecture** developed in this project of thesis. In particular, we will analyze the development environment, the configuration of security parameters inside the OpenFHE CryptoContext and the structure of the C++ code. We will also discuss the software binding integration which is essential to connect the different system modules. Finally, we will show how the complete architecture enables efficient homomorphic encryption of vectors on the embedded module and the subsequent homomorphic matching and comparison on the Cloud server between the generated spectral footprints.

Chapter 3

Signal Processing

3.1 Limitations of Spatial Domain Matching and Algorithmic Incompatibility

Implementing a video tracking or image recognition system for drones requires taking into account several aspects. One of the main challenges is the visual instability caused by the flight. When the drone camera captures a target, the scene can continuously change as the drone moves, making the target appear in different positions and from different perspectives. Variations in altitude can also change the distance from the target, affecting how it can appear in the image. These changes can make the tracking process more difficult, especially when the system has to follow the same object over time. In real scenarios, the software system must therefore be able to compensate, in real time, for these continuous variations in scale, position and angle in order to avoid losing track of the target of interest.

In order to see if the picture captured by the drone presents a specific target, a simple approach would be to compare the captured image directly with a reference image of the target, pixel by pixel. However, this method has several practical limitations. In particular, applying traditional metrics such as **squared difference** or **cross-correlation** may cause the system to fail even when there is only a small change between the images. The squared difference measures the difference between pixels values of the two images at the same positions, while cross-correlation measures how well pixels values in one image match those in the other. In this context, the image comparison is performed by looking at the pixels in the two images and checking how similar their values are at the same positions. However, a small change due to a little movement, rotation, or change in the target's size may change the position or the value of pixels in the captured image. Therefore these pixels may no longer correspond to the same pixels in the reference image, where the target is stored in its original position. As a result, the similarity between the two images can decrease, even though the target is still present in the captured scene. This can lead to an incorrect result and, in particular, to a false negative.

To overcome this problem, classical computer vision methods do not directly compare the entire pixel matrix of the image. Instead, they use local feature extraction algorithms such as **SIFT (Scale-Invariant Feature Transform)** or **ORB (Oriented FAST and Rotated BRIEF)**. These methods aim to search for stable and distinctive points in the image, called **keypoints**. The keypoints can then be used to describe and compare the same target in different images, even when its position, scale, or orientation changes.

In particular, the **SIFT** algorithm [72] is designed to find keypoints that are robust to changes in scale, rotation, and illumination. To handle changes in scale, SIFT analyzes the same image at different levels of detail. To do this, it progressively applies a Gaussian blur to the image. Gaussian blur is an image processing operation that smooths the image by reducing small details and noise. As the amount of blur increases, small details gradually disappear, while larger structures remain visible. By comparing these different versions of the image, SIFT can identify points that remain distinctive at different scales. The blurred images are organized into groups called octaves, Containing several versions of the image with increasing levels of blur. After one octave is completed, the image is reduced in size and the same process is repeated to create the next octave. In this way, SIFT can search for keypoints at different image sizes and levels of detail. This allows

the algorithm to detect the same relevant points even when an object appears larger or smaller in different images. To handle rotation, SIFT analyzes how the image intensity changes around each keypoint. These changes are called intensity gradients and indicate the direction in which the image becomes brighter or darker. From these gradients, SIFT determines a main direction, called the dominant orientation, for each keypoint. The descriptor is then created with respect to this orientation, so that the same keypoint can still be recognized even if the object is rotated. Finally, SIFT describes each keypoint using the local intensity patterns around it, creating a distinctive numerical descriptor that can be compared with descriptors from another image. In this way, the same target can still be recognized even when its position, scale, or orientation changes. So, if on one hand the SIFT algorithm represents the standard solution for visual accuracy, on the other hand the **ORB** algorithm [73] is developed as a computationally efficient alternative, specifically designed for real time scenarios or environments with limited hardware resources. This algorithm works by combining two different techniques:it uses the **FAST (Features from Accelerated Segment Test)** algorithm for the rapid detection of keypoints and the **BRIEF (Binary Robust Independent Elementary Features)** descriptor for their representation. FAST detects keypoints by examining the pixels around a candidate point. In particular, it checks whether several neighboring pixels are significantly brighter or darker than the candidate point. If this condition is satisfied, the point is considered sufficiently distinctive and can be selected as a keypoint. BRIEF then describes each keypoint by comparing the intensity of pairs of pixels in the area around it. For each pair, the result of the comparison is represented by a binary value, either 0 or 1. By repeating this comparison for many pairs of pixels, BRIEF creates a binary string that represents the local appearance of the keypoint. This representation is compact and can be quickly compared with descriptors from another image using simple binary operations. However, the original BRIEF descriptor is sensitive to image rotation. To overcome this limitation, ORB calculates the dominant orientation of each keypoint by analyzing how pixel intensities are distributed around it. The BRIEF descriptor is then adjusted according to this orientation. As a result, ORB can recognize similar keypoints even when the object is rotated in the image. Another important advantage of ORB is its binary representation. Unlike SIFT, which uses floating-point values to describe image features, ORB represents each descriptor as a sequence of 0 and 1. These binary descriptors can be compared using simple and fast operations, reducing the computational cost. For this reason, ORB is particularly suitable for real time applications and systems with limited hardware resources.

However, both approaches become impractical when the computations have to be performed directly on encrypted data using FHE. As discussed previously, the CKKS scheme imposes some mathematical constraints on the operations that can be efficiently performed on encrypted data. In particular, CKKS mainly supports additions and multiplications on encrypted vectors, with multiple values processed in parallel across the available slots. As a result, the non linear operations required by algorithms such as SIFT and ORB become an important obstacle.

In the proposed Zero-Trust architecture, the remote Cloud server is considered untrusted and must therefore process the data without learning their content and without having access to the secret key required for decryption. However, the same restriction does not apply to the UAV itself. The drone is considered a trusted device and can therefore process its locally acquired data in plaintext before sending any information to the Cloud. This distinction is important when considering traditional computer vision algorithms such as SIFT and ORB. These algorithms are composed of several elementary processing steps that are simple when executed on plaintext images, but are not directly supported by the CKKS scheme. In particular, they may require operations such as:

- **Comparisons between numerical values:** for example, determining whether the intensity of one pixel is greater or smaller than another value.
- **Threshold operations:** deciding whether a value is above or below a predefined threshold in order to accept, reject, or classify a candidate point.
- **Maximum or minimum searches:** for example, selecting the strongest response among several neighboring pixels or identifying a dominant value.
- **Conditional operations:** executing different instructions depending on the result of a comparison, such as if-else statements.

These operations are used often inside feature extraction algorithms. For example, SIFT needs to compare candidate points with their neighbors in order to determine whether they represent local extrema, while ORB and FAST compare pixel intensities to decide whether a candidate point should be considered a keypoint.

CKKS, instead, is mainly designed to perform approximate arithmetic operations on encrypted numerical values, especially additions and multiplications. Direct comparisons, threshold decisions, and conditional branching are not native CKKS operations. Sometimes they can be approximated using more complex polynomial computations, but this would increase the computational cost, the multiplicative depth, and consequently the complexity of the homomorphic computation. For the proposed architecture, reproducing traditional feature extraction algorithms such as SIFT or ORB directly in the encrypted domain would be both computationally expensive and unnecessary. In fact, these algorithms are discussed only as examples of conventional computer vision approaches, while they are not used in the implementation of the proposed system. The main objective of the architecture is not to keep the data encrypted while they are processed inside the UAV, since the drone is considered a trusted device. Instead, confidentiality becomes necessary when data is transmitted to and processed by the remote Cloud, which is considered an honest-but-curious component. For this reason, the computational workload is divided between the UAV and the Cloud according to their different trust levels. The UAV performs the image processing operations locally and in plaintext. In particular, both the reference image and the cuptured one are processed using the FFT and the Fourier-Mellin transform, in order to obtain compact frequency domain feature representations. The UAV then computes the element-wise difference between the feature vector extracted from the current image and the reference feature vector. Once this difference vector has been obtained, it is encrypted using the CKKS scheme and transmitted to the Cloud. Therefore, the remote server never receives the original images, the plaintext feature vectors, or the plaintext differences.

At this stage, the Cloud performs only a simple arithmetic operation that is naturally supported by CKKS: the element-wise squaring of the encrypted difference vector through homomorphic multiplication. These squared values represent the individual contributions to the squared Euclidean distance between the two feature representations. The encrypted result is then returned to the UAV, where it is decrypted and used to compute the final error or similarity measure. The drone can then compare this value with the selected tolerance threshold and make the final recognition decision. In this way, complex image processing operations are kept inside the trusted UAV, while only the part of the matching computation that can be efficiently expressed through homomorphic arithmetic is outsourced to the untrusted Cloud. This design avoids the need to reproduce complex computer vision logic over encrypted data while still ensuring that the information processed by the remote infrastructure remains protected.

3.2 Moving to the Frequency Domain: The Discrete Fourier Transform (DFT)

Traditional image analysis methods that rely on the spatial arrangement of pixels are incompatible with homomorphic encryption schemes like CKKS. To perform addition and multiplication operations supported by FHE, we must shift our perspective. In fact, instead of seeing the pixel matrix (image) as a static grid of coordinates, we consider it as a discrete two-dimensional signal. The key element of this approach is the **Discrete Fourier Transform (DFT)**, which converts the signal from the spatial domain into its frequency spectrum. As known from signal processing theory, the DFT is the mathematical tool used to compute the frequency spectrum of a discrete signal obtained by sampling a continuous function.

To fully understand the DFT algorithm, let is start by considering one-dimensional signal composed of a sequence of N complex numbers x_n, where $n = 0, \ldots, N-1$. The DFT associates this sequence with another sequence of N complex numbers X_k, defined by the following fundamental equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn} [74]$$

for every frequency index $k = 0, \ldots, N-1$, where j denotes the imaginary unit. When this definition is extended to the two-dimensional domain of images, represented as pixel matrices of

size $M \times N$, the 2D-DFT is obtained by applying the transformation sequentially along both the rows and the columns of the matrix:

$$X_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} e^{-j2\pi\left(\frac{km}{M} + \frac{ln}{N}\right)}$$

By computing the magnitude of each resulting complex coefficient, $|X_k|$ (or $|X_{k,l}|$ in 2D), we obtain the magnitude (or amplitude) spectrum, while its argument defines the phase spectrum. The main property of the DFT when focusing on encrypted computation, is the behavior of the magnitude spectrum under spatial translations, which is determined by the linearity and shift properties of the transform. Specifically, if an image is translated by a spatial offset (x_0, y_0), its Fourier transform undergoes only a phase variation, appearing as a linear phase shift in the complex exponential. Since the absolute value of this complex phase factor is equal to one, the magnitude spectrum remains completely unchanged. This shift invariance solves the target positioning problem because, from a theoretical standpoint, if the target moves within the image frame, the energy distribution of its spectral magnitudes stays the same.

However, computing the DFT on large matrices requires a number of arithmetic operations that grows quadratically with the signal size, leading to a computational complexity proportional to a order of $O(N^2)$ operations in 1D, or $O(M^2N^2)$ in the two-dimensional case. Therefore, to ensure the process to be performed in real time on the onboard hardware, it is necessary to use the **Fast Fourier Transform (FFT)**. FFT does not define a different mathematical transformation from the DFT; rather, it represents an optimized class of algorithms specifically designed to reduce the number of required arithmetic operations. Although similar mathematical ideas were explored much earlier by pioneers such as Carl Friedrich Gauss, the modern algorithm relies on the divide-and-conquer principle formalized by James Cooley and John Tukey in 1965. This method exploits the intrinsic symmetry and periodicity properties of complex trigonometric coefficients to reduce computational complexity. For a one-dimensional sequence of length N, assuming N is a power of two, the DFT summation can be recursively split into two independent subproblems of size $N/2$: one corresponding to the even-indexed samples ($n = 2m$) and the other to the odd-indexed samples ($n = 2m + 1$). This leads to the following fundamental relationship:

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-j\frac{2\pi}{N/2}km} + e^{-j\frac{2\pi}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-j\frac{2\pi}{N/2}km} [75]$$

If we introduce the complex rotation factor defined as $W_N^k = e^{-j\frac{2\pi}{N}k}$, and denote A_k and B_k as the $N/2$-point DFTs associated with the even and odd samples respectively, the equation simplifies into the following linear combination:

$$X_k = A_k + W_N^k B_k$$

Thanks to the periodicity of the partial transforms and the property $W_N^{k+N/2} = -W_N^k$, the values corresponding to the second half of the frequency spectrum ($k \geq N/2$) can be obtained by reusing the same intermediate computations with a simple sign change:

$$X_{k+\frac{N}{2}} = A_k - W_N^k B_k$$

This pair of equations combining A_k and $W_N^k B_k$ through addition for the first half of the spectrum and subtraction for the second one, forms the core of the butterfly operation. The name derived from the shape of its signal flow graph, where the crossing data paths between inputs and outputs resemble a butterfly's wings. Reusing these intermediate results completely removes redundant steps. In fact, it reduces the overall computational complexity from the quadratic order $O(N^2)$ down to $O(N \log_2 N)$. For large datasets, this algorithmic efficiency cuts processing time by several orders of magnitude, turning otherwise intractable computations into fast, manageable tasks.

In the two-dimensional case, for an image of size $M \times N$, the FFT can be computed efficiently by exploiting the separability of the Fourier Transform. Instead of processing the entire image at once, the 1D-FFT is first applied to each row and then to each column of the resulting matrix. This reduces the computational complexity to $O(MN \log_2(MN))$, making the frequency domain

transformation suitable for devices with limited computational resources, such as the onboard module of a drone. In the proposed architecture, the FFT and the following Fourier-Mellin processing are performed locally on the UAV before encryption. This allows the drone to extract a compact frequency domain representation of the image while the data is still in plaintext and the device is considered trusted. As a result, the Cloud does not need to analyze the original pixel matrix or perform complex image processing operations. Instead, it receives an encrypted numerical vector that can be processed using simple arithmetic operations supported by CKKS. In this way, the more complex image processing is performed locally, while only the simpler matching computation is outsourced to the remote Cloud.

3.2.1 The Fourier-Mellin Transform for Scale and Rotation Invariance

Although the FFT removes the effects of translations, it is not enough to guarantee a robust detection of the target. In fact, the magnitude of the FFT is invariant only to translations, while it is not able to compensate for rotations or scale changes. This is a critical issue for our system because the Cloud compares the encrypted feature values element by element, assuming that corresponding information is located in the same positions. However, after a rotation or a change in scale, the spectral components may move to different positions in the matrix. As a result, the Cloud may compare values that no longer represent the same part of the target. Therefore, even if the observed object is actually the correct target, the two feature representations may appear different and produce a large matching error. This can lead to an incorrect recognition result.

To overcome this limitation, we can introduce the **Fourier–Mellin Transform (FMT)**, a mathematical tool originally proposed for image registration. Its main advantage is that it can transform changes in rotation and scale into a form that is easier to handle mathematically. The idea starts by representing the image in polar coordinates, where each point is described by a radius r and an angle θ. In this representation, a rotation of the image simply produces a shift along the angular coordinate θ. Similarly, if the radius is expressed in logarithmic form, a scale change becomes a shift along the radial coordinate. Therefore, rotation and scale variations, which are difficult to compare directly in Cartesian coordinates, can be represented as translations in the polar or log-polar domain. For a function expressed in polar coordinates as $f(r, \theta)$, the Fourier–Mellin Transform can be defined as

$$M_f(u, v) = \frac{1}{2\pi} \int_0^{\infty} \int_0^{2\pi} f(r, \theta)\, r^{-ju} e^{-jv\theta}\, d\theta\, \frac{dr}{r} \text{[76]}$$

The term $e^{-jv\theta}$ performs a Fourier analysis along the angular direction, while the term r^{-ju} performs the corresponding analysis along the radial scale direction. In this way, the transform describes the image changes with respect to both rotation and scale. Now consider two images that contain the same target but differ by a scale factor α and a rotation angle β, so that

$$f_1(r, \theta) = f_2(\alpha r, \theta + \beta).$$

By substituting this expression into the Fourier–Mellin integral and applying a change of variables for the radius and the angle, this leads to:

$$M_{f_1}(u, v) = \alpha^{ju} e^{jv\beta} M_{f_2}(u, v),$$

up to the sign convention adopted in the transform definition. The important point is that the factors α^{ju} and $e^{jv\beta}$ affect only the phase of the transformed signal. Since their magnitude is equal to one:

$$|M_{f_1}(u, v)| = |M_{f_2}(u, v)|.$$

This means that, ideally, the magnitude of the Fourier–Mellin representation remains unchanged even when the target is rotated or appears at a different scale. For this reason, the Fourier–Mellin Transform provides a useful representation for image matching when the same object may appear with different orientations or sizes. The same idea can also be understood from the classical translation, rotation, and scaling properties of the Fourier Transform:

1. If two images differ by a spatial translation and a rotation of angle θ, the translation changes only the phase of the Fourier Transform, while the magnitude spectrum is rotated by the same angle θ.

2. If two images differ by a scale factor α, their Fourier transforms undergo an inverse scaling of the frequency coordinates, apart from an amplitude factor. Therefore, when the object becomes larger in the spatial domain, its frequency spectrum becomes correspondingly compressed, and vice versa.

These properties explain why the Fourier domain is useful, but the magnitude spectrum alone is not sufficient. Taking the magnitude removes the effect of spatial translations, but rotations and scale changes are still present in the spectrum.

For this reason, the practical solution adopted in our implementation is therefore based on the **Log-Polar Transform**. A point (x, y), expressed with respect to the center of the image (x_c, y_c), is re-mapped into log-polar coordinates (ρ, θ) according to:

$$\rho = \log\left(\sqrt{(x - x_c)^2 + (y - y_c)^2}\right) [77]$$

$$\theta = \text{atan2}(y - y_c, x - x_c)[77]$$

where ρ represents the logarithm of the radial distance from the image centre, while θ represents the angular position of the point with respect to the horizontal axis. The main advantage of this representation is that rotation and scale changes become simple translations. A rotation of the target produces a shift along the angular coordinate θ, while a change in scale produces a shift along the logarithmic radial coordinate ρ. In particular, the logarithmic representation transforms multiplicative scale changes into an additive shifts.

This is also the connection between the log-polar representation and the Mellin Transform. By introducing the substitution $r = e^{\rho}$, the radial term of the Mellin Transform can be rewritten using the logarithmic coordinate ρ. In this form, it has the same exponential structure used by the Fourier Transform. Therefore, the Mellin analysis of scale variations can be implemented in a discrete image processing system by applying a Fourier Transform to the log-polar representation.

In practice, the magnitude spectrum of the image is first obtained through a Fourier Transform and is then converted into log-polar coordinates. At this point, the effects of rotation and scale appear as translations along the θ and ρ axes. A further Fourier Transform is applied to this log-polar representation. Since translations affect the phase of a Fourier Transform but not its magnitude, considering the magnitude of this result reduces the effect of the remaining rotation and scale shifts.

The final result is a spectral representation that is more robust to changes in target position, orientation, and size. This is particularly useful in the proposed architecture because these image processing operations can be performed locally on the trusted UAV before the encryption stage.

Once the spectral representations of the reference target and the current image have been obtained, they can be used to build the numerical data required by the matching procedure. The information that must be remotely processed is then encrypted using CKKS and transmitted to the Cloud. In this way, the remote server only needs to perform simple arithmetic operations on encrypted numerical data and does not need to apply geometric transformations or conditional image processing operations directly on ciphertexts.

3.2.2 Practical Fourier–Mellin Processing Pipeline

In this section, we will describe how we have implemented the Fourier-Mellin process that is used in both images, the captured one and the target one. After loading and resizing the input image we split it into three channels: blue, green, and red (RGB). Then we apply the following steps (see Fig. 3.1) to each channel through a "for" cycle.

- `First FFT (translation invariance)`: the first step consists on converting the channel data from the spatial domain to the frequency domain. This is performed by `f = np.fft.fft2(channel)`, which computes the two-dimensional DFT using the FFT algorithm. The resulting spectrum is then rearranged through `fshift = np.fft.fftshift(f)`, moving the zero-frequency component to the center. The instruction that retains only the

```
1     # A. First FFT (translation invariance)
2     f = np.fft.fft2(channel)
3     fshift = np.fft.fftshift(f)
4     magnitude = np.abs(fshift)
5
6     # B. Log-polar mapping (scale/rotation -> translation)
7     log_polar = cv2.logPolar(
8         magnitude, center, M,
9         cv2.WARP_FILL_OUTLIERS + cv2.INTER_LINEAR
10    )
11
12    # C. Second FFT in the log-polar domain
13    f_mellin = np.fft.fft2(log_polar)
14    f_mellin_shift = np.fft.fftshift(f_mellin)
15    final_spectrum = np.abs(f_mellin_shift)
16
17    # D. Normalize between 0 and 1 for CKKS
18    normal_spectrum = cv2.normalize(
19        final_spectrum, None, 0, 1,
20        cv2.NORM_MINMAX)
```

Figure 3.1. Practical Fourier–Mellin processing code.

magnitude of the Fourier coefficients is `magnitude = np.abs(fshift)` . This operation removes the phase information, which is affected by translations of the target, while the magnitude spectrum remains unchanged.

- `Log-polar mapping (scale/rotation -> translation)`: the resulting magnitude spectrum is then mapped from Cartesian to log-polar coordinates using the conversion expressed as `log_polar = cv2.logPolar(magnitude, center, M, ...)`. This transformation represents each point according to its distance and angle from the center of the spectrum of the image. In this way, a rotation of the target becomes a shift along the angular direction. This makes rotation easier to be processed in the following steps.

- `Second FFT in the log-polar domain`: a second two-dimensional FFT is then applied to the log-polar representation through `f_mellin = np.fft.fft2(log_polar)`. As in the previous step, the spectrum is centered using `f_mellin_shift = np.fft.fftshift(f_mellin)` and only its magnitude is retained through `final_spectrum = np.abs(f_mellin_shift)`. Since rotations and scale variations have been converted into translations in the log-polar domain, these translations affect the phase of the second Fourier Transform while leaving its magnitude unchanged. The resulting `spettro_finale` therefore provides a spectral representation that is less sensitive to variations in target position, rotation, and scale.

- `Normalize between 0 and 1 for CKKS`: finally, the spectrum is normalized by using this instruction `cv2.normalize(final_spectrum, None, 0, 1, cv2.NORM_MINMAX)`, obtaining values within the range [0,1] and providing a common numerical scale for the three channels. Each normalized matrix is then converted into a one-dimensional vector through `spettro_norm.flatten().tolist()` and stored in `results_rgb[c]`. At the end of the loop, the function returns the three Fourier-Mellin vectors corresponding to the red, green, and blue channels, which are used in the subsequent matching and encryption stages.

3.3 Extraction of the Invariant Spectral Footprint and Homomorphic Matching

The Fourier–Mellin processing described in the previous section produces a spectral representation of the observed target that is more robust to changes in position, rotation, and scale. The output of this processing can be interpreted as a numerical footprint of the image. Instead of comparing the original two-dimensional pixel matrices, the system works with a fixed size one-dimensional vector composed of floating point values.

This representation is optimal for the proposed architecture because every processed image is converted into the same numerical structure, which can then be directly encoded using the CKKS scheme. The use of a fixed size global descriptor `ckks.init()` is also important for the cryptographic point of view. Traditional methods based on keypoints, such as SIFT or ORB, may generate a variable number of descriptors depending on the image content and often require comparisons, thresholding operations, and conditional decisions. These operations are difficult to reproduce efficiently on CKKS ciphertexts.

In the proposed architecture, the spectral footprints of the reference target and the current image are computed locally on the UAV, which is considered a trusted device. The UAV then calculates the element-wise difference between the two vectors:

$$d_i = f_{\text{current},i} - f_{\text{target},i}$$

This operation is performed before encryption. The resulting difference vector is then encrypted using CKKS and transmitted to the remote Cloud. Therefore, the Cloud never receives the original images, the plaintext spectral footprints, or the plaintext differences. The Cloud only performs the arithmetic operation required for the outsourced part of the matching procedure. In particular, it homomorphically multiplies the encrypted difference vector by itself, obtaining the squared value of each component:

$$d_i^2 = \left(f_{\text{current},i} - f_{\text{target},i}\right)^2$$

This operation can be directly performed with CKKS homomorphic multiplication. In this way, the Cloud processes the encrypted numerical differences without accessing their plaintext content. The resulting encrypted vector is then returned to the trusted UAV, where it is decrypted and used to obtain the final value. This is then compared with a predefined tolerance threshold in order to determine whether the target is identified or not. The main advantage of this approach is that the complex image processing stage is completed before the encryption step, while the untrusted Cloud only performs a simple polynomial computation on encrypted numerical data. In this way, the original image matching problem is reduced to a sequence of arithmetic operations that can be efficiently combined with the CKKS homomorphic encryption scheme.

3.4 Dimensional Optimization and Adaptation to CKKS Scheme Constraints

As already discussed, working with homomorphic encryption requires a trade off between the amount of visual information that can be processed and the computational sustainability of the operations involved. In traditional computer vision, an image or its magnitude spectrum usually requires a high resolution in order to preserve every small geometric detail. However, in our project, converting and encrypting a high resolution Fourier-Mellin spectrum would represent an unsuitable choice of design due to the fact that the computational cost would become excessively high. In fact, CKKS ciphertexts are dense and complex mathematical structures based on high degree polynomials, and if we use excessively large raw data matrices inside them, this may lead to a significant increase in the computational overhead and may slow down the response time of the Cloud server, making real time tracking impossible.

To overcome this structural limit, the system architecture introduces a specific optimization and dimensionality reduction technique before the encryption phase. In fact, before transferring the data into the cryptographic context, the algorithm does not use the entire two-dimensional spectrum, but it selects and extracts only a concentrated geometric portion, removing the highest

or redundant frequencies that do not affect to much the matching accuracy. This targeted sampling allows the original matrix to be compressed into a much more compact and lightweight one-dimensional vector. This dimensional reduction is related to a precise mathematical constraint implemented in the OpenFHE library that is the SIMD packing mechanism.

In fact, if we focus on the CKKS scheme, this used to arrange the message space into a fixed number of protected memory positions, the encryption slots. In this way, by reducing the size of the target footprint, the entire vector can be placed inside the slots of a single ciphertext, allowing the exploitation of the logical parallelism of the scheme. This geometric optimization is fundamental during the comparison phase performed by the Cloud server. During the homomorphic matching as described in the previous section, SIMD packing allows the element-wise squaring operation to be applied simultaneously to all packed components. So, the Cloud server does not need to perform expensive iterative loops or distribute the computation across multiple independent ciphertexts. This approach contributes to reduce the processing latency required for real time drone operation, while maintaining the cryptographic protection of the exchanged data.

Chapter 4

Drone-Cloud Implementation and System Design

4.1 Overall System Architecture

The implementation of the proposed system is based on a distributed Drone-Cloud architecture composed of two main software components: the **Client**, representing the UAV, and the **Server**, representing the remote Cloud infrastructure. The purpose of this separation is to distribute the processing tasks according to both the computational requirements of the system and the level of trust assigned to each component. The UAV is considered a trusted device and can therefore access and process the acquired information in plaintext. The remote Cloud, instead, is treated as an external computational resource that must operate without accessing the original data or the secret decryption key, so it operates under blind conditions. In our architecture:

- **Client** is responsible for acquiring the image and performing the local processing required to obtain the numerical representation used for target matching. As described in Chapter 3, the image-processing stage is based on the Fourier-Mellin approach and is completed locally before encryption. The UAV also prepares the numerical data required for the remote computation, protects them using the CKKS scheme, and manages the cryptographic material required by the system. The secret key always remains on the trusted side of the architecture.
- **Server** receives only encrypted numerical data together with the evaluation material required to perform the authorized homomorphic operation. During the matching phase, the Cloud processes the encrypted difference vector without decrypting it and returns an encrypted result to the Client. Therefore, the server acts only as a computational component and does not have access to the plaintext image, the numerical representation of the target, or the final recognition decision that is managed by the drone.

This separation of responsibilities is the main characteristic of the proposed architecture. Operations that require access to plaintext data are kept inside the trusted UAV, while the computation selected for remote execution is expressed through arithmetic operations that can be directly evaluated on CKKS ciphertexts. In this way, the Cloud can contribute to the matching process without becoming a trusted holder of the processed information.

To better understand the role of homomorphic encryption in the proposed architecture, we considered three possible approaches for protecting the information exchanged between the UAV and the remote Cloud. These scenarios mainly differ in the level of confidentiality they provide during transmission and remote processing.

Scheme without encryption

This configuration represents the starting point of our analysis and is used as a baseline for the comparison with the following security scenarios. In this case, the numerical information extracted by the UAV is transmitted to the Cloud in plaintext through the HTTP protocol,

without applying any cryptographic protection. Therefore represents the simplest and fastest solution from a computational point of view since no cryptographic algorithms need to be executed. For this reason, this scenario provides a useful reference for evaluating the additional computation time, processing overhead, and communication cost introduced by the protection mechanisms adopted in the different configurations. However, it does not provide confidentiality mainly due to:

- **In-transit vulnerability:** Since data travel in plaintext through the Internet network, an attacker able to intercept the communication channel could directly access the exchanged numerical information. Furthermore, the absence of protection exposes the system to man-in-the-middle attacks in which an attacker could intercept the packet and modify the numerical values during transmission.
- **Destination vulnerability and lack of trust in the Cloud:** assuming that the transmission channel is secure, once the data reach the Cloud server, they remain available in plaintext and can therefore be accessed by an attacker able to compromise it. In scenarios involving sensitive information, this represents a significant limitation because confidentiality would depend entirely on the trustworthiness of the external server.

Scheme with channel encryption - TLS (Transport Layer Security)

The second scenario introduces TLS [78], one of the most widely used solutions for protecting communications between clients and remote servers. In our case, the drone communicates with the Cloud through HTTPS, where TLS protects the information exchanged over the network. Compared with the previous plaintext scenario, this approach introduces a standard security layer and provides three main properties:

- **Authentication:** ensures that the identity of the communicating entity can be verified, reducing the risk of connecting to an unauthorized or malicious server. In our scenario, this allows the drone to verify the identity of the Cloud server through its digital certificate.
- **Confidentiality:** ensures that the information exchanged between the communicating parties cannot be read by unauthorized entities. The transmitted data are encrypted so that an attacker intercepting the communication cannot directly access their content.
- **Integrity:** ensures that the transmitted information has not been modified or altered during communication. TLS therefore allows the receiving entity to detect unauthorized changes to the exchanged data.

If we focus on the internal operations, the protocol works through two separate logical layers that are activated in a sequential way:

1. **The TLS Handshake Protocol:** This is the initial phase that establishes if the connection is secure or not. During this phase, the client starts the exchange with a *ClientHello*, while the server replies with a *ServerHello* and provides its digital certificate for authentication. In TLS 1.3, which is the latest version, key exchange is performed through the ECDHE algorithm (Elliptic Curve Diffie-Hellman Ephemeral). This mechanism allows the drone and the Cloud server to generate temporary and single use keys and to agree on a shared secret without ever transmitting it in plain text over the network. The use of this type of keys guarantees forward secrecy, meaning that even if an attacker compromises the server's private key, it will be not able to decrypt previous communication sessions.
2. **The TLS Record Protocol:** once the handshake has been successfully concluded and the identities have been verified, the asymmetric encryption process is stopped. At this point, the Record Protocol takes control of the data flow and it applies symmetric encryption using a shared secret key, which compared to the asymmetric one, is faster and more efficient. This layer is responsible of taking the image footprint generated by the drone, converting it into ciphertext using the previously negotiated session keys and encapsulating it for secure transmission to the Cloud.

This scenario, even if it represent an improvement, its protection is mainly limited to data in transit. Once the encrypted TLS packet reaches the destination, the Cloud server must recover the plaintext information in order to process it with a conventional application. This is the main limitation of TLS. Although an external attacker cannot directly read the data while they are travelling through the network, the information becomes accessible again inside the remote server during computation. Therefore, if the Cloud infrastructure is compromised or inspected by a privileged entity, the processed data may still be exposed. For this reason, TLS improves communication security but does not completely address the confidentiality requirement considered in this work, where the Cloud itself is not assumed to be fully trusted. This limitation motivates the third scenario, based on CKKS homomorphic encryption, where the objective is to keep the numerical information protected not only during transmission but also while it is being processed by the remote server.

Scheme with homomorphic encryption - CKKS

The third scenario represents the final solution implemented in this work. With this approach, the philosophy behind data protection changes radically. In fact, data protection is no longer limited to the communication channel. As described in Subsection 2.7.1, CKKS allows arithmetic operations to be performed directly on encrypted data. In our architecture, this property is used to keep sensitive information protected even while it is processed by the remote Cloud.

This approach introduces three main advantages:

- **Transmission inviolability regardless of the communication channel:** Since the data leaves the drone as a ciphertext, the security of the transmission process and of the final target information depends no more on the reliability of the communication channel. In this case, even if the connection could be compromised and intercepted by an attacker, the packets are arranged into a random numerical sequences without any logical meaning. So without having the private key, it is mathematically impossible to recover the original information. However, homomorphic encryption does not replace communication mechanisms for authentication and integrity, which may still be required in a complete operational system, but this was not the focus of our thesis.

- **Blind computation on the Cloud (Security in use):** Thanks to the algebraic properties of homomorphic encryption, the CKKS scheme allows mathematical operations to be performed directly on encrypted polynomials. When the Cloud has to compute the quadratic distance, it does not need to access the plaintext values or to allocate temporarily decryption keys in RAM. This will eliminate the critical vulnerability that instead we have to face with in TLS. In this way, the server performs the subsequent multiplication directly on the encrypted data, squaring each element of the difference vector according to the multiplicative homomorphic properties of CKKS.

- **Reduced trust in the Cloud infrastructure:** once the server has completed the homomorphic computation, the result is still returned as an encrypted ciphertext. The confidentiality of the processed data depends on the cryptographic protection provided by CKKS rather than on the assumption that the remote infrastructure is fully secure. This characteristic is consistent with the Zero-Trust approach adopted in the proposed architecture and discussed in next section.

The comparison among the three scenarios explains why homomorphic encryption was selected for the final architecture. The plaintext solution provides the lowest computational cost but offers no confidentiality, while TLS protects the information during transmission but still requires the Cloud to access plaintext data during processing. Instead, the combination of the Fourier-Mellin algorithm and CKKS homomorphic encryption is the only able to overcome the traditional trade-off between security and usability. For this reason, the use of homomorphic encryption represents the most suitable solution. The overall architecture is shown in Fig. 4.6 To support these theoretical considerations, we tested the architecture by evaluating the impact of the three scenarios on the overall system performance in terms of execution time, computational workload, and communication overhead. This will be done in Section 5.3.5.

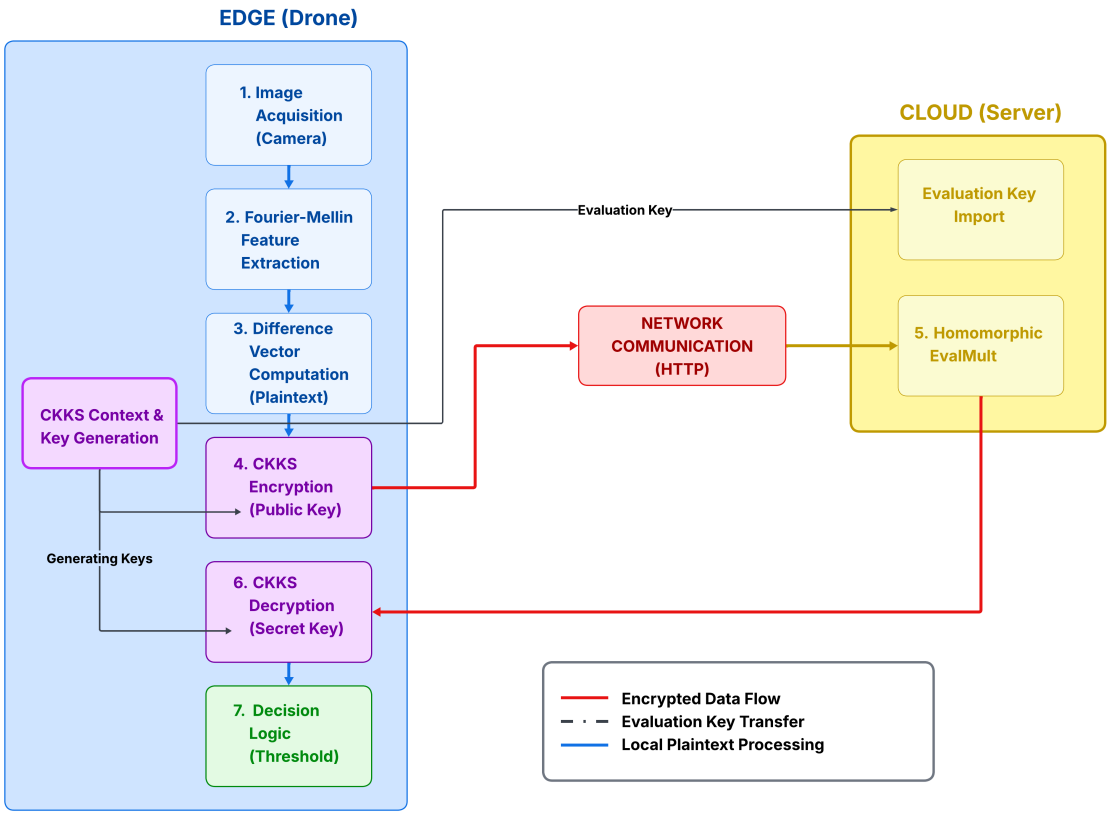


Figure 4.1. Overall Drone-Cloud architecture

4.2 Security Model and Trust Assumptions

The general principles of the Zero-Trust model and their relation to Fully Homomorphic Encryption were introduced in 1.3. This section focuses on the trust assumptions adopted for the implemented Drone-Cloud architecture and defines the security boundary considered in the experimental system.

The architecture distinguishes between two environments with different levels of trust. The UAV is considered the trusted component and can therefore process sensitive information in plaintext and retain the cryptographic material required for decryption. The remote Cloud, instead, is treated as an external computational resource that must not require access to sensitive plaintext information or to the UAV secret key. This design follows the Zero Trust Architecture defined by the National Institute of Standards and Technology (NIST), according to which entities should not be implicitly trusted only because of their network location or infrastructure domain [79].

The prototype adopts an honest-but-curious Cloud model. Under this assumption, the server is expected to correctly follow the protocol and execute the requested homomorphic computation, while at the same time it may attempt to learn information from the data available during processing. The use of CKKS limits this exposure by ensuring that the numerical information outsourced to the Cloud remains encrypted and that the secret key required for decryption is retained exclusively by the UAV.

The security objective considered in this work is therefore the confidentiality of outsourced data. Threats such as unauthorized access, compromised infrastructure, privileged internal users, or accidental disclosure motivate the decision not to expose plaintext information to the remote server. However, the prototype does not attempt to provide protection against every possible active attack on the communication or execution environment.

In particular, mutual authentication, message integrity verification, secure key management,

replay protection, and protection against actively malicious servers are outside the scope of this thesis. These mechanisms would be required for a complete operational deployment and could be integrated as future extensions. Docker containers are used in the experimental environment to reproduce the logical separation between the UAV and Cloud components, but they are not considered part of the cryptographic security guarantees.

Within these assumptions, FHE reduces the level of trust that must be placed in the remote infrastructure: the Cloud can perform the authorized computation on encrypted data, while access to plaintext information and the final recognition decision remain under the control of the trusted UAV.

4.3 Data Flow

To better understand how the proposed system operates, we can focus on the logical data flow and the mathematical transformations that data undergoes, starting from the acquisition of the image by the drone up to the final identification of the target. In particular, in the proposed solution, the data flow is not only a simple exchange of packets between the client and the server, but it consists of a coordinated sequence of signal-processing, cryptographic, and network operations performed in order to minimize communication overhead and preserving its confidentiality. The logical data flow implemented in our system can be divided into four sequential macro-phases:

1. **Initialization of the cryptographic context and exchange of keys**
2. **Image acquisition and extraction of local features through DFT and Fourier-Mellin**
3. **Optimization and local encryption of the difference vector**
4. **Homomorphic computation on the Cloud and generation of the final decision**

4.3.1 Initialization of the cryptographic context and exchange of keys

The initial phase of the system is performed only once during the startup phase, and it is done before the UAV begins the image matching procedure. During this phase, the drone initializes its CKKS cryptographic context through the C++ module based on the OpenFHE library and generates the cryptographic material required by the system. In the current implementation, the CKKS batch size is set according to the length of the Fourier-Mellin vector to be encoded in the available SIMD slots. For a 128×128 image, the extracted vector contains 16384 values; therefore, its length is used as the batch size. For this reason, the Client initializes the CKKS context using the length of the extracted vector as the batch size.

The corresponding Client-side initialization is shown in Fig. 4.2.

```
1  ckks = ckks_context.CKKSContext()
2
3  if 'temp_r' in locals():
4      ckks.init(len(temp_r))
```

Figure 4.2. Initialization of the CKKS context on the UAV.

The value passed to `ckks.init()` corresponds to the length of the extracted feature vector and is forwarded to the native C++ implementation. Inside the `init()` function, this value is used to configure the CKKS batch size together with the other cryptographic parameters. The function then creates the OpenFHE cryptographic context, generates the key pair, and produces the evaluation key required for homomorphic multiplication. The selection and experimental tuning of these parameters are discussed in detail in Section 5.1.3. The main cryptographic initialization operations performed by the C++ module are summarized in Fig. 4.3.

```
parameters.SetBatchSize(batchSize);
cc = GenCryptoContext(parameters);
keys = cc->KeyGen();
cc->EvalMultKeyGen(keys.secretKey);
```

Figure 4.3. Configuration of the CKKS context and generation of the cryptographic keys in the C++ module.

Once the cryptographic context has been initialized, the UAV keeps the generated secret key locally, since it is required for decryption and must never be transmitted to the Cloud. The multiplication evaluation key generated through `EvalMultKeyGen()` is instead required by the server to evaluate ciphertext multiplications. For this reason, only the multiplication evaluation key is exported by the UAV. It is serialized, encoded in Base64, and transmitted to the Cloud through an HTTP POST request to the `/set_evaluation_key` endpoint. The Cloud can then use this key to perform the authorized homomorphic multiplication without obtaining the secret key required to decrypt the ciphertexts.

The transmission of the multiplication evaluation key is implemented on the Client side as shown in Fig. 4.4.

```
evaluation_key = ckks.export_mult_key()

payload_key = {
    "eval_mult_key":
        base64.b64encode(evaluation_key).decode("utf-8")
}

requests.post(
    "http://cloud-server:5000/set_evaluation_key",
    json=payload_key
)
```

Figure 4.4. Export and transmission of the CKKS multiplication evaluation key.

On the Cloud side, the `/set_evaluation_key` endpoint receives the JSON payload and extracts the Base64-encoded key. The encoded value is converted back into its binary representation and imported into the server-side CKKS context through `import_mult_key()`.

After this exchange, the Cloud has received the evaluation key required to perform homomorphic multiplication on ciphertexts generated by the UAV as shown in Fig. 4.5. At the end of this initialization phase, both components are ready for the subsequent encrypted computation. The Cloud possesses only the evaluation material required to execute ciphertext multiplication, whereas the secret key remains exclusively on the UAV and will be used only during the final decryption phase.

4.3.2 Image acquisition and extraction of local features (RGB Fourier-Mellin)

In the second phase, the system acquires the image captured and locally extracts its frequency domain representation used for target matching. In the implemented prototype we tried different types of images being captured, the one chosen as example is represented by Fig. 4.6 on the left, while the reference image, target is shown in Fig. 4.6 on the right.

As described in detail in Subsection 3.2.2, each image is resized to a fixed resolution of 128×128 pixels, separated into its red, green, and blue channels, and processed through the function

```
key_b64 = data.get("eval_mult_key")
key_bytes = base64.b64decode(key_b64)

ckks.import_mult_key(key_bytes)
```

Figure 4.5. Import of the multiplication evaluation key into the Cloud server.



Figure 4.6. Captured image and reference of the target tank

`calculate_fourier_mellin_rgb()`. The function applies the complete Fourier-Mellin processing pipeline independently to each channel and returns three normalized one-dimensional real-valued vectors. These vectors constitute the spectral representation used by the matching procedure and are passed to the next phase, where the difference between the current image and the reference target is computed before encryption.

4.3.3 Optimization and local encryption of the difference vector

In the third phase, the two spectral representations obtained in the previous step are prepared for encrypted processing. Instead of encrypting and transmitting both vectors separately, the UAV computes their element-wise difference locally before encryption. This reduces the amount of data outsourced to the Cloud and removes the need to perform the subtraction homomorphically.

The resulting difference vector is then passed to the CKKS encryption function, where its values are encoded into the available SIMD slots and encrypted into a ciphertext. The main Client-side operations are shown in Fig. 4.7:

4.3.4 Homomorphic computation on the Cloud and local final decision

The fourth phase completes the matching procedure by combining the homomorphic computation performed by the Cloud with the final decision made by the UAV. The encrypted difference vector generated in the previous phase is transmitted to the Cloud through an HTTP POST request to the `/match_fourier_mellin` endpoint. After recovering the ciphertext from the received payload, the server uses the multiplication evaluation key obtained during the initialization phase to apply the `ckks.eval_mult()` operation.

More specifically, the encrypted difference vector is multiplied by itself, so that its components are squared homomorphically without exposing the corresponding plaintext values. The main operation performed by the server is shown in Fig. 4.8:

The resulting ciphertext is then returned to the UAV, where it is decrypted using the locally stored secret key. The decrypted values are aggregated into the spectral score used by the matching logic and compared with the predefined tolerance threshold. Based on this comparison, the UAV determines whether the target has been identified. The detailed implementation of the local decryption and decision logic is presented in the following sections.

```
1  diff_vector = arr_template - arr_scenografy
2  enc_temp_r = ckks.encrypt(diff_vector.tolist())
```

Figure 4.7. Local computation and encryption of the difference vector.

```
1  spectral_encrypted_distance = ckks.eval_mult(
2      spectral_diff_bytes,
3      spectral_diff_bytes
4  )
```

Figure 4.8. Homomorphic squaring of the encrypted difference vector on the Cloud.

4.4 Cloud Server Implementation (Flask/Docker)

The Cloud server is implemented by combining a Python communication layer with a native C++ cryptographic module. Flask, a lightweight Python web framework used to create and manage web services and HTTP endpoints, is employed to handle the HTTP requests exchanged with the drone, while the homomorphic operations are executed through the OpenFHE based module `ckks_context`. Since the initialization of the cryptographic context, the exchange of the evaluation key, and the matching operations have already been described in the previous sections, here we focus on how these software components are integrated and deployed within the experimental environment.

The connection between Python and C++ is provided by Pybind11. The binding layer exposes the methods implemented by the `CKKSContext` class to Python, allowing the Flask application to invoke the cryptographic functions without reimplementing them in the Python layer. In particular, operations such as the import of the multiplication evaluation key and the homomorphic multiplication can be called directly through `import_mult_key()` and `eval_mult()`.

The native extension is built through the script `setup.py`. This configuration includes the two C++ source files `ckks_context.cpp` and `bindings.cpp`, together with the required Pybind11 and OpenFHE headers. The extension is linked against the `OPENFHEpke` and `OPENFHEcore` libraries and compiled as the Python module `ckks_context`. Once we have built them, this module is imported by `cloud_server.py` and used by the Flask application.

The main elements of this configuration are summarized in Fig. 4.9:

The complete Cloud software is packaged inside a Docker container. By doing this the server can combine components belonging to different software environments; on one hand the communication layer written in Python, on the other hand the cryptographic engine compiled in C++ code and external OpenFHE libraries. Containerization makes it possible to keep these dependencies within the same isolated and reproducible environment rather than relying on the configuration of the host machine.

The server image is defined through the file `Dockerfile.server`. Its purpose is to prepare the environment required by both the Flask application and the native `ckks_context` extension. In particular, the container must provide the libraries, headers, and compilation environment required by OpenFHE and Pybind11 so that the native module can be built and loaded correctly.

The complete prototype is coordinated through the `docker-compose.yml` configuration file, which manages the Client and Server as separate services and creates the virtual network used for their communication. Within this network, the two components can communicate through their logical service names. This is reflected in the Client implementation, where the Cloud is addressed using URLs such as `http://cloud-server:5000`. Containerization therefore reproduces the distributed organization of the prototype and simplifies the deployment of its software dependencies. However, Docker is used for software isolation and reproducibility, while confidentiality is provided by the CKKS cryptographic layer.

```
Extension(
    "ckks_context",
    sources=[
        "ckks_context.cpp",
        "bindings.cpp"
    ],
    libraries=[
        "OPENFHEpke",
        "OPENFHEcore"
    ],
    language="c++"
)
```

Figure 4.9. Configuration of the native CKKS Python extension.

The current Flask implementation also contains the auxiliary endpoint `/upload`. This endpoint belongs to the separate FFT transmission and reconstruction path used during the development of the prototype and is not part of the Fourier-Mellin homomorphic matching flow considered in the final architecture.

Overall, the Cloud implementation combines three complementary software layers: Flask manages the communication interface, the native C++ module executes the OpenFHE operations, and Docker provides the execution environment in which these components are integrated.

4.5 Local Decryption and Decision Logic

The previous data-flow section introduced the return of the encrypted result to the UAV; this section focuses on the implementation of the local decryption and decision procedure.

The UAV extracts the encrypted result from the `risultato_r` field of the received response, decodes it, and passes the resulting ciphertext to `ckks.decrypt()`. At the C++ level, the ciphertext is de-serialized and decrypted using the secret key stored in the cryptographic context of the drone, after which OpenFHE returns the decoded packed values as a vector of real numbers. In the current proof-of-concept, the homomorphic matching stage uses only the red-channel feature vectors `temp_r` and `scen_r`. Consequently, `risultato_r` contains the homomorphically squared components associated with this channel. This choice reduces the encrypted data and homomorphic processing required to validate the complete workflow; extending the matching procedure to all three RGB channels is left as a future improvement.

After decryption, the Client converts the returned values into a NumPy array and computes the scalar spectral score used by the decision logic. In the current implementation, this score is obtained by summing the absolute values of the first 16384 decrypted components:

$$D = \sum_{i=1}^{16384} \left| d_i^2 \right|$$

where d_i represents the corresponding component of the difference vector computed locally before encryption.

The decrypted values are directly aggregated without an additional filtering stage; the numerical approximation introduced by CKKS is discussed in the next section. The resulting score is compared with the predefined tolerance threshold `SOGLIA_TOLLERANZA`, set to 0.02 in the current prototype. A score below the threshold indicates a sufficiently close spectral match and the target is considered identified; otherwise, the matching is rejected.

The main Client-side operations are shown in Fig. 4.10:

The threshold comparison and the resulting recognition decision are therefore performed locally on the UAV.

```
result_encrypted_string =
    fourier_encrypted_result["result_r"]

ciphertext_bytes = base64.b64decode(
    result_encrypted_string
)

distance_vector = np.array(
    ckks.decrypt(ciphertext_bytes)
)

spectral_distance = np.sum(
    np.abs(distance_vector[:16384])
)

THRESHOLD_TOLLERANCE = 0.02

if spectral_distance < THRESHOLD_TOLLERANCE:
    print("TARGET IDENTIFIED")
else:
    print("TARGET NOT IDENTIFIED")
```

Figure 4.10. Local decryption and final target recognition decision on the UAV.

4.6 Software Constraints and CKKS Integration

During the implementation of the proposed system, the integration between the Fourier-Mellin processing pipeline, the CKKS cryptographic layer, and the network communication interface introduced several practical constraints. These mainly concerned the numerical representation exchanged with the encryption module, the management of approximate homomorphic computation, and the serialization of ciphertexts for transmission between the UAV and the Cloud.

4.6.1 Data Representation and Compatibility with CKKS

One of the first problems concerned the representation of the data generated by the signal-processing pipeline. The FFT naturally produces complex valued coefficients, while the encryption interface developed for the project operates on vectors of real values. In the Fourier-Mellin algorithm path, this issue is avoided by using the magnitude of the transformed spectrum.

The resulting is normalized and flattened into a one-dimensional vector of real values, which can be directly passed to the CKKS encryption interface. This representation avoids the need to manage complex components separately during the homomorphic matching stage. A separate FFT reconstruction path implemented during development handles real and imaginary components independently, but it is not part of the Fourier-Mellin matching workflow considered in the final architecture.

4.6.2 Approximation and Homomorphic Depth Management

CKKS performs approximate arithmetic and may therefore introduce small numerical deviations between the decrypted result and the corresponding plaintext computation. The impact of these approximations becomes more relevant as the depth of the encrypted computation increases, particularly when several homomorphic multiplications are executed sequentially. For this reason, the encrypted part of the proposed matching procedure was intentionally kept short. The Cloud performs only the homomorphic squaring of the encrypted difference vector, while the subtraction is computed before encryption and the aggregation and threshold comparison are

performed locally on the UAV. This organization limits the multiplicative depth of the encrypted computation and reduces the accumulation of approximation errors.

The objective of the prototype is therefore not to move the complete matching algorithm to the Cloud, but to demonstrate that the required outsourced numerical operation can be executed directly on encrypted data. The OpenFHE parameters used to support this computation and their experimental tuning are discussed separately in Subsection 5.1.3.

4.6.3 Transport Encoding and Ciphertext Serialization

A further implementation issue concerns the transmission of CKKS ciphertexts between the UAV and the Cloud. OpenFHE serializes ciphertexts in binary form, whereas the communication interface of the prototype exchanges JSON messages over HTTP. The serialized ciphertexts are therefore encoded in Base64 before being inserted into the JSON payload and decoded back into binary form at the receiving side. The same procedure is applied to the encrypted response returned by the Cloud.

Base64 is used exclusively as a transport encoding mechanism and does not provide cryptographic protection. Its purpose is to make the binary OpenFHE representation compatible with the JSON-based communication layer, while confidentiality is provided by the CKKS encryption of the transmitted payload.

Chapter 5

Experimental Analysis and Discussion of Results

This chapter presents and discusses the experimental results of the Drone-Cloud architecture developed in this thesis. The main purpose is to understand how the proposed system behaves both in terms of target recognition and computational performance.

The first part of the analysis focuses on the matching procedure. The system is tested under four different scenarios: a nominal case (Auto-matching), a negative scenario (desert), a geometric variation of the target and the one with a more complex urban background. These tests are used to understand if the Fourier-Mellin algorithm and the CKKS scheme are able to provide the right matching decision with different image.

The second part focuses on performance. We analyzed the execution times of the main stages of the CKKS workflow, including cryptographic initialization, encryption, communication with the Cloud, remote processing, and local decryption. Since the experiments are performed in a Docker based environment, the measured values are used to study the computational cost of the prototype rather than to represent the performance of a real drone.

Finally, the CKKS solution is compared with two simpler approaches: plaintext communication without encryption, and communication protected through TLS. The comparison considers the different levels of data protection, the workload introduced on the drone, the type of processing performed by the Cloud, and the communication overhead. In this way, the chapter highlights the main trade off of the proposed architecture: CKKS provides stronger protection of the outsourced data, but at the cost of a higher computational overhead.

5.1 Test Environment Setup

The experimental environment was designed to provide a controlled and reproducible setup for evaluating the implemented Drone-Cloud architecture. The following subsections describe the organization of the Client and Server containers, and the cryptographic parameters adopted for the CKKS implementation. The dataset and the different validation scenarios are presented separately in the next section.

An overview of the experimental testbed is provided in Fig. 5.1. The figure summarizes the organization of the physical host, the Docker-based execution environment, the Drone Client and Cloud Server containers, and the communication between the two components. It also shows the three evaluated configurations—Plaintext, TLS, and CKKS together with the input scenarios and the main metrics collected during the experiments. This representation provides the context for the hardware, software, and container configuration described in the following subsections.

5.1.1 Hardware and Software Environment

All the experimental measurements were collected using the same physical host machine. The main characteristics of the experimental environment are summarized below:

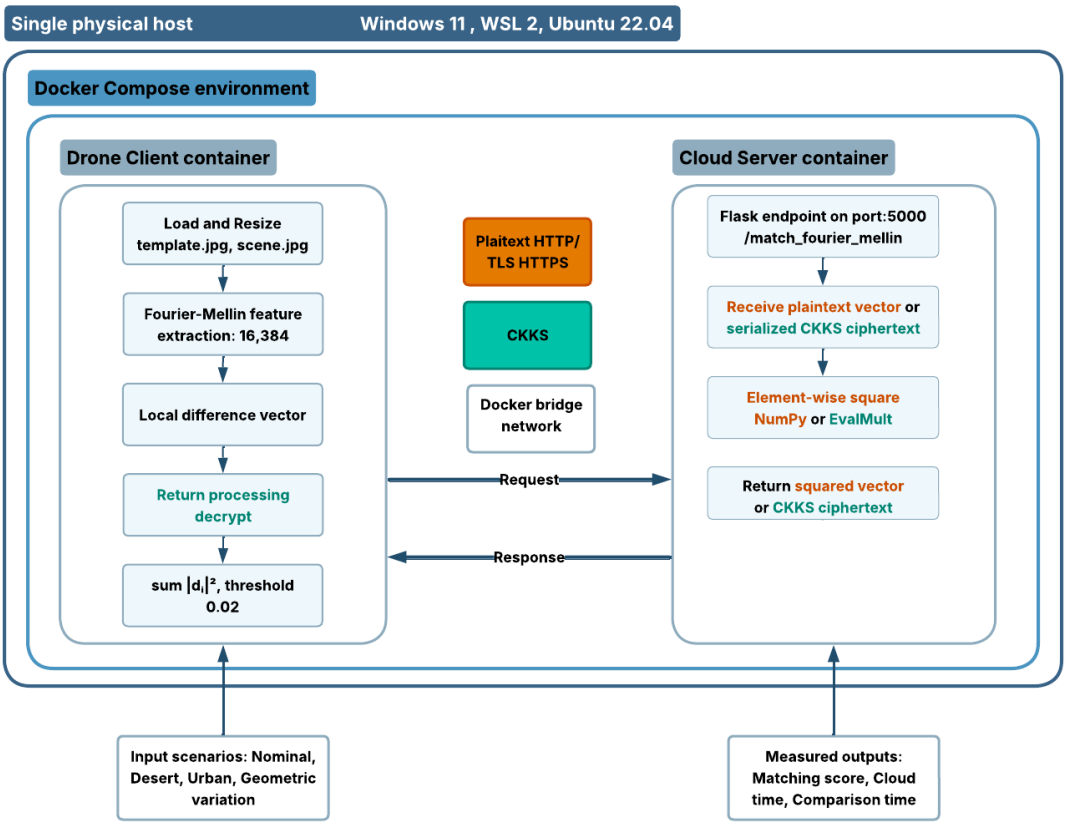


Figure 5.1. Experimental setup used for the performance evaluation.

- **Processor:** 11th Generation Intel Core i5-1135G7, with a base frequency of 2.40 GHz, 4 physical cores and 8 logical processors.
- **Memory:** 8 GB of installed RAM.
- **Operating system:** Microsoft Windows 11 Home, with WSL 2 running Ubuntu 22.04.5 LTS as the Linux based execution environment.

We executed the implemented architecture using **Docker** and **Docker Compose**, with the Client and Server running in two separate containers connected through a Docker virtual network.

Both containers were executed on the same physical machine and therefore shared the available CPU and memory resources. We did not use GPU acceleration or dedicated cryptographic hardware, but all image processing and CKKS operations were executed on the CPU.

Since the Client and Server communicated within the Docker environment, the measured times do not include the effects of a real distributed network, such as wireless latency, bandwidth limitations, packet loss, or interference. Therefore, the execution times reported in this chapter refer to this hardware and software configuration.

5.1.2 Containers Architecture

The Client and Cloud Server were executed in separate Docker containers in order to reproduce the logical separation between the UAV and the remote processing node, while keeping their software environments isolated at container level. Communication between the two components takes place through a dedicated Docker virtual network called `drone-net`.

The environment consists of two main containers:

- **The Drone module (`drone-client`):** this container represents the onboard processing unit of the UAV. It executes the Python Client together with the native C++ module used for CKKS operations. The Client performs the local processing stages, including image loading, Fourier-Mellin feature extraction, computation of the difference vector, and encryption before transmission. After receiving the result from the Cloud, it decrypts the returned ciphertext using the locally stored secret key and performs the threshold comparison required for the final decision.

- **The Server infrastructure (`cloud-server`):** this container represents the remote processing node, corresponding to the Cloud server. It hosts the Flask application that receives HTTP requests from the Client on port 5000. The server performs the homomorphic squaring of the encrypted difference vector using the multiplication evaluation key, and returns the resulting ciphertext to the drone for local decryption and final processing.

Although the two components are logically separated, both containers run on the same physical host and so they share its computational resources. Consequently, the experimental environment does not reproduce the characteristics of a real wireless communication channel, such as network latency, packet loss, interference, or bandwidth limitations. Moreover, the measured execution times remain dependent on the CPU and memory resources of the host machine.

The setup we implemented, based on Docker, is mainly used to provide software isolation, reproducibility, and a controlled logical separation between the Client and Server components. The collected measurements are suitable for evaluating the computational overhead of the prototype under controlled conditions, while the performance of a real distributed deployment would additionally depend on the UAV hardware, the Cloud infrastructure, and the communication network.

5.1.3 Cryptographic Parameter Tuning (OpenFHE)

As described in Section 4.3.1, the CKKS cryptographic context is initialized on the drone before the matching procedure begins. This section focuses specifically on the selection of the OpenFHE parameters used in the experimental implementation, on their impact on the number of values that can be packed into a single ciphertext, on numerical precision, computational cost, and homomorphic depth.

During the development and tuning phase, the following parameters were configured:

- **Parallel vector capacity (`BatchSize`):** defines the number of values encoded and processed in parallel through the SIMD mechanism provided by CKKS. In the proposed implementation, this value is selected according to the length of the Fourier-Mellin vector. Since a 128×128 representation produces 16384 values, the batch size is set to 16384, allowing the complete vector used for matching to be packed into the CKKS slots and processed in parallel.

- **Polynomial ring dimension (`RingDim`):** defines the degree of the polynomial ring used by CKKS and directly affects the size of the cryptographic structures and the computational cost. In CKKS, the number of available complex slots is at most half of the ring dimension, so `RingDim` must be at least twice the number of slots required to encode the input vector. In the experimental implementation, it was set to 65536. We selected this value because it is large enough to contain the complete 16384 element vector in a single ciphertext. A larger ring dimension increases the size of the polynomial structures involved in the cryptographic operations and therefore also increases memory consumption and execution time.

- **Multiplicative depth:** determines the number of sequential multiplication levels supported by the configured CKKS modulus chain. In this work we adopted a multiplicative depth of 5, this provides a conservative margin for the encrypted operations performed. The current matching procedure requires only one homomorphic multiplication, used for the squaring operation through `EvalMult()`.

- **Scaling modulus size (`ScalingModSize`):** this parameter determines the bit length of the scaling factor used by CKKS to represent real-valued numbers with approximate arithmetic. In the experimental configuration, it was set to 50 bits, corresponding to a scaling factor

```
1      parameters.SetMultiplicativeDepth(5);
2      parameters.SetScalingModSize(50);
3      parameters.SetBatchSize(batchSize);
4      parameters.SetRingDim(65536);
5      parameters.SetSecurityLevel(HEStd_NotSet);
```

Figure 5.2. OpenFHE CKKS parameter initialization.

of approximately 2^{50}. A larger scaling factor allows fractional values to be represented with higher numerical precision, reducing the quantization error introduced during CKKS encoding and homomorphic computation. However, increasing the scaling modulus also enlarges the ciphertext modulus and therefore increases the computational and memory cost of the cryptographic operations. The value of 50 bits was therefore selected as a conservative compromise between numerical precision and computational overhead for the Fourier-Mellin values processed in the proposed matching procedure.

- **Security level (`HEStd_NotSet`):** this parameter controls whether OpenFHE automatically applies a predefined standardized security level. In the experimental configuration, it was set to `HEStd_NotSet`, meaning that no specific standard security level is automatically set, and the main CKKS parameters were configured manually.

The corresponding OpenFHE configuration is shown in Fig. 5.2.

The selected configuration was used throughout the experimental evaluation. The batch size was set according to the length of the Fourier-Mellin vector, while the remaining CKKS parameters were chosen conservatively in order to provide a stable configuration with sufficient numerical precision for the encrypted computation. These values should therefore be considered part of the experimental setup rather than an optimized parameter configuration. The performance results presented in the following sections evaluate the overall computational overhead of the resulting CKKS workflow, while the individual impact of each cryptographic parameter was not separately analyzed.

5.2 Dataset and Validation Scenarios

The experimental evaluation was performed using a set of images representing the reference target and different scenarios. All images were processed at a fixed resolution of 128×128 pixels, providing a compromise between the amount of details used for matching and the dimensionality of the vectors processed by CKKS.

Moreover, four different scenarios were considered: an ideal auto-matching case, a negative case in which the target is absent, a geometrically modified target, and a target embedded in a more complex background. These tests were designed to evaluate whether the complete encrypted workflow preserves a sufficient separation between matching and non matching cases under different conditions.

5.2.1 Nominal Case (Auto-Matching)

The first experiment represents the ideal matching condition and is used as a reference for the following tests. In this case, the same image is used both as the reference template and as the test scene. Therefore, the Fourier-Mellin representations produced for the comparison are expected to be identical. In fact, the encrypted processing returns a final matching score of approximately 0.000, which remains below the tolerance threshold of 0.02. The target is therefore correctly identified. This experiment confirms that the CKKS processing does not alter the expected matching decision under ideal conditions.

5.2.2 Negative Scenario (Desert Environment)

The second experiment evaluates the behavior of the system when the target is completely absent, the test scene contains just a desert landscape. The Fourier-Mellin representation extracted from this scene differs from the representation obtained from the tank template. As a consequence, the local subtraction produces a difference vector with larger component values than in the nominal case. After the result is returned and decrypted by the drone, the final matching score is approximately 0.03. Since this value is above the tolerance threshold of 0.02, the scene is correctly classified as a non matching case. This experiment confirms that, when the target is absent, the matching procedure based on CKKS correctly work.

5.2.3 Geometric Variation Scenario

The third experiment evaluates the behavior of the system when the target is still present, but its orientation and size differ from those of the reference template. The Fourier-Mellin representation is used to reduce sensitivity to rotation and scale variations. However, in the CKKS configuration, the final matching score obtained for this scenario is approximately 1.8, which is well above the tolerance threshold of 0.02. The system therefore returns a negative matching decision, producing a false negative. An important observation is that the same geometric scenario is correctly identified in the plaintext and TLS configurations, as discussed later in the performance comparison. In these cases, the matching score remains close to 0.017 and therefore below the selected threshold. For this reason, the negative result obtained with CKKS cannot be explained only by the rotation and scaling applied to the input image. This can be explained by taking into account that CKKS uses approximate arithmetic, and when dealing with the encoding, homomorphic multiplication, decryption, and decoding stages they may introduce small numerical deviations. Since the final matching score is obtained by aggregating the values of a vector composed of 16,384 elements, these deviations may also accumulate in the final result. A more detailed comparison with the plaintext and TLS configurations is presented in the following sections.

5.2.4 Complex Urban Background

The fourth experiment considers a more realistic scene in which the target is placed inside a complex urban environment. The vehicle is surrounded by buildings, shadows, road elements, and other visual structures that are not present in the reference template. Since the Fourier-Mellin representation is computed over the complete image, the surrounding environment also contributes to the resulting spectrum. The extracted representation therefore contains information associated not only with the target but also with the geometric structures of the background. As a consequence, the difference vector obtained comparing to the isolated reference target is larger than it is in the nominal case. In the performed experiment, the final matching score is approximately 0.005, which remains below the tolerance threshold of 0.02. The target is therefore correctly identified. This result shows that the current approach can still recognize the target in the tested urban scene, although the surrounding background contributes to the global spectral representation.

5.2.5 Final Considerations on the Four Experiments

The four experiments highlighted both the strengths and the limitations of the current prototype. In the CKKS configuration, the nominal and urban scenarios are correctly identified, while the desert scene is correctly rejected. The main difference appears in the geometric variation scenario, where the CKKS score increases to approximately 1.8 and produces a false negative. However, the same scenario remains below the decision threshold in the Plaintext and TLS configurations as already anticipated. This result shows that the rotation and scale variation applied to the target cannot fully explain the different decision obtained with CKKS. The geometric transformation modifies the Fourier-Mellin representation before encryption, but this difference is not sufficient to explain the CKKS result, since the same input is correctly recognized in the other two configurations. A possible additional contribution comes from the approximate numerical processing introduced by CKKS. In particular, small deviations may be introduced due

to the encoding and decoding of the values, the limited numerical precision of the scheme, and the homomorphic multiplication. Since the final score is obtained by aggregating 16,384 values, these deviations can also accumulate and have a larger effect on the final result. The specific CKKS parameter configuration, especially the scaling and precision settings, may also influence the numerical accuracy of the computation. These factors represent possible explanations for the different behavior observed in the geometric scenario. More generally, the experiments show that changes in target scale and orientation, as well as complex backgrounds, can modify the Fourier-Mellin representation and therefore affect the final matching score. These differences are already introduced during the feature extraction stage, before encryption takes place. For this reason, the final matching decision cannot be based on an exact comparison with zero. A tolerance threshold is therefore used to determine whether the obtained spectral distance is sufficiently small to consider the two representations as a matching case. If the final score is below the threshold, the target is considered identified; otherwise, the scene is classified as a non matching case. The threshold represents the boundary between acceptable variations of the same target and differences that are considered too large for a reliable match. We have decided its value experimentally, testing different cases and taking into consideration their final decision. A threshold that is too high may increase the number of false positives, while a threshold that is too low may reject valid matches and increase the number of false negatives.

In future developments, additional preprocessing and target isolation techniques could be introduced to improve the robustness of the matching procedure.

5.3 Performance Evaluation

Once we have validated the behavior of the system through the different test scenarios, the next step was to evaluate the performance of the implemented architecture. The aim of this analysis is to understand the computational overhead introduced by the solution based on CKKS, and to identify the operations that have the greatest impact on the overall execution time. Since the proposed architecture is developed to be used on UAV platforms, the computational cost of the Client-side cryptographic operations is particularly relevant. The measurements collected in our experimental environment do not reproduce the performance of a specific physical drone, but they provide an indication of the relative cost of the different processing stages and of the additional overhead introduced by homomorphic encryption.

We have focused the evaluation on two main aspects:

- **Execution time:** the timing measurements introduced in the Client are used to analyze the main stages of the prototype, including CKKS initialization, local encryption, communication with the Cloud and remote processing, and local decryption of the returned ciphertext.
- **Impact of CKKS approximation on the matching decision:** the experiments are also used to verify if the approximate numerical results produced by CKKS remain suitable for the final decision logic. Since CKKS performs approximate arithmetic, small numerical deviations may appear after decryption. The purpose of this analysis is therefore not to directly quantify the CKKS approximation error, but to verify whether the returned values still allow the system to distinguish between matching and non matching cases using the selected decision threshold.

This analysis allows us to identify the main computational bottlenecks of the prototype and to evaluate the trade off between the additional privacy provided by homomorphic processing and the corresponding performance overhead. The results should therefore be interpreted as an evaluation of the current experimental implementation rather than as a direct measurement of the performance achievable on a real UAV platform.

5.3.1 Analysis of CKKS Setup and Client-Side Computational Times

The timing parameters implemented in our architecture measure the main CKKS setup and Client-side cryptographic stages of the implemented workflow:

1. **CKKS initialization time:** the OpenFHE cryptographic context is initialized and the cryptographic material required by the CKKS workflow is generated, including the key pair and the multiplication evaluation key.
2. **Evaluation key exchange time:** measures the time required to export the multiplication evaluation key from the local CKKS context, encode it in Base64, transmit it to the Cloud through the `/set_evaluation_key` endpoint, and receive the corresponding server response. This operation is performed right after CKKS initialization phase and provides the Cloud with the evaluation material required to execute the homomorphic multiplication.
3. **Encryption time:** the locally computed difference vector is encrypted before being transmitted to the Cloud.
4. **Local decryption time:** after receiving the encrypted result from the Cloud, the Client decrypts the returned ciphertext using the locally stored secret key.

We have considered these time measurements for the four test scenarios to provide an indication of how the computational cost is distributed among the main CKKS setup and cryptographic stages and to estimate the overhead introduced by the CKKS-based processing. Since the CKKS parameters, the evaluation key size, and the encrypted vector size remain unchanged across the different scenarios, the execution times are expected to remain within the same order of magnitude. For each scenario, the reported execution times correspond to the average of three independent executions, as shown in Table 5.1.

- **Nominal case (auto-matching):** the initialization of the CKKS context requires approximately 2.368 seconds, and the key exchange around 1.284, while encryption of the difference vector takes about 1.797 seconds. Local decryption requires approximately 0.523 seconds.
- **Negative scenario (desert environment):** the CKKS initialization requires approximately 3.342 seconds, and the key exchange around 3.149, while encryption requires about 1.272 seconds. Local decryption remains close to 0.640 seconds.
- **Urban scenario:** the CKKS initialization is around 2.423 seconds, and the key exchange approximately 3.762, while encryption requires about 1.427 seconds. The local decryption time remains close to 0.688 seconds.
- **Geometric variation scenario:** the CKKS initialization is approximately 4.321 seconds, and the key exchange around 4.362, while encryption requires about 2.713 seconds. Local decryption again remains close to 1.239 seconds.

Scenario	**Initialization [s]**	**Key Exchange [s]**	**Encryption [s]**	**Decryption [s]**
Nominal	2.368	1.284	1.797	0.523
Desert	3.342	3.149	1.272	0.640
Urban	2.423	3.762	1.427	0.688
Geometric Variation	4.321	4.362	2.713	1.239

Table 5.1. Average execution times of the main cryptographic operations of the CKKS workflows for the four validation scenarios.

The results show that the measured CKKS setup and cryptographic stages remain within the same order of magnitude across the different scenarios, although some execution variability can be observed. This behavior is expected because the CKKS configuration, the size of the evaluation key, and the size of the encrypted difference vector remain fixed. Therefore, the amount of visual details present in the input image does not directly determine the computational cost of these operations. The observed variations can instead be associated with the experimental execution environment, including CPU load, memory allocation, Docker container scheduling, network communication between the containers, and internal OpenFHE operations. In particular, the evaluation key exchange includes not only local processing, but also key export, Base64 encoding, HTTP transmission, server-side reception and import, and the return of the corresponding response.

Among the measured stages, CKKS initialization and evaluation key exchange represent the largest setup costs overall. Encryption introduces an additional significant Client-side computational cost, whereas local decryption is consistently the least expensive of the measured cryptographic operations. These results confirm that the main overhead is associated with the cryptographic setup and processing required by the CKKS workflow rather than with the visual content of the input images. It is also important to observe that CKKS initialization and evaluation key exchange are performed during the initial setup of the system. Therefore, when the same cryptographic context and evaluation key are reused for multiple matching operations, these costs do not necessarily need to be incurred for every processed image and can be amortized over multiple matching requests.

For a future deployment on UAVs with limited hardware resources, further software optimization and the possible use of hardware acceleration for homomorphic encryption could be considered to reduce the computational overhead of the cryptographic operations.

5.3.2 Analysis of Cloud and Comparison Times

This section introduces the two timing measurements used to evaluate the performance of the matching procedure: the **Cloud time** and the **Comparison time**. These two measurements are used to isolate the part of the workflow that is affected by the different protection mechanisms considered in the next experimental evaluation.

- **Cloud time**: represents the complete request-response phase between the Client and the Cloud. It starts when the Client sends the data to the Server and ends when the corresponding result is received. This measurement therefore represents only the part of the comparison procedure associated with the communication with the Cloud and the computation performed on the Server.

- **Comparison time:** is measured starting from the moment in which the difference vector has already been computed and ends when the final matching score is obtained leading to de identification or not of the target. All the operations performed before this point, including image acquisition, image resizing, Fourier-Mellin feature extraction, normalization, and computation of the difference vector, are not included in the comparison time. These operations are identical in the plaintext, TLS, and CKKS configurations and therefore do not contribute to the performance differences introduced by the different security mechanisms.

In the CKKS configuration, the Comparison time additionally includes the cryptographic initialization, the evaluation key exchange with the Cloud, the local encryption of the difference vector, and the local decryption of the returned ciphertext.

The Cloud and comparison times measured for the four CKKS validation scenarios are shown in Tab. 5.2:

- **Nominal case (Auto-matching):** the Cloud phase requires approximately 1.392 seconds, while the Comparison phase is approximately 7.421 seconds.

- **Negative scenario (desert environment):** the Cloud phase is approximately 3.316 seconds, with a Comparison phase of about 11.794 seconds.

- **Urban scenario:** the Cloud phase is approximately 4.036 seconds, while the Comparison phase is around 12.417 seconds.

- **Geometric variation scenario:** the Cloud phase is around 4.672 seconds, with a Comparison phase of about 17.635 seconds.

The results provide an overview of the execution times associated with the CKKS workflow, considering both the Cloud time and the Comparison time. These measurements will be used in the following section to compare the performance of the plaintext, TLS, and CKKS configurations and to evaluate the additional overhead introduced by homomorphic encryption.

Scenario	Cloud Time [s]	Comparison Time [s]
Nominal	1.392	7.421
Desert	3.316	11.794
Urban	4.036	12.417
Geometric Variation	4.672	17.635

Table 5.2. Average execution times of the Cloud time and Comparison time for the CKKS configuration.

5.3.3 Performance Comparison between Plaintext and TLS

To better evaluate the cost introduced by homomorphic encryption, we have considered the matching workflow under three different configurations in terms of protection and security: plaintext communication over HTTP, communication protected through TLS, and CKKS based encrypted processing.

In the plaintext scenario, the difference vector is transmitted directly to the Cloud over HTTP. The Server receives the numerical values in clear form, squares each component, and returns the resulting vector to the Client, where the values are summed to obtain the final matching score. This configuration represents the reference case with the lowest computational overhead, as no security mechanism is applied to protect the data during transmission or processing.

The TLS scenario follows the same workflow and performs the same element wise squaring operation on the Server. The main difference with plaintext is that the communication between the Client and the Server is protected through TLS. Therefore, the data are protected while being transmitted, but once they reach the Server application they are available in plaintext for processing. TLS introduces some overhead related to the protection it has to ensure on the communication channel, while the numerical computation we have to perform is the same.

In the CKKS scenario, the difference vector is encrypted locally before transmission. The Cloud receives a ciphertext and performs the element wise squaring directly in the encrypted domain. The resulting ciphertext is then returned to the Client, where it is decrypted locally and aggregated to obtain the final matching score. In this case, the additional overhead is introduced by cryptographic context initialization, key management, encryption, homomorphic processing, ciphertext serialization and transmission, and final decryption.

So we have focused the comparison on the execution time required from when the difference vector is available until we obtain the final matching score. From a performance perspective, the three configurations are expected to show increasing computational and communication cost from plaintext to TLS and finally to CKKS. The comparison therefore highlights the main trade off between performance and confidentiality. Plaintext provides the best execution efficiency but no protection, TLS provides protection for data in transit with limited additional cost, while CKKS offers confidentiality during remote processing at the price of a significantly higher computational and communication overhead.

Plaintext Configuration

As we have done for CKKS, we use across the three configurations the same tolerance threshold of 0.02. This allows the final recognition decision to be evaluated using the same criteria independently of the protection mechanism applied to the data. For each scenario, the reported Cloud and comparison times correspond to the average of three independent executions, as shown in Tab.5.3, are:

- **Nominal case:** the difference vector is equal to zero with a final matching score of 0.000000. The measured Cloud time is approximately 0.0732 seconds, while the Comparison time measured is approximately 0.0950 seconds.

- **Desert scenario:** the difference between the two spectral representations increases, resulting in a matching score of 0.0304. Since this value is above the selected threshold of 0.02, the target is classified as not identified. The Cloud phase requires approximately 0.0764 seconds, while the Comparison time is approximately 0.0951 seconds.

- **Urban scenario:** the matching score is equal to 0.005003. Although the background differs from the reference image, the score remains below the selected threshold and the target is therefore identified. The measured Cloud time is approximately 0.0759 seconds, while the Comparison time is approximately 0.09058 seconds.
- **Geometric variation scenario:** this case produces a matching score of 0.0168. Although rotation and scaling increase the difference compared to the nominal case, the resulting score remains below the tolerance threshold of 0.02 so target is therefore identified. The measured Cloud time is approximately 0.0818 seconds, while the Comparison time is approximately 0.1396 seconds.

Scenario	**Cloud Time [s]**	**Comparison Time [s]**
Nominal	0.0732	0.0950
Desert	0.0764	0.0951
Urban	0.0759	0.09058
Geometric Variation	0.0818	0.1396

Table 5.3. Average execution times of the Cloud time and Comparison time for Plaintext

The plaintext configuration shows relatively stable execution times across the four scenarios. This behavior is expected because the size of the processed difference vector remains fixed, and the Cloud performs the same element wise squaring operation independently of the image content. The small differences we are dealing with can be associated with normal execution variability, container scheduling, serialization, and communication overhead.

These results provide the plaintext reference against which the additional overhead introduced by TLS and CKKS can be evaluated.

TLS Configuration

For the TLS configuration, we have maintained the same matching procedure and the same tolerance threshold we have used for the plaintext configuration. The main difference is about the communication channel between the Client and the Cloud, which is protected through TLS. This meaning that the difference vector is protected during transmission, while the Cloud processes it in plaintext after it has been established a secure connection. For each scenario, the reported Cloud and Comparison times correspond to the average of three independent executions, as shown in Tab. 5.4:

- **Nominal case:** the matching score is 0.000000. Therefore, the target is correctly identified. The Cloud time is approximately 0.052 seconds, while the Comparison time is approximately 0.062 seconds.
- **Desert scenario:** the difference between the two spectral representations produces a matching score of 0.030408. Since this value is above the tolerance threshold of 0.02, the target is classified as not identified. The Cloud time is approximately 0.129 seconds, while the Comparison time is approximately 0.147 seconds.
- **Urban scenario:** the matching score is equal to 0.005003. Although the background differs from the reference image, the score remains below the selected threshold and the target is therefore identified. The Cloud time is approximately 0.078 seconds, while the Comparison time is approximately 0.096 seconds.
- **Geometric variation scenario:** the rotation and scaling applied to the target produce a matching score of 0.016873. Since this value remains below the tolerance threshold of 0.02, the target is identified. The Cloud time is approximately 0.084 seconds, while the Comparison time is approximately 0.106 seconds.

The TLS configuration shows execution times of the same order of magnitude across the four validation scenarios. As in the plaintext configuration, the size of the processed difference vector remains fixed and the Cloud performs the same element wise squaring operation in every test.

Scenario	Cloud Time [s]	Comparison Time [s]
Nominal	0.052	0.062
Desert	0.129	0.147
Urban	0.078	0.096
Geometric Variation	0.084	0.106

Table 5.4. Average execution times of the Cloud time and Comparison time for TLS

5.3.4 Final Comparison between Plaintext, TLS and CKKS

Matching results

Before comparing the matching results and execution times, Fig. 5.3 provides an overview of the data flow and protection scope of the three evaluated configurations. The local preprocessing and final matching decision remain conceptually similar in all cases, while the main differences concern how the difference vector is transmitted and processed by the Cloud. In the Plaintext configuration, the vector is transmitted and processed without confidentiality protection. TLS protects the data while they are transmitted, but the Cloud recovers the vector in plaintext after terminating the secure channel. In the CKKS configuration, instead, the vector is encrypted locally and the Cloud performs the required squaring operation directly on the ciphertext, without accessing the underlying numerical values.

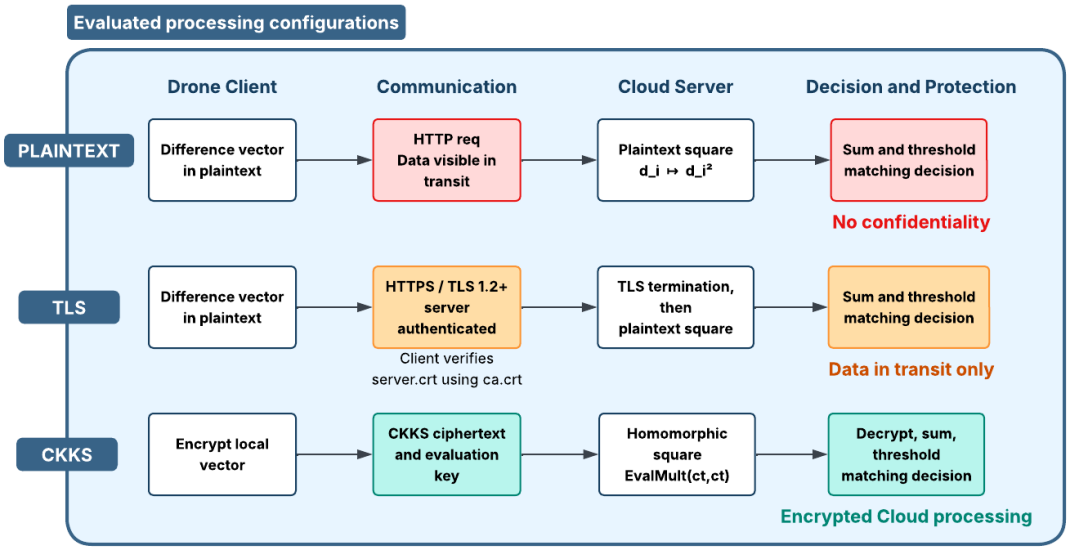


Figure 5.3. Comparison of the data flow and protection scope of the Plaintext, TLS, and CKKS configurations.

Based on these functional and security differences, the following comparison examines both the matching results obtained in the considered scenarios and the computational overhead introduced by each configuration.

In Tab. 5.5, we have summarized the matching results of Plaintext, TLS, and CKKS:

To ensure a consistent comparison, we used the same threshold of 0.02 in all three configurations. In the nominal, desert, and urban scenarios, Plaintext, TLS, and CKKS produce similar matching scores and lead to the same final decision. The target is correctly identified in the nominal and urban scenarios, while it is correctly rejected in the desert scenario.

The main difference appears in the geometric variation scenario. In this case, Plaintext and TLS produce matching scores close to 0.0168, which remain below the selected threshold and therefore lead to a positive identification. Their similar behavior is expected because the numerical

Scenario	Plaintext / TLS	CKKS
Nominal	0.000000	0.000000
Desert	0.030408	0.030408
Urban	0.00500	0.00500
Geometric Variation	0.016873	≈ 1.8

Table 5.5. Matching scores obtained with Plaintext/TLS and CKKS configurations

matching operation performed in the two configurations is the same, while TLS only changes the protection of the communication channel.

In the CKKS configuration, instead, the score increases to approximately 1.8, which is well above the threshold and produces a false negative. This difference may be related to the different numerical path followed by CKKS. In particular, CKKS uses approximate arithmetic, and the encoding, homomorphic multiplication, decryption, and decoding stages may introduce numerical deviations with respect to the corresponding plaintext computation. Since the final matching score is obtained by aggregating 16,384 values, these deviations may also accumulate in the final result. However, the value obtained in the geometric variation scenario should not be interpreted as a direct measurement of the CKKS approximation error, since the current experiments do not isolate the numerical contribution of each stage of the encrypted workflow.

Performance comparison

In Table 5.6 we summarize the measurements time described in the previous section for the different security configurations across the four validation scenarios. For each configuration, we consider only the Cloud time and the Comparison time because they are the ones that are directly affected by the different protection mechanisms.

Scenario	Plaintext		TLS		CKKS	
	Cloud	Comparison [s]	Cloud	Comparison [s]	Cloud	Comparison [s]
Nominal	0.0732	0.0950	0.052	0.062	1.392	7.421
Desert	0.0764	0.0951	0.129	0.147	3.316	11.794
Urban	0.0759	0.0906	0.078	0.096	4.036	12.417
Geometric Variation	0.0818	0.1396	0.084	0.106	4.672	17.635

Table 5.6. Final comparison for Plaintext, TLS, and CKKS configurations.

From a performance point of view, the results reported in Table 5.6 show a clear difference between the Plaintext/TLS configurations and CKKS. Plaintext and TLS remain within the same order of magnitude in all four scenarios, with Cloud and Comparison times generally on the order of 10^{-1} seconds. The TLS configuration introduces the additional mechanisms required to protect the communication channel, but the measured results do not show a systematic increase compared to Plaintext. In some scenarios, TLS is slightly slower, while in others the measured time is comparable or even lower. These small differences can be associated with the normal variability of the experimental environment and are considerably smaller than the overhead introduced by CKKS.

The difference becomes much more evident when CKKS is considered. For the Cloud time, CKKS requires between approximately 19× and 57× the execution time measured for Plaintext, depending on the scenario. Considering the average values over the four experiments, the Cloud time increases from approximately 0.077 s for Plaintext and 0.086 s for TLS to approximately 3.354 s for CKKS. This corresponds to an average increase of about 44× compared to Plaintext and 39× compared to TLS.

A substantially larger difference can be observed for the Comparison time. Plaintext and TLS require on average approximately 0.105 s and 0.103 s, respectively, whereas the CKKS configuration requires approximately 12.317 s. Therefore, the average CKKS Comparison time is approximately 117× higher than Plaintext and 120× higher than TLS. Considering the individual scenarios, the increase with respect to Plaintext ranges from approximately 78× in the nominal

case to approximately 137× in the urban scenario. The desert and geometric variation scenarios show increases of approximately 124× and 126×, respectively.

These results show that the additional protection provided by CKKS has a significant impact on the end-to-end comparison latency. While the Cloud-side processing alone introduces an overhead of several tens of times with respect to Plaintext and TLS, the complete CKKS Comparison time is approximately two orders of magnitude higher on average. This larger difference is expected because the Comparison time includes not only the homomorphic evaluation performed by the Cloud, but also the CKKS initialization, the evaluation key exchange, the local encryption of the difference vector, the serialization and transmission of ciphertexts, and the local decryption of the returned result. It should also be observed that CKKS initialization and evaluation key exchange belong to the initial setup phase of the system. Therefore, although their cost is included in the Comparison time considered in this experimental evaluation, these operations could be reused across multiple matching requests when the same cryptographic context and evaluation key remain active. In such a scenario, their cost would be amortized over multiple comparisons and the per-request overhead would be lower than the end-to-end values reported here.

The comparison therefore quantifies the main trade-off of the proposed architecture. Plaintext provides the lowest computational cost but offers no protection for the transmitted or processed data, while TLS protects the communication channel with a comparatively limited performance impact. CKKS, instead, extends confidentiality to the outsourced computation itself, preventing the Cloud from accessing the plaintext values, but this stronger protection results in a substantially higher computational and communication overhead in the current prototype.

5.3.5 Security-Performance Trade-offs and Practical Implications

The comparison between the three configurations highlights the trade off between the level of data protection and the computational cost introduced by each approach. The experimental results presented in the previous sections allow us to evaluate not only the differences from the measured times, but also to understand how the adopted protection mechanism changes the processing and communication model of the developed architecture.

The main differences between the three approaches are summarized in Tab. 5.7:

Trade-off Dimension	**Plaintext**	**TLS**	**CKKS**
Protection during transmission	None	Yes, confidentiality provided by TLS	Yes, data are encrypted before transmission
Protection during Cloud computation	None	No, data must be available in plaintext to the Server application	Yes, the outsourced operation is performed directly on ciphertexts
Cryptographic workload on the UAV	None	Limited to the communication security mechanisms	Higher due to CKKS initialization, key management, encryption, and decryption
Remote computation	Plaintext arithmetic	Plaintext arithmetic after TLS termination	Homomorphic element-wise squaring through `EvalMult`
Data representation during transmission	Unencrypted application data	TLS-protected application data	Serialized CKKS ciphertext encoded in Base64

Table 5.7. Qualitative comparison of the three protection configurations.

Another aspect we have to take into account is the size of the transmitted data. In the Plaintext and TLS configurations, the difference vector is transmitted in its numerical representation, while in the CKKS configuration the same information is represented as a larger ciphertext. Then, as

already known, the OpenFHE ciphertext has to be serialized and encrypted before transmission. This contributes to the additional communication overhead introduced by the encrypted workflow.

Applicability and Deployment Scenarios

The architecture we have developed may be suitable for applications where the confidentiality of the processed data is more important than obtaining an immediate or real time response. In particular, it can be useful in scenarios where UAVs collect sensitive information that should not be directly exposed to the remote Cloud infrastructure. Possible application scenarios can include:

- **Military and Border Surveillance:** UAVs can be used to monitor military areas, borders, restricted zone or other sensitive regions in order to detect vehicles, personnel or unusual activities. In these scenarios, the acquired images and extracted features may contain confidential information about targets, positions or monitored areas.
- **Critical Infrastructure Monitoring:** the same approach can be applied to the surveillance of strategic facilities such as power plants, communication infrastructures, military bases, airports or other critical sites. In this case, the images may reveal structural details, access points or other information that should remain confidential.
- **Environmental and Wildlife Monitoring:** UAVs can be used to monitor protected natural areas, wildlife populations or endangered species. In these applications, images and extracted information may reveal sensitive locations, such as nesting areas or the position of protected animals.

In all of the previous scenario, processing part of the data in encrypted form can reduce the amount of sensitive information directly accessible to the remote Server.

However, the current prototype is less suitable for applications that require very short response times or a very high processing rate. In particular applications like:

- **Real Time Autonomous Navigation:** in these applications, the UAV has to react immediately to what happens around it. For example, obstacle avoidance and flight control require decisions to be made in a very short time in order to keep the drone stable and avoid collisions. Since the current CKKS implementation introduces several seconds of processing time, it is not suitable for these time critical operations.
- **High Throughput Image Processing:** some applications require the UAV to process many images continuously and at a high rate. In this case, the additional time required for encryption, decryption, and homomorphic operations would slow down the processing pipeline. For this reason, further optimization or dedicated hardware would be needed to handle a large number of images efficiently.
- **Applications with Limited Confidentiality Requirements:** in some applications, the processed data are not highly sensitive and do not require protection during Cloud computation. In these cases, the additional complexity and execution time introduced by CKKS may not be necessary, and a simpler solution such as TLS protected communication may be more appropriate.

These observations show that the field of use of the proposed architecture depends on the requirements of the specific application, in particular on the balance between confidentiality, latency, and processing throughput.

Overall, the experimental results show that CKKS extends confidentiality to the outsourced computation, but introduces a significantly higher computational cost than Plaintext and TLS. For this reason, the current prototype is more suitable for applications where protecting sensitive data during remote processing is more important than achieving very low latency.

In conclusion, the work of thesis we have developed, shows that Fully Homomorphic Encryption can be effectively integrated into a Drone-Cloud architecture to protect sensitive data not only during transmission, but also while they are processed by the Cloud. The developed architecture demonstrates the feasibility of this approach and highlights both its advantages and its current limitations. Although the computational overhead of CKKS is still significant, the results we have

obtained confirm that privacy-preserving Cloud processing is a realistic solution for applications where data confidentiality is a necessary. Future improvements in th selection of cryptographic parameter, software optimization, and hardware support could further reduce the execution time and make this type of architecture more suitable for real UAV deployments.

Chapter 6

Conclusions and Future Developments

6.1 Goals Achieved

The main goal we want to achieve with our work of thesis was to design, implement, and experimentally evaluate a Drone-Cloud architecture able to perform the tank matching procedure while keeping the outsourced data encrypted. The purpose of our work was not to develop the most accurate algorithm used for target recognition, but to understand if FHE could be integrated into a realistic processing pipeline and used to protect data during remote computation.

From the implementation and the different experimental scenarios, several important results can be highlighted:

Protection of data during outsourced computation

The comparison between plaintext processing, TLS, and CKKS showed the different levels of protection offered by the three different approaches. With plaintext processing the Cloud directly accesses the data. TLS protects the data while they are transmitted, but the server still needs to recover them in plaintext before being processed. With CKKS, instead, the selected computation is performed directly on encrypted data. This minimizes the trust we need in the Cloud, according to the Zero Trust architecture paradigm, and aligning with the honest-but-curious model adopted in this thesis.

Integration of Fourier-Mellin processing with CKKS

Another result of the project was the integration between the Fourier-Mellin transform and the CKKS encryption scheme. In fact, once signals are processed, the spectral representation is normalized and converted into a numerical vector that can be packed into CKKS slots, making it suitable for homomorphic computations. After homomorphic computations performed by the Cloud, values are returned back to the drone where locally is taken the final decision of matching, after decryption. This keeps the encrypted computation simple and avoids the need for implementing bootstrapping mechanism.

Preservation of the matching decision under encrypted processing

The experimental results showed that the CKKS workflow preserves the expected decision in most of the considered scenarios. The nominal and urban cases are correctly identified, while the desert scene is correctly rejected. The geometric variation scenario represents the only exception, since the rotated and scaled target produces a false negative in the CKKS configuration. The same scenario is instead correctly identified with Plaintext and TLS using the same tolerance threshold. This behavior is discussed in detail in Subsection 5.3.4, where the numerical differences introduced by the encrypted processing path are analyzed, although their exact contribution to the false negative could not be isolated. Overall, the results show that the encrypted workflow can

preserve the recognition decision in several test conditions, while the geometric variation remains the most critical case.

Identification of the main limitations of the prototype

The experimental evaluation also highlighted some practical limitations of the current system. First, the Fourier-Mellin representation is influenced by the surrounding background, as observed in the urban scenario, where buildings, shadows, and other visual elements contribute to the global spectral representation. This can reduce the separation between matching and non-matching cases. A second limitation concerns performance, since the CKKS workflow introduces a considerably higher computational cost than Plaintext and TLS due to encryption, decryption, ciphertext management, and homomorphic processing. Moreover, the experiments were performed in a Docker-based environment rather than on a real UAV platform. Therefore, further optimization and tests on resource-constrained hardware would be required before considering the prototype suitable for applications with strict latency and computational requirements.

The performance measurements also quantified the cost of this additional confidentiality: while Plaintext and TLS required comparison times on the order of tenths of a second in the tested environment, the CKKS configuration required several seconds, confirming that computational overhead remains the main limitation of the current prototype.

Overall, we were able to achieve the desired goal in fact with our architecture we have demonstrated that a matching operation can be outsourced to an untrusted Cloud and executed directly on encrypted data using CKKS. The Cloud can contribute to the computation without receiving the UAV secret key or accessing content in plaintext, but still letting to the drone the final decryption and recognition decision.

At the same time, this means that there are some problems we have to address: the computational overhead introduced by homomorphic encryption and the limitations of the current Fourier-Mellin matching approach when significant geometric changes or complex backgrounds are present. These aspects represent the starting point for the future improvements discussed in the next section.

6.2 Future Developments

The analysis and experimental results obtained highlighted both the potential of the proposed architecture and the aspects that need to be improved. Starting from the limitations observed during the tests, future work can mainly focus on three directions:

- **Better accuracy for recognition:** making the matching procedure more robust against geometric variations, complex backgrounds, and different acquisition conditions.
- **Faster computing performance:** optimizing the CKKS configuration and the implementation in order to decrease encryption, decryption, and homomorphic processing times.
- **Extending the architecture:** moving from the current proof-of-concept toward more complete scenarios involving multiple targets, multiple UAVs, real communication networks, and more advanced encrypted computations.

6.2.1 Better accuracy for recognition

One of the main limitations observed is the robustness of the matching procedure. In the implemented architecture we have used the Fourier-Mellin trasform together with a fixed decision threshold. Although this approach was sufficient to validate the encrypted workflow, the tests showed that geometric changes and complex backgrounds can modify the spectral representation and affect the final decision.

For allowing a more robust recognition processing, several improvements could be considered:

- **Expanding the reference set:** instead of comparing the captured scenario with just the reference template, we could built a larger reference set with different views of the same target, for example at different orientations, scales, or acquisition conditions. In this way the drone could then compare the observed scene with multiple reference descriptors and

select the closest result. This would reduce the dependence on a single ideal representation of the target.

- **Using multiple image channels:** The current implementation extracts Fourier-Mellin features from the RGB channels, but we use only the red channel representation for the homomorphic computations and matching. A future implementation could combine the information from all three channels or evaluate alternative representations such as grayscale or luminance for increasing the amount of information used for the final decision.

- **Adaptive decision threshold using Machine Learning:** The current implementation uses a fixed tolerance threshold to separate matching and non matching cases. A more systematic approach would be to determine this value from a larger set of scenarios. Statistical analysis could then be used to select a threshold that provides a better balance between false positives and false negatives.

- **Replacing global spectral features with learned embeddings:** A possible future improvement could be replacing the Fourier-Mellin computation with features extracted by a Convolutional Neural Network (CNN). So, instead of describing the whole image through a global spectral transform, the CNN would convert each image into a compact numerical vector. Then the network could be trained so that images containing the same target produce similar vectors, while different targets produce more distant vectors. This would make the matching process more robust when the background changes. These vectors would still be compatible with the CKKS based architecture, because they have a fixed numerical size and can be packed into ciphertext slots. The Cloud could then compare the encrypted vectors without accessing their plaintext content. In this way, the recognition stage could be improved without changing the general Edge-to-Cloud structure.

6.2.2 Improving Computational Performance

The experimental results showed that the main performance limitation are related to the computational overhead introduced by the cryptographic operations. In fact, CKKS initialization, encryption, decryption, and encrypted evaluation require significantly more resources.

Several improvements may be:

- **Optimization of CKKS parameters:** The parameters used in the current implementation were selected conservatively in order to obtain a stable experimental configuration. However, the homomorphic matching circuit requires only a limited multiplicative depth. A future implementation could investigate a more efficient combination of ring dimension, multiplicative depth, scaling modulus, and batch size while maintaining the required numerical precision and security level. Reducing large parameters could decrease memory consumption and execution time, in fact the current configuration was designed primarily for functional validation rather than as a production-ready cryptographic setup. Since the OpenFHE security level was set to `HEStd_NotSet`, future deployments should select and validate the cryptographic parameters against an appropriate standardized security target.

- **Hardware acceleration for homomorphic operations:** We can move the most demanding CKKS operations such as encryption and decryption to GPUs, FPGAs, or dedicated cryptographic accelerators. These platforms are optimal for highly parallel polynomial and Number Theoretic Transform operations used by modern FHE libraries.

- **Parallel processing and request batching on the Cloud:** While the CKKS scheme allows us to process data in parallel within a single ciphertext using SIMD, we can push system performance by processing multiple independent tasks at the same time. In this way we can distribute separate ciphertexts across multiple Cloud worker nodes or hardware accelerators. While this does not reduce individual latency, it increases overall throughput, optimal for larger and real world workloads.

- **Scheme switching for more complex encrypted operations:** If in the future we move more task about decision to the Cloud, we will hit a limit with CKKS because it struggles with conditional checks, like comparisons and thresholding. To solve this, we could combine

CKKS with Boolean based encryption schemes like TFHE or FHEW using scheme switching techniques. While switching between these schemes adds extra processing overhead, it would allow us to run complex decision logic directly on encrypted data without having to send intermediate results back to the drone.

6.2.3 Expanding the system

The current implementation was designed for working with one drone, one reference target and one Cloud server. A future version could extend the same architecture to more realistic and complex operational scenarios:

- **Homomorphic inference for Machine Learning models:** In the current architecture, the Cloud performs only the homomorphic squaring of the encrypted difference vector. A possible future development would be to move toward homomorphic Machine Learning inference, where a trained model is adapted to operate directly on encrypted data. In this case the Cloud could use the encrypted image features to classify the observed object. For example, the system may be able to distinguish between different classes of vehicles rather than only deciding whether a specific target is present or not. The main advantage is that the Cloud could perform this more complex analysis without accessing the plaintext features. However, the Machine Learning model would need to be designed or adapted to use operations that can be efficiently evaluated with homomorphic encryption.

- **Multi-UAV and multi-user architectures:** A future version of the system could involve several UAVs connected to the same Cloud infrastructure. In this way each drone could collect its own data and use its own cryptographic keys, while still contributing to a common analysis. Solutions such as Multi-Key FHE or Threshold FHE could be investigated to allow encrypted data coming from different users or devices to be processed together without working with drones that share the same secret key.

- **Verification of Cloud computation:** Here we worked with the Cloud considered to be honest-but-curious. This means that the server is expected to perform the requested computation correctly, even if it is not trusted with the plaintext data. A future system could also consider the possibility that the Cloud returns an incorrect or modified result. In this case, additional verifiable-computation mechanisms could be investigated to allow the UAV to verify that the Cloud executed the requested operation correctly without revealing the plaintext data to the Cloud and without having to repeat the entire outsourced computation locally.

- **Deployment on real UAV hardware and communication networks:** Here we use the Docker enviroment to test our different scenarios. A future step could be to run the Client on hardware that is closer to a real drone platform and connect it to a physically remote Cloud server. Tests could then be performed over Wi-Fi, cellular or radio transmissions. This would make it possible to study the real impact of network delay, bandwidth limitations, packet loss, energy consumption, and the limited computing resources available on board.

- **Combining homomorphic encryption with authenticated communication:** CKKS protects the confidentiality of the data while they are processed, but it does not provide all the security functions required for network communication. For this reason, a practical system could combine CKKS with TLS or another authenticated communication protocol. CKKS would keep the application data encrypted during Cloud processing, while TLS would protect the communication channel by providing authentication, integrity, and secure transport between the UAV and the server.

6.2.4 Final Considerations

The future developments discussed in this chapter can be grouped into three main areas: improving recognition robustness, reducing the computational overhead of homomorphic processing, and extending the prototype toward more realistic deployment and security scenarios. The tests showed that geometric variations and complex backgrounds can affect the spectral representation

and the final matching score. In particular, the geometric variation scenario was still correctly identified with plaintext and TLS, while it produced a false negative in the CKKS configuration. This result suggests that future improvements should consider a more robust feature representation, a deeper analysis of the numerical behavior of the encrypted processing path, or more advanced feature extraction methods based on Machine Learning.

The second limitation concerns the computational cost of CKKS. Homomorphic encryption provides an important advantage in terms of confidentiality, but encryption and decryption require more resources than conventional plaintext processing. A way we have for reducing this overhead is to perform better parameter selection, software optimization, parallel processing, and dedicated hardware accelerators.

Another important step we could perform is testing the architecture outside the current Docker environment. Running instead the Client on real embedded hardware and communicating with a physically remote Cloud server may provide a more realistic evaluation of network latency, bandwidth limitations, and energy consumption. The architecture could also be extended to multiple UAVs and to stronger security models in which the drone is able to verify if the Cloud has performed in the correct way the requested computation.

For all these reasons, the system developed in this thesis should be considered a starting point for further development. The results demonstrate not only that an encrypted matching process can be integrated into a Drone-Cloud architecture, but also what may be the aspects to be improved before using the system in more complex and demanding scenarios.

Bibliography

[1] C. Gentry, "A fully homomorphic encryption scheme", *Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.* https://crypto.stanford.edu/craig/craig-thesis.pdf

[2] Imperial War Museums, "A brief history of drones," *IWM*, [Online]. Available: https://www.iwm.org.uk/history/contemporary-conflict/a-brief-history-of-drones

[3] A. G. Pearson, "The secret history of drones," *Air & Space Quarterly*, vol. 1, no. 12, https://airandspace.si.edu/air-and-space-quarterly/issue-12/secret-history-of-drones

[4] V. Melnyk, "Edge + Cloud Inference for Drone-in-a-Box: The Split That Fits Every Need", *Dronehub Blog*. https://dronehub.ai/blog/edge-cloud-inference-drone-in-a-box

[5] MITRE, "Memory Scraping", *MITRE FiGHT*, FGT5005. https://fight.mitre.org/techniques/FGT5005/

[6] Microsoft Security, "What is Zero Trust Architecture?", *Microsoft Security 101*. https://www.microsoft.com/it-it/security/business/security-101/what-is-zero-trust-architecture

[7] IBM Think, "What is Encryption?", *IBM Think Topics*. https://www.ibm.com/think/topics/encryption

[8] National Institute of Standards and Technology, "Integrity", in *NIST Computer Security Resource Center Glossary*, 2025. Available: https://csrc.nist.gov/glossary/term/integrity

[9] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008. DOI 10.17487/RFC5280

[10] A. Z. Lunkes and F. Borges, "The application of homomorphism in cryptography", *Trends in Computational and Applied Mathematics*, vol. 25, p. e01772, 2024. DOI 10.5540/tcam.2024.025.e01772

[11] M. Mohan, M. K. Kavitha Devi, and V. Jeevan Prakash, "Homomorphic Encryption-State of the Art", *2017 International Conference on Intelligent Computing and Control (I2C2)*, Coimbatore (India), June 23-24, 2017, pp. 1–6. DOI 10.1109/I2C2.2017.8321774

[12] A. A. El-Latif, B. Abd-El-Atty, I. Mehmood, *et al.*, "Survey on Fully Homomorphic Encryption, Theory, and Applications", *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1572–1609, Oct. 2022. DOI 10.1109/JPROC.2022.3199515

[13] H. Chen, S. Han, I. Iliashenko, J. H. Park, and A. Sabelfeld, "Title of the ePrint Report 2025/1684", *IACR Cryptology ePrint Archive*, Report 2025/1684, 2025. https://eprint.iacr.org/2025/1684

[14] Zama Team, "Zama: Open-Source Homomorphic Encryption Solutions", *Official Website*, 2026. https://www.zama.ai

[15] P. Grontas, A. Pagourtzis, and M. Spyrakou, "Voting with Coercion Resistance and Everlasting Privacy Using Linkable Ring Signatures", *IACR Cryptology ePrint Archive*, Report 2025/1775, 2025. https://eprint.iacr.org/2025/1775

[16] S. Halevi and V. Shoup, "Design and Implementation of a Homomorphic-Encryption Library", *IBM Research*, 2014. https://github.com/homenc/HElib

[17] M. A. HomomorphicEncryption.org, "Microsoft SEAL (release 4.4)", *Microsoft Research*, 2026. https://github.com/microsoft/SEAL

[18] A. Polyakov, C. Peikert, et al., "PALISADE: A General Purpose Lattice Cryptography Library", 2019. https://gitlab.com/palisade/palisade-release

[19] Lattigo v6, "Lattigo: A Library for Lattice-Based Multiparty Homomorphic Encryption in Go", 2024. https://github.com/tuneinsight/lattigo

[20] L. Ducas and D. Micciancio, "FHEW: A Fully Homomorphic Encryption Library", *GitHub repository, version 2.0-beta*, 2017. https://github.com/lducas/FHEW

[21] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second", *EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia (Bulgaria), Apr. 26-30, 2015, pp. 617-640. DOI 10.1007/978-3-662-46800-5_24

[22] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption over the Torus", *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020. DOI 10.1007/s00145-019-09319-x

[23] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Library over the Torus", *GitHub repository, version 1.1*, 2020. https://github.com/tfhe/tfhe

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

[25] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "HEAAN: Homomorphic Encryption Library", GitHub repository, version 2.1. https://github.com/wangqi01/HEAAN

[26] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers", *ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, Hong Kong (China), Dec. 3-7, 2017, pp. 409-437. DOI 10.1007/978-3-319-70694-8_15

[27] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption", *SAC 2018: 25th International Conference on Selected Areas in Cryptography*, Calgary, AB (Canada), Aug. 15-17, 2018, pp. 347-368, Springer, 2019. DOI 10.1007/978-3-030-10970-7_16

[28] K. Han, "FullRNS-HEAAN", GitHub repository, 2018. https://github.com/KyoohyungHan/FullRNS-HEAAN

[29] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption", *Cryptology ePrint Archive, Report 2012/144*, 2012. https://eprint.iacr.org/2012/144

[30] CryptoExperts, "FV-NFLlib: Library Implementing the Fan-Vercauteren Homomorphic Encryption Scheme", *GitHub repository*, 2016. https://github.com/CryptoExperts/FV-NFLlib

[31] VernamLab, "cuFHE: CUDA-accelerated Fully Homomorphic Encryption Library", *GitHub repository, version 1.0-beta*, 2018. https://github.com/vernamlab/cuFHE

[32] NuCypher, "NuFHE: A GPU-Powered Torus Fully Homomorphic Encryption Library", *GitHub repository*, 2023. https://github.com/nucypher/nufhe

[33] A. Polyakov et al., "OpenFHE: Open-Source Fully Homomorphic Encryption Library", *GitHub repository* 2026. https://github.com/openfheorg/openfhe-development

[34] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption", *ICLR 2021 Workshop on Distributed and Private Machine Learning*, Virtual Event, May 7, 2021. DOI 10.48550/arXiv.2104.03152

[35] OpenMined, "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption", *GitHub repository*, 2026. https://github.com/OpenMined/TenSEAL

[36] Zama, "Concrete Documentation: Get Started with TFHE", *Official Technical Documentation*, 2026. https://docs.zama.org/concrete/get-started/get-started

[37] Zama, "Concrete-ML: Privacy-Preserving Machine Learning with Fully Homomorphic Encryption", *GitHub repository*, 2022. https://github.com/zama-ai/concrete-ml

[38] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data", *GitHub repository*, 2022. https://github.com/zama-ai/tfhe-rs

[39] Zama, "FHEVM: A Full-Stack Framework for Integrating Fully Homomorphic Encryption (FHE) with Blockchain Applications", *GitHub repository*, 2025. https://github.com/zama-ai/fhevm

[40] OpenFHE Development Team, "OpenFHE Documentation", *Official Technical Documentation*, 2026. https://openfhe-development.readthedocs.io/en/latest/index.html

[41] C. Peikert et al., "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY", *GitHub repository*. https://github.com/cpeikert/ALCHEMY

[42] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY", *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, Toronto, ON (Canada), Oct. 15-19, 2018, pp. 1020-1037. DOI 10.1145/3243734.3243828

[43] CEA-LIST, "Cingulata: Compiler Toolchain and RTE for Fully Homomorphic Encryption", *GitHub repository*. https://github.com/CEA-LIST/Cingulata

[44] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A Compilation Chain for Privacy Preserving Applications", *3rd International Workshop on Security in Cloud Computing (SCC 2015)*, Singapore (Singapore), Apr. 14, 2015, pp. 13-19. DOI 10.1145/2732516.2732520

[45] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: Encrypt-Everything-Everywhere", *GitHub repository*. https://github.com/momalab/e3

[46] Alan Turing Institute, "SHEEP: Homomorphic Encryption Evaluation Platform", *GitHub repository*. https://github.com/alan-turing-institute/SHEEP

[47] Microsoft Research, "EVA: Compiler for Microsoft SEAL", *GitHub repository*. https://github.com/microsoft/EVA

[48] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation", *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, London (United Kingdom), June 15-20, 2020, pp. 546-561. DOI 10.1145/3385412.3386023

[49] MarbleHE, "ABC: FHE Compiler Prototype", *GitHub repository*. https://github.com/MarbleHE/ABC

[50] A. Viand and H. Shafagh, "Marble: Making Fully Homomorphic Encryption Accessible to All", *6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC 2018)*, Toronto, ON (Canada), Oct. 19, 2018, pp. 49–60. DOI 10.1145/3267973.3267978

[51] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: Fully Homomorphic Encryption Compiler", *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA (USA), Aug. 9–11, 2023, pp. 4715–4732. https://www.usenix.org/conference/usenixsecurity23/presentation/viand

[52] MarbleHE, "HECO: Optimizing Compiler for Fully Homomorphic Encryption (FHE)", *GitHub repository*. https://github.com/MarbleHE/HECO

[53] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. J. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A Programmer-Friendly System for Building Homomorphic Encryption Applications", *7th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC 2019)*, London (United Kingdom), Nov. 11, 2019, pp. 57–68. DOI 10.1145/3338469.3358945

[54] Google, "Fully Homomorphic Encryption Transpiler", *GitHub repository*. https://github.com/google/fully-homomorphic-encryption/tree/main/transpiler

[55] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing", *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, Phoenix, AZ (USA), June 22-26, 2019, pp. 142-156. DOI 10.1145/3314221.3314628

[56] Intel, "HE Transformer for nGraph", *GitHub repository*. https://github.com/IntelAI/he-transformer

[57] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data", *16th ACM International Conference on Computing Frontiers (CF 2019)*, Alghero (Italy), Apr. 30-May 2, 2019, pp. 3-13. DOI 10.1145/3310273.3323047

[58] T. van Elsloo, G. Patrini, and H. Ivey-Law, "SEALion: A Framework for Neural Network Inference on Encrypted Data", *arXiv preprint arXiv:1904.12840* 2019. DOI 10.48550/arXiv.1904.12840

[59] Sunscreen, "Sunscreen: A Compiler for Fully Homomorphic Encryption and Zero Knowledge Proofs", *GitHub repository*. https://github.com/Sunscreen-tech/Sunscreen

[60] Google, "HEIR: Homomorphic Encryption Intermediate Representation", *GitHub repository*. https://github.com/google/heir

[61] Fhelipe, "Fhelipe: An Easy-to-Use Tensor Compiler for FHE", *GitHub repository*. https://github.com/fhelipe-compiler/fhelipe

[62] A. Feldmann, C. Golden, F. Elsabbagh, H. R. Lee, A. Krastev, V. Vaikuntanathan, and others, "A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption", *Proceedings of the ACM on Programming Languages*, Vol. 8, No. OOPSLA2, 2024. DOI 10.1145/3656382

[63] Modern Compilers Lab, "CHEHAB: A Fully Homomorphic Encryption Compiler", *GitHub repository*. https://github.com/Modern-Compilers-Lab/CHEHAB

[64] A. Seddiki, M. Arab, Z. Hebbal, A. Chabounia, E. Chielle, K. Benatchba, Y. Challal, D. E. Menacer, M. Maniatakos, and R. Baghdadi, "CHEHAB: Automatic Compiler Code Optimization for Fully Homomorphic Encryption", *35th ACM SIGPLAN International Conference on Compiler Construction (CC 2026)*, Sydney, NSW (Australia), Jan. 31-Feb. 1, 2026, pp. 81-93. DOI 10.1145/3771775.3786269

[65] Jelle Vos, "Oraqle: A Depth-Aware Secure Computation Compiler", *GitHub repository*. https://github.com/jellevos/oraqle

[66] J. Vos, M. Conti, and Z. Erkin, "Oraqle: A Depth-Aware Secure Computation Compiler", *12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC 2024)*, Salt Lake City, UT (USA), Oct. 14, 2024, pp. 43-50. DOI 10.1145/3689945.3694808

[67] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU", *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Montreal, QC (Canada), Feb. 25-Mar. 1, 2023, pp. 922-934. DOI 10.1109/HPCA56546.2023.10071017

[68] Duality Technologies, "Hardware Acceleration of Fully Homomorphic Encryption: Making Privacy-Preserving Machine Learning Practical", *Duality Tech Blog*, 2026. https://dualitytech.com/blog/hardware-acceleration-of-fully-homomorphic-encryption-making-privacy-preserving-machine-learning-practical/

[69] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme", *EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Tallinn (Estonia), May 15-19, 2011, pp. 129-148. DOI 10.1007/978-3-642-20465-4_9

[70] Optalysys, "Optalysys Resources: Silicon Photonics and Hardware Acceleration for Fully Homomorphic Encryption", *Official Technical Resources and Articles*, 2026. https://optalysys.com/resources/

[71] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers", *IACR Cryptology ePrint Archive*, Report 2016/421, 2016. https://eprint.iacr.org/2016/421

[72] Ultralytics, "What is the Scale-Invariant Feature Transform (SIFT)?", *Ultralytics Blog*, 2026. https://www.ultralytics.com/it/blog/what-is-the-scale-invariant-feature-transform-sift

[73] I. Nawar, "ORB Feature Descriptor Implementation", *GitHub Repository*, 2026. https://github.com/ImranNawar/orb_feature_descriptor

[74] J. García, Ed., "Encyclopedia of Electrical and Electronic Power Engineering", in *Encyclopedia of Electrical and Electronic Power Engineering*, Elsevier, 2023, ch. 00131-8, pp. 1–10. DOI 10.1016/B978-0-12-821204-2.00131-8

[75] S. Shank, "The Fast Fourier Transform: How and why it works", *IBM Research Blog*, Aug. 2024. [Online]. Available: https://research.ibm.com/blog/the-fast-fourier-transform-how-and-why-it-works

[76] S. Thoduka, "Fourier-Mellin Transform", *Image Registration using Fourier-Mellin Transform Documentation*. https://sthoduka.github.io/imreg_fmt/docs/fourier-mellin-transform/

[77] S. Thoduka, "Log-Polar Transform", *Image Registration using Fourier-Mellin Transform Documentation*. https://sthoduka.github.io/imreg_fmt/docs/log-polar-transform/

[78] IBM Think, "What is Transport Layer Security (TLS)?", *IBM Think Topics*. https://www.ibm.com/think/topics/transport-layer-security

[79] S. Rose, O. Borchert, S. Mitchell and S. Connelly, "Zero Trust Architecture", *NIST Special Publication 800-207*, Aug. 2020. DOI 10.6028/NIST.SP.800-207