UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA MAGISTRALE

Data aggregation using Homomorphic Encryption in Mobile CrowdSensing context

Relatori:

Prof. Stefano Chessa

Dott. Michele Girolami

Candidato:

Chiara Boni

ANNO ACCADEMICO 2021/2022

Contents

List of Figures

Listings

Chapter 1

INTRODUCTION

1.1 Mobile CrowdSensing

The number of active, user-owned devices is growing by unprecedented amounts, generating an ever-expanding stream of information and data. The organizational machine of the cloud providing infrastructure has to face the management challenge that this considerable flow brings with it. It is upon this starting point that the *Mobile CrowdSensing* (MCS) is based, thus a study that relies on a network of mobile sensors, which are able to collect information on the environment. The core and common idea is relying on smartphones to sense the pulse of smart cities through the development of apps and the involvement of several users in large living labs typically lasting several months [15]. Data collection is made possible through dedicated requests, called *tasks*. A crucial issue of the MCS paradigm is the definition of strategies for the recruitment of volunteers and for the scheduling of the tasks assigned to the end-users [15]. Since the intelligence and mobility of humans can be leveraged to help applications collect higher quality or semantically complex data, but devices may incur energy and monetary costs [11], it is necessary to succeed in this user engagement campaign.
Depending on the type of project being developed, the tasks required to users differ in their nature: they are defined as passive if they do not need to execute an action, intended as active participation in sampling, but rather defined automatically by the application running on the device. Evidence of this can be an MCS application that depends upon Bluetooth activation to study proximity between appliances, but once this is put into action, it requires no further operation. The opposite case, an active task, on the other hand, expects periodic attention from the user, since he is called to perform a certain action himself. A monitoring application

on the urban planning of a city is one of the most common cases for this type of sampling: in fact, users can be, to cite an example, also asked to take photographs of the landscape to demonstrate the study.

The continuous execution of tasks, regardless of their nature, contributes to what is called user profiling: succeeding in the reconstruction of habits and movements, thanks to the analysis of data collected during a certain period of time. The information generated by a single participant is then combined and aggregated with similar information from all other components of the study, to provide a sampled response to the questions initially posed by the project. The idea of MCS remains to have an inquiry into a given question, but not placed on the individual, rather shifted to what the community is and how it acts.
Through participant profiling, the provider and the support architectures it uses are able to build datasets that can be analyzed. Containing real data related to the habits or, sometimes, even closely correspondent to the clinical picture of participants, this information is highly sensitive and therefore it is necessary to strengthen every possible security aspect. If openly documented and user-friendly, data processing guidelines can be used as an incentive to engage even the most reluctant users. The architecture is complex and spread over several layers, but these details are and should be completely hidden from the end user, who is not interested in the low-level practices of information manipulation, but rather focused on how to choose to participate in a particular study: proper security management may soon become the deciding factor in this choice.
As detailed in [30], there are some properties considered fundamental for a MCS architecture, which cannot be violated in terms of security. Since this scheme is also able to provide a product, in the form of service offered to components, of data storage and analysis, it is required to submit to what are the basic principles in the treatment of sensitive information. The main aspect that was chosen to be addressed is confidentiality: building a model that guarantees its users total secrecy of information, both individually and aggregated. This control must start from the base of the architecture, thus covering the phase of sending data over the network, which must not be visible from the outside, to the point of guaranteeing its safety when it is saved on the platform. A further concern that arises with MCS, is the possibility of information coming from the aggregation of data: as they are collected from individual participants, but then these are modelled and considered as a whole, a malicious

user may be able to trace the identities of volunteers from the aggregated information. Users who participate in the MCS study must be able to rely on the fact that their identity cannot be traced, despite the amount of information they provide to the platform.

1.2 Methodology

One of the means that infrastructures can take advantage of, in order to achieve higher security, is to rely on cryptography. This practice increases the level of users' confidence in platforms, but it cannot be managed as an isolated technique: it must be combined with a broader context of continuous maintenance, redundancy and data isolation; procedures that are capable of requiring both software and hardware interventions.
Cryptography is the practice and study of hiding information. A cryptographic algorithm works in combination with a key —a word, number, or phrase— to encrypt the plaintext. The security of encrypted data is entirely dependent on two things: the strength of the cryptographic algorithm and the secrecy of the key [4].
Depending on how the communications and exchanges occur between the parties involved, cryptography is defined as symmetric if it is based on the same key, used in both the encryption and decryption phases. The use of two distinct keys, instead, is found in an asymmetric encryption scheme: in this precise case the text is encrypted with a key, independent from any other detail, but decrypted with a different one, private and associated to each subject involved. It is therefore a more complex procedure, slower and with extensive mathematical processes, but an exchange of keys is avoided during the initial phase of communication, subject to possible violation of external parties, but necessary to agree on the use of the same key required by the symmetric scheme.[4]

The proposed thesis aims to study and apply a type of encryption defined as homomorphic such that, in addition to the intrinsic positive characteristics of a cryptographic scheme, allows to increase the security of communications as it is able to perform arithmetic operations, such as sums or multiplications, directly on the encrypted data, without the need to decrypt them. The result of an operation between ciphers, once decrypted, will be the same as it would have been if the data had been decrypted previously. In this way, an external party, an aggregator, has the opportunity to see only the encrypted data within the communication and, not having the need to modify them, they move in the architecture without removing the protection given by the encryption.

Homomorphic encryption finds easy adaptation in a decentralized and multi-tier model like MCS, as data move from the sensors placed in the field, which can be the same smartphones provided to users, to countless network nodes and then to a specific cloud platform. Depending on the mobility study carried out by the latter, the data may or may not need to be processed, aggregated, and with these schemes these operations are secure: the encryption will only be removed by the last component involved, i.e. the provider itself.

1.3 Thesis Objectives

The application scenario of interest, based on an MCS architecture, concerns the manipulation and homomorphic aggregation of GPS coordinates. This study was conducted using the GeoLife dataset [13], which has been organized by Microsoft Research Asia with about 182 users in very long period, ranging from April 2007 to August 2012. In the interest of the following thesis, such data has been pre-processed so that, taken a set of points that form the trajectory of a path for a user, it was possible to calculate the distance traveled. These values will be combined to calculate the total space traversed.

The architecture of the model is composed as follows: at the base there are the sensors, smartphones, which are simulated by the dataset made available; an aggregator, as a subject external to both the sensor network and the cloud platform, with the task of receiving the flow of packets and aggregate them using homomorphic encryption and, lastly, the provider node, which will be in charge of storing and analyzing the data. Homomorphic aggregation was made possible through the use of PALISADE [22], which is an open-source library that provides efficient implementation of lattice cryptographic and homomorphic schemes, such that Brakerski-Gentry-Vaikuntanathan (BGV)[5] scheme for integer arithmetic and Cheon-Kim-Kim-Song (CKKS)[7] scheme for real-number approximate arithmetic.

An additional innovative aspect was to employ the Redundant Residue Number System (RRNS) [21] to achieve more robustness in data submission. With this model, redundancy is provided to the sampled data by encoding more residues than are necessary for representation: if at least those defined as legitimate can be guaranteed to be received, the aggregation operation is proven to be successful. In a decentralized MCS environment, a type of assurance such as the above lays the foundation for more robust architectures, with the ability to not only detect issues in the data, but also be able to restore it in the event of a loss, without having to resort to re-transmission. In [19] the authors report the work done over IoT sensors in order to create an efficient transmission scheme, based on the RRNS system. They primarily focus their research on techniques for discovering errors in data packets and then subsequent recovery, including an analysis on how having redundant modules can save energy to the sensors, decreasing the number of re-transmissions they have to perform and also save them network bandwith.

In this development context, this thesis project aims to:

- Study a Mobile CrowdSensing model, focusing on the privacy-preserving issues it presents for users participating in it.
- Investigate homomorphic encryption into the MCS model, to ensure secure data aggregations.
- Simulate such research in a concrete scenario by analyzing the PALISADE library.
- Strengthen the data aggregation model by incorporating redundancy, when sending data, using the RRNS methodology.

The reported results refer to:

- The adaptation of homomorphic encryption in a real MCS context and its limits.
- A performance analysis of the use of the PALISADE library.
- Redundancy management via the RRNS system and consequent adaptation for the concrete application scenario.

The work done is reported below, broken down as follows: Chapter 2 provides a general analysis of the state of the art regarding homomorphic encryption and RRNS. It is also included an overview of the PALISADE library, illustrating its key points and its advantages useful for the project. In Chapter 3, the model of the studied system is presented, both from a design and a logical point of view. Lastly, in Chapter 4, the experiments performed on a real architecture are introduced, in order to measure the performances of the designed model.

Chapter 2

STATE OF THE ART AND BACKGROUND

This chapter aims to present the main technologies that have been used, in order to give a clear picture of the state of the art at the time of the development of this project.
Initially the Residue Number System is summarized, underlining the advantages and the structure that led to the choice to introduce it into the study; then the concept of homomorphic encryption is elaborated as a central element of interest and, lastly, it is given an overview of the use of the chosen library, which is PALISADE [23].

2.1 Residue Number System

The Residue Number System (RNS) is a data representation system that allows interpreting an integer number as a set of smaller values, with respect to a given base. A RNS base is defined by a set of relatively prime integers $m_1, m_2, ..., m_r$.
The range of representation of the system is given by the product of all the moduli:

$$M = \prod_i m_i \tag{2.1}$$

Any integer $X \in [0, M-1]$ has unique RNS representation given by:

$$X = (\langle X \rangle_{m_1}, \langle X \rangle_{m_2}, ..., \langle X \rangle_{m_r})$$

where each $\langle X \rangle_i = X \bmod m_i$.

Operations such as additions and multiplications are done in parallel, on the different RNS moduli:

$$Z = X\,op\,Y = \begin{cases} Z_{m_1} = \langle X_{m_1}\,op\,Y_{m_1}\rangle_{m_1} \\ Z_{m_2} = \langle X_{m_2}\,op\,Y_{m_2}\rangle_{m_2} \\ ... \\ Z_{m_r} = \langle X_{m_P}\,op\,Y_{m_r}\rangle_{m_r} \end{cases}$$

[21]

2.1.1 Chinese Remainder Theorem

It is one of the main theorems in RNS as it states that, knowing the remainders of the integer division, with respect to the elements of a base, it is possible to restore it back to the initial value, expressed with the positional numerical system. The important aspect to emphasize is that the elements of the base must be pairwise relatively prime.
The notation given by Euclid's lemma is inserted, which states that taken $a, b \in \mathbb{Z}$, with a prime number, $a|b$ indicates that a divides b.

Lemma 2.1.1. *[8] Let $a_1, a_2, ..., a_n \in \mathbb{Z}$ be pairwise relatively prime. If $b \in \mathbb{Z}$ and $a_i|b\,forall\,i$, then $a_1a_2...a_n|b$.*

Proof. By induction on n.
We take as a base case $n = 2$. Suppose a_1 and a_2 are relatively prime and both divide b. Use the Bézout's lemma to write $ra_1 + ra_2 = 1$ for some $r, s \in \mathbb{Z}$; also write $b = b_1a_1$ and $b = b_2a_2$. We multiply the Bézout relation by b and then substitute in the divisibility equations:

$$b = bra_1 + bsa_2 = b_2a_2ra_1 + b_1a_1sa_2 = (b_2r + b_1s)a_1a_2$$

which implies that $a_1a_2|b$.
Now assume that the result holds for some $n \geq 2$. Let $a_1, a_2, ..., a_n, a_{n+1} \in \mathbb{Z}$ be pairwise relatively prime and suppose that $a_i|b\,for\,all\,i$. By inductive hypothesis $a_1a_2...a_n|b$.
It suffices to show that $a_1, a_2, ..., a_n, a_{n+1}$ are relatively prime, for the result will then follow from $n = 2$ case. Let $d = (a_1a_2...a_n, a_{n+1})$. If d $\neq 1$, then there is a prime $p|d$. It follows that $p|a_1a_2..a_n$ and a_{n+1}. By Euclid's lemma, we must have $p|a_i$ for some $1 \leq$ i $\leq$ n.

But then p is nontrivial common divisor of a_i and a_{n+1}, contradicting the fact that (a_i, a_{n+1})=1. Hence d = 1, the n+1 case is established and, by induction, the lemma holds for all $n \geq 2$. $\square$

Corollary 2.1.1. *If $a_1, a_2, \dots a_n \in \mathbb{Z}$ are pairwise relatively prime, then $(a_1a_2...a_{n-1}, a_n) = 1$.*

Theorem 2.1.2 (Chinese Remainder Theorem). *[8] Let $n_1, n_2, ..., n_r \in \mathbb{N}$ be pairwise relatively prime. For any $a_1, a_2, ..., a_r \in \mathbb{Z}$ the solution set of the system of simultaneous congruences*

$$x \equiv a_1 \, (mod \, n_1)$$

$$x \equiv a_2 \, (mod \, n_2)$$

$$...$$

$$x \equiv a_r \, (mod \, n_r) \tag{2.2}$$

consists of a unique congruence class modulo $N = n_1n_2...n_r$.

Proof. Consider the map

$$\rho : \mathbb{Z}/N\mathbb{Z} \to \mathbb{Z}/n_1\mathbb{Z} * \mathbb{Z}/n_2\mathbb{Z} ... * \mathbb{Z}/n_1\mathbb{Z}$$

$$a + N\mathbb{Z} \mapsto (a + \mathbb{Z}/n_1\mathbb{Z}, a + \mathbb{Z}/n_2\mathbb{Z}, \dots a + \mathbb{Z}/n_r\mathbb{Z}).$$

i.e. ρ maps to the class of a modulo N to the r-tuple of classes of a modulo n_i. ρ is well-defined since if $a + N\mathbb{Z} = b + N\mathbb{Z}$ then $N|a - b$. As $n_i|N$ for all i, this means $n_i|a - b$ and hence $a + n_i\mathbb{Z} = b + n_i\mathbb{Z}$ for all i.

To prove the theorem it suffices to prove that ρ is a bijection. To see it, first it has to be noticed that x solves the system 2.2 if and only if $x + n_i\mathbb{Z} = a_i + n_i\mathbb{Z}$ for all i and this happens if and only if $\rho(x + N\mathbb{Z}) = (a_1 + \mathbb{Z}/n_1\mathbb{Z}, a_2 + \mathbb{Z}/n_2\mathbb{Z}, \dots a_r + \mathbb{Z}/n_r\mathbb{Z})$. If ρ is a bijection, then there exists a unique $a+N\mathbb{Z} \in \mathbb{Z}/N\mathbb{Z}$ so that $\rho(a+N\mathbb{Z}) = (a_1 + \mathbb{Z}/n_1\mathbb{Z}, a_2 + \mathbb{Z}/n_2\mathbb{Z}, \dots a_r + \mathbb{Z}/n_r\mathbb{Z})$. This means that x solves the system 2.2 if and only if $x \in a + N\mathbb{Z}$ and this is what the theorem states.

Finally, to prove that ρ is bijective is necessary to prove that it is injective, since both its domain and codomain have size $n_1n_2...n_r = N$. Suppose that $\rho(a + N\mathbb{Z}) = \rho(b + N\mathbb{Z})$ for some $a, b \in \mathbb{Z}$. Then $a + n_i\mathbb{Z} = b + n_i\mathbb{Z}$ or $n_i|a - b$, for all i. Since the n_i are pairwise

relatively prime, this means their product, N, divides $a - b$ by Lemma 1.
Hence, $a + N\mathbb{Z} = b + N\mathbb{Z}$ and ρ is injective and that proves the theorem. $\square$

The Chinese Remainder Theorem asserts that there is a unique class $(a + N\mathbb{Z})$, so that x solves the system 2.2, if and only if $x \in a + N\mathbb{Z}$, i.e. $x \equiv a(mod\, N)$. Thus, the system 2.2 is equivalent to a single congruence modulo N. Suppose $n_1, n_2, ..., n_r$ are pairwise relatively prime. Let $N = n_1 n_2 ... n_r$ and $N_i = N/n_i$ (so that N_i is the product of all the n_j except n_i). Then n_i and N_i are relatively prime for all i by Corollary 2.1.1.

Theorem 2.1.3. *[8] Let $n_1, n_2, ..., n_r \in \mathbb{N}$ be pairwise relatively prime and define N, N_i as above. Let m_i be a modular inverse of N_i modulo n_i, i.e. $m_i N_i \equiv 1(mod\, n_i)$. Given $a_1, a_2, ..., a_r \in \mathbb{Z}$ set*

$$a = a_1 m_1 N_1 + a_2 m_2 N_2 + ... + a_r m_r N_r.$$

then a solves the system 2.2 of the Chinese Remainder Theorem. Therefore the solution set of 2.2 is $a + N\mathbb{Z}$, i.e. x is a solution if and only if $x \equiv a(mod\, N)$.

Proof. In light of Theorem 2.1.2, if suffices to show that a is a solution to the system 2.2, hence that $a \equiv a_1(mod\, n_1)$. The same argument works for all of the other congruences. Since $n_1 | N_j\, for\, j \geq 2$, $a \equiv a_1 m_1 N_1 \equiv a_1(mod\, n_1)$ by our choice of m_1. $\square$

2.1.2 Redundant RNS

If a RNS is constructed not only for the representation of data, but also for its protection, usually one shapes it using redundant moduli, such that the system has the ability of self-checking, error-detection and error-correction. In this case the operand X is limited to the information range of $[0, M = \prod_{i=1}^{v} m_i)$, where $v \leq r$, and $m_{r+1}, m_{r+2}, .., m_v$ are the redundant moduli. The interval $[0, M)$ constitutes the legitimate range of the operand X, meanwhile the range $[M, M_v)$ is the associated illegitimate range.
An important property of the Redundant Residue Number System (RRNS) is that an integer represented by residues, can be recovered by any group of r numbers of moduli and their corresponding residue digits. A RRNS having r number of information moduli and $v - r$ number of redundant moduli is denoted as a RRNS(r, v) code [31].
Even in the case of redundant residues, the Chinese Remainder Theorem is still applied in order to be able to convert the represented value to the positional form, only if it is guaranteed to have, at least, the number of legitimate residues.

2.2 Homomorphic Encryption

Homomorphic encryption (HE) is a cryptographic scheme that enables homomorphic operations on encrypted data without decryption [7]. More formally, the computational capability of HE and the equivalency between encrypted computation and the computation over plaintexts is shown as:

$$y = f(x) \Leftrightarrow Enc(y) = g(Enc(x))$$

In this case, $f(x)$ is any arithmetic function in the plaintext domain, $Enc()$ refers to an encryption of a plaintext value, and $g(Enc(x))$ parallels $f(x)$ in the encrypted domain. This enables users to encrypt data and outsource computations like $f(x)$ to the cloud, without sacrificing privacy, which in turn will execute $g(Enc(x))$ and return $Enc(y)$ to the user, who can decrypt the result with their private key [16].
The complete idea of what is meant by homomorphic function is described in Figure 2.1. Taken two spaces such as P for all the plaintexts and C for all the ciphertexts, it is necessary to have a function $Enc(.)$, which maps an element from P to exactly one element in C. Now assuming that each of the sets P, C is equipped with some arbitrary operations $(\circ, \Diamond)$ respectively, it is possible to perform the operation $a \circ b$ for the elements a, b in the set P and the result also will belong to P. The same holds for the set C for the operation of $\Diamond$ [27].
Having a homomorphism between the present sets means being able to map the result of the operation directly onto the set of encrypted elements:

$$E(a \circ b) = E(a) \Diamond E(b)$$

Instead of calculating an unencrypted operation $a \circ b$ and then encrypting it, it is more feasible to perform the homomorphic operation $E(a) \Diamond E(b)$, upon the corresponding ciphertexts $E(a)$, $E(b)$ and the result, when decrypted, would correspond to the desired operation $a \circ b$ [27].

However, there are various types of homomorphic encryption, such that they are divided according to their respective functionalities: partial (PHE), somewhat (SHE) and fully (FHE). The first model, the Partial Homomorphic Encryption scheme, it is able to implement just a single homomorphic operation, such as addition or multiplication, with an unlimited number of times. The second scheme reported, Somewhat Homomorphic Encryption, is able to have a more complex structure: in fact it is possible, with this model, to implement both addition

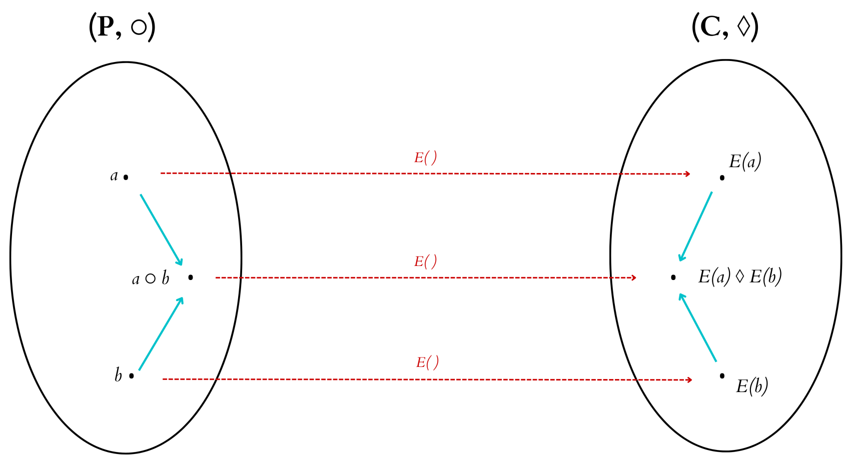


Figure 2.1: Homomorphic Function

and multiplication, but it has a limitation around the depth of the circuits that it is capable to compute. There is a variant of the SHE model, called Leveled Homomorphic Encryption (LHE), that allows instead to have a variable depth of the circuits, but this criterion has to be defined in the initial phase of encryption, during the parameters setting operation, and it cannot be changed later on in the computation. As the last model proposed, it is the Fully Homomorphic Encryption scheme that turns out to be the most innovative of the three: it is in fact the most performing, since it is able to implement any operation, with an unlimited number of times.

The deep differences between types of homomorphic schemes refer to the low level details of their functions: first of all the shape of the ciphered data, the approximation errors and the management of the noise in phase of ciphering, necessary to increase the degree of security. If the design introduces a value of approximation greater than a specific threshold, it is not more in a position to decipher in the correct way the data, therefore there are not more possible homomorphic operations available.

Despite their differences, all cryptographic schemes share the same main operations, which can be summarised as follows:

- Setup: it represents the initial phase of the preparation of the architecture, it turns out however to be also the most important, because the decisions made cannot be changed once started. Key important operations in this phase are the choice of the cryptographic scheme to be implemented and the security parameters.

- Key Generation: once the outline has been defined, the second phase closely concerns the cryptographic aspects, as it involves the keys' generation to encrypt the messages. Different keys are produced, from public, to rotation, to private, which are the ones to be hidden from outside parties and on which all other keys are based. Encryption is possible with both public and private keys, but only private ones can be used for decryption.

- Encryption: main operation that involve the application of keys to hide the original message. At a high level it concerns the transformation from the input value into a ciphertext, i.e. two polynomials. To increase efficiency, this operation applies the Single Instruction Multiple Data (SIMD) methodology, so it is performed not on a single value, but on a vector of messages that are encrypted in a single ciphertext. Then, additions and multiplications of ciphertexts are carried out component wise [6].
  Taken the input message, applied an encoding algorithm, this is transformed into a polynomial (plaintext), which however is still readable from the outside. The ciphertext will be created only at the actual moment of encryption: it has an apparently random behavior and does not leak any information.

- Evaluation: this is the phase involving computation on secure data. Depending on the type of application under study, these operations could take the form of an aggregation of sums, multiplications or other arithmetic manipulations.

- Decryption: last step taking place at the final phase of the communication and, using the private key, from the ciphertext, is feasible to get the message result of the computation. In a specular way in comparison to the operation of encoding, also in this case it is necessary to execute an operation of decoding, in order to obtain the message not under the form of polynomial, but translated to its original form.

An overview of the main operations is shown in Figure 2.2: starting from an initial plaintext message, it is encoded then in a polynomial, but still readable, then transformed in a ciphertext represented by two polynomials. At this time, homomorphic operations are possible according to the implemented scheme: the private key will be used to decrypt, then there will be the decoding process and the result of the plaintext computation.

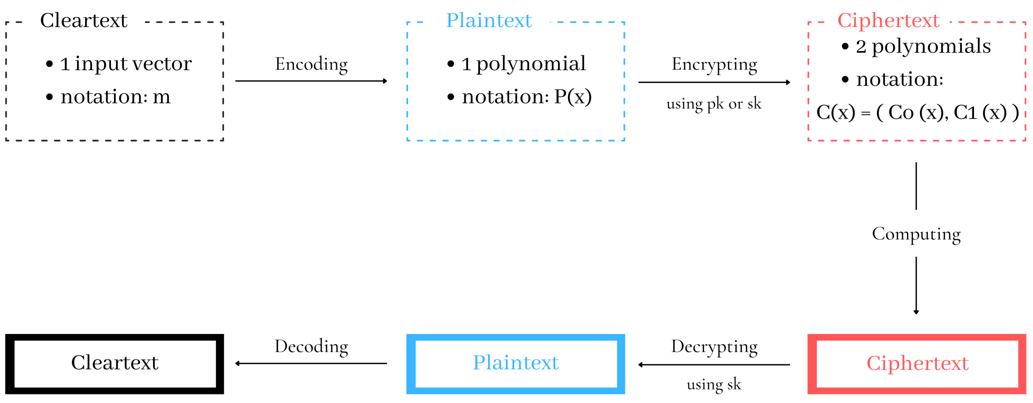


Figure 2.2: Cryptographic Scheme Overview

Once one has clarified the passage of operations to be performed, the low-level algebraic structures are discussed in detail. These models employ complex structures such as polynomial rings, i.e. a ring formed from the set of polynomials in one or more variables, with coefficients in another ring, often a field. Such ring is defined as $R = \mathbb{Z}[X]/(X^n + 1)$ where the dimension n is a power of 2. The arithmetic within this ring is always modulo $(X^n + 1)$. Such algebraic structures form the basis of how incoming data are encoded and modelled, even before encryption is applied: values must first be encoded within such polynomial rings.

Definition 2.2.1 (n^{th} Root of Unity)**.** *[25] Let n be a positive integer. A complex number ω is an n^{th} root of unity if $\omega^n = 1$.*

Definition 2.2.2 (Primitive n^{th} Root of Unity)**.** *[25] A primitive n^{th} root of unity is an n^{th} root of unity whose order is n.*

Lemma 2.2.1. *[25] The function ψ from $\mathbb{Z}_n$ to the n^{th} roots of unity given by $\psi(k) = e^{\frac{2\pi i}{n}k}$ is a group isomorphism.*

Proof. Suppose $e^{\frac{2\pi i}{n}k}$ is an n^{th} root of unity; then $k \in \mathbb{Z}_n$, and $\psi(k) = e^{\frac{2\pi i}{n}k}$. Hence ψ is onto. Suppose $j, k \in \mathbb{Z}_n$ and $\psi(k) = \psi(j)$. In this case, $e^{\frac{2\pi i}{n}k} = e^{\frac{2\pi i}{n}j}$, which happens only if $j = k$. Therefore ψ is one-to-one.
Again suppose that $j, k \in \mathbb{Z}_n$. Say that $j + k \equiv r \pmod{n}$, then it follows that $j + k = n \cdot q + r$ for some $q \in \mathbb{Z}$. Hence, $\psi(j + k) = \psi(r) = e^{\frac{2\pi i}{n}r} = e^{\frac{2\pi i}{n}(n \cdot q + r)} = e^{\frac{2\pi i}{n}j} e^{\frac{2\pi i}{n}k} = \psi(j)\psi(k)$. Thus ψ is operation preserving, and so it is an isomorphism. □

Theorem 2.2.2. *[25] If n is a positive integer, then the primitive n^{th} roots of unity are*

$$\left\{e^{\frac{2\pi i}{n}k} : 1 \leq k \leq n, gcd(k, n) = 1\right\}$$

Proof. An element k of $\mathbb{Z}_n$ has order n if and only if gcd(n, k) = 1. It follows from Lemma 2.2.1 that an n^{th} root of unity, $e^{\frac{2\pi i}{n}k}$, has order n if and only if gcd(n, k) = 1. □

Having settled the main algebraic concepts, it is possible to give a formal definition of cyclotomic polynomials.

Definition 2.2.3 (n^{th} Cyclotomic Polynomial)**.** *[25] For any positive integer n the n^{th} cyclotomic polynomial, $\Phi_n(x)$, is given by*

$$\Phi_n(x) = (x - \omega_1)(x - \omega_2)...(x - \omega_s),$$

where $\omega_1, \omega_2, \dots, \omega_s$ are the primitive n^{th} roots of unity.

For any positive integer n, the n^{th} cyclotomic polynomial is the unique irreducible polynomial with integer coefficients that are divisors of $x^n - 1$ and not a divisor of $x^k - 1$, for any $k \leq n$. Its roots are all n^{th} primitive roots of unity $e^{\frac{2\pi i}{n}k}$, where k runs over the positive integers not greater than n and coprime to n.
It follows the Theorem 2.2.2 that it is possible to write the n^{th} cyclotomic polynomial as

$$\Phi_n(x) = \prod_{1 \leq k \leq n} (x - e^{\frac{2\pi ik}{n}})$$

Example 2.2.1 and 2.2.2 are included in order to give a practical case of use.

Example 2.2.1. *For prime p, the factorization of $x^p - 1$ is*

$$\Phi_p(x) = \frac{x^p - 1}{x - 1} = x^{p-1} + x^{p-2} + ... + x^2 + x + 1$$

Example 2.2.2. *For $n = 2p$, with odd prime p*

$$\Phi_{2p}(x) = \frac{x^{2p} - 1}{\Phi_1(x)\Phi_2(x)\Phi_p(x)} = \frac{x^{2p} - 1}{\Phi_2(x)(x^p - 1)} = \frac{x^p + 1}{x + 1} = x^{p-1} - x^{p-2} + x^{p-3} - ... + x^2 - x + 1$$

Depending on the scheme and type of data to be encoded, models are specialised according to how the values are sampled in the various polynomial rings. Mostly used are either uniform sampling or from a Gaussian distribution. As defined above, there are two types of spaces in which to sample elements: plaintext space and the ciphertext one. These two sets are respectively rings, in module to the size of the set they represent: the plaintext space is R_t, where $t \geq 2$ is the integer modulus. A plaintext is normally a single element in R_t encoding the original plaintext message. Likewise, the ciphertext space R_q, has $q \gg t$ as the coefficient modulus. Similar to R_t, polymials in R_q are reduced modulo q as well. A ciphertext c is a pair of two elements in R_q, denoted by $(c[0], c[1])$ [2].
In order to get a clearer idea of the notions of rings, an Example 2.2.3, contained in [18], detailing the steps has been included.

Example 2.2.3. *[18] As specified, the underlying algebra concerns polynomials, with a few differences to the standard ones: the first concerns the coefficients, which are all integer t modulus; the second difference focuses on the fact that the polynomials themselves are also objects of division with respect to another polynomial. The form of this polynomial is* (X^n+1)*. Let* $t = 24$*,* $n = 16$*, it is possible to consider only polynomials with modulo* $X^{16}+1$*, thus from* x^0 *to* x^{15}*. Any larger powers will be reduced by division by the polynomial modulus. The polynomials to be considered will be of the form:*

$$a_{15}x^{15} + a_{14}x^{14} + a_{13}x^{13} + ... + a_1x + a_0$$

where each a_i *can range from* 0 *to* $t-1$*.*

In the Figure 2.3, each loop represents the 24 possible values of the coefficient of one of the powers of x that appears in the polynomial. The green dots represent where the 0 value of the coefficient lies.

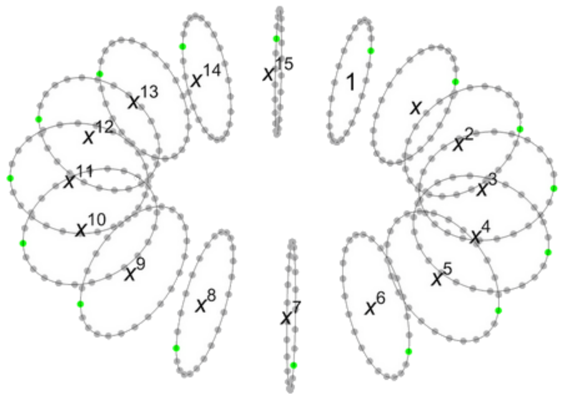
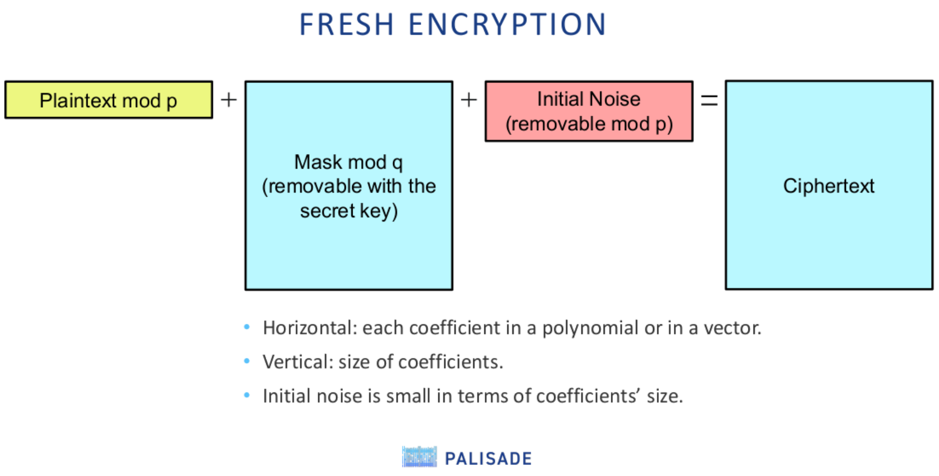


Figure 2.3: Plaintext seen as polynomial rings. Taken from [18]

Once the general encoding of values into plaintext, i.e. polynomials, have been clarified, the last step to be analyzed is the actual homomorphic encryption. It is highlighted that both LHE and FHE encrypt data as a tuple of polynomials modulo an irreducible cyclotomic polynomial, where the N^{th} cyclotomic polynomial has roots corresponding to the N^{th} primitive roots of unity [16].

A further detail that is characteristic of these schemes is noise, inherent in the ciphertext construction. FHE schemes require users to track the (unavoidable) noise growth in homomorphic ciphertexts. In particular, the presence of noise is necessary for security purposes and a small amount is injected in each ciphertext during initial encryption. Then, evaluating consecutive operations on the encrypted data causes this noise to grow and once a certain threshold is reached, the ciphertext can no longer be decrypted and is rendered useless [16]. FHE schemes inject noise into ciphertexts in accordance with the Learning With Errors (LWE) paradigm [26] to make them resilient to cryptanalyses. The hardness of the LWE problem and its variant Ring LWE (RLWE) guarantee the security of all FHE schemes. The induced noise needs to be kept below a threshold to ensure successful decryption [16].

Figure 2.4 shows the procedure for calculating the ciphertext. It is called fresh encryption as no operations have yet been performed on the cipher. Taken the plaintext, a secret mask is added to it, which can be removed using the private key, and noise, which is also manageable and removable. With this procedure, having the private key, decryption will be correct. Figure 2.5, on the other hand, shows the contents of the objects after various homomorphic operations have been performed on the cipher. In this case, the noise has increased, due to the continuous aggregations, to the point where it can no longer be removed. This will lead to not being able to decrypt the ciphertext with the private key any more.


FRESH ENCRYPTION
Plaintext mod p
+
Mask mod q (removable with the secret key)
+
Initial Noise (removable mod p)
=
Ciphertext
PALISADE

- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.
- Initial noise is small in terms of coefficients' size.

Figure 2.4: Fresh Encryption. Taken from [22]

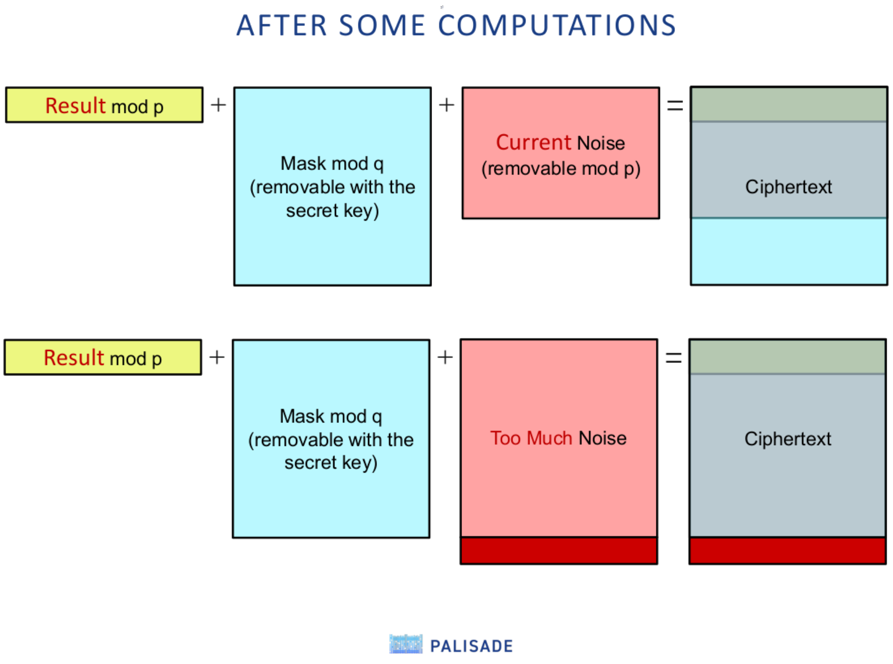

AFTER SOME COMPUTATIONS
Result mod p
+
Mask mod q (removable with the secret key)
+
Current Noise (removable mod p)
=
Ciphertext
Result mod p
+
Mask mod q (removable with the secret key)
+
Too Much Noise
=
Ciphertext
PALISADE

Figure 2.5: Encryption After Computations. Taken from [22]

Example 2.2.4. *[18] The plaintext is a polynomial from the ring with polynomial modulus $x^n + 1$ and a coefficient modulus, t. The encryption of a plaintext is a ciphertext which is represented by two polynomials from the ring with the same polynomial modulus, but a coefficient modulus q, which is typically much larger that t.*
We will use $n = 16$, $t = 7$, $q = 874$ (those parameters are not secure).
For the private or secret key, which we call s, we generate a random polynomial with coefficients of either -1, 0 or 1. For example,

$$s = x^{15} - x^{13} - x^{12} - x^{11} - x^9 + x^8 + x^6 - x^4 + x^2 + x - 1$$

Next we generate a public key, using a random polynomial from the ciphertext space, with coefficients modulo q, which we call a.

$$\begin{aligned} a = 42x^{15} - 256x^{14} - 393x^{13} - 229x^{12} + 447x^{11} - 369x^{10} \\ - 212x^9 + 107x^8 + 52x^7 + 70x^6 - 138x^5 + 322x^4 + 186x^3 - 282x^2 - 60^x + 84 \end{aligned}$$

We also define an error polynomial, which is "small" in the sense that it has small coefficients drawn from a discrete Gaussian distribution.

$$e = -3x^{15} + x^{14} + x^{13} + 7x^{12} - 6x^{11} - 6x^{10} + x^9 + 4x^8 - x^6 + 3x^5 - 4x^4 + 4x^3 + 4x + 1$$

The public key is then defined as the pair of polynomials, $pk = ([-as + e]_q, a)$.
Encryption takes a plaintext, which is a polynomial with coefficients modulo t, and converts it to a pair of polynomials with coefficients modulo q. As an illustration, we will encrypt a very simple polynomial message $-m = 3 + 4x^8 \equiv 3 - 3x?8$ - with only two non-zero coefficients.
Encryption requires three more small polynomials. Two error polynomials drawn from the same sort of discrete Gaussian distribution used in the error polynomial in the public key, and another polynomial we will call u which has coefficients drawn from $-1, 0\, or\, 1$, just like the private key.

$$e_1 = -5x^{15} - 2x^{14} + 3x^{13} - x^{12} - 4x^{11} + 3x^{10} + x^9 + 4x^8 + 4x^7 + 5x^6 - 4x^5 - 3x^4 - 3x^3 + 2x^2 - 6x + 4$$

$$e_2 = -7x^{15} + 2x^{14} - 4x^{13} + 5x^{11} + 2x^{10} - x^9 + 4x^8 - 4x^7 - 3x^6 + 2x^5 - 2x^4 + x^3 - 2x + 2$$

and

$$u = x^{14} + x^{13} + x^{12} - x^8 - x^5 - x^3 + 1$$

The public key is then defined as the pair of polynomials, $pk = ([-as + e]_q, a)$*, where the polynomial arithmetic is done modulo the polynomial modulus and the coefficient arithmetic modulo* q*. The ciphertext is represented by two polynomials calculated as* $ct = ([pk_0 u + e_1 + qm/t]_q, [pk_1 u + e_2]_q)$*. In Figure 2.6 there is the illustration of the first part of the cipher,* ct_0*.*

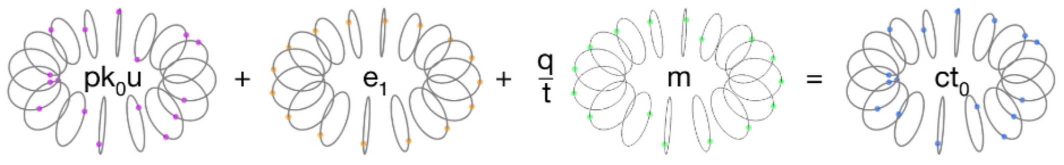


Figure 2.6: Encryption. Taken from [18]

Different management of noise allows various schemes to have the possibility of doing an unlimited number of arithmetic operations, but always remaining with a cipher capable of being deciphered correctly. The primary mechanism for reducing noise in LHE is modulus switching, which effectively reduces the ciphertext size while also scaling down the noise. More specifically, this procedure removes a prime from q, the ciphertext modulus, effectively lowering the bit size of the coefficients of the ciphertext polynomials. However, modulus switching can only be invoked a limited number of times; eventually one will run out of primes in q and the noise can no longer be mitigated [16].
The FHE scheme, instead, manages to perform an unlimited number of operations, inserting a bootstrapping mechanism that allows to reduce the noise. As Gentry showed in [12], this noise reduction operation is performed by acting on the ciphertext, but in the context of the decryption algorithm. The latter is converted into a circuit which takes the ciphertext and an encrypted version of the key, while producing as a result a ciphertext which represents the same initial plaintext, but with the noise reduced to zero [16]. What Gentry has demonstrated is the possibility to have the same starting ciphertext, but without noise, so with the possibility to continue with the aggregation operations. Unfortunately, bootstrapping is the bottleneck of FHE: it is worth reducing the number of these operations to improve performance.

2.2.1 Comparison between schemes

There are various homomorphic encryption schemes that can be found in the literature, dividing between the type of data they manipulate, the noise handling schemes and the various scaling factors they have within them. The broad distinction between schemes can be made according to whether they implement exact or approximate arithmetic, over integer or real numbers.

The two most widely used models on integer arithmetic are BGV [5] and BFV [9], which base their execution on encrypting integers modulo a predetermined p, the plaintext modulo. This detail is fundamental, because it is necessary to establish p in order not to have an overflow during the aggregation operations. Both BGV and BFV have considerably slow bootstrapping procedures; depending on parameter choices (such as the plaintext modulus and cyclotomic order), a single bootstrap can take anywhere from several minutes to several hours, which is impractical for any realistic HE computation. Thus, most implementations of BGV/BFV do not include bootstrapping and are used exclusively in LHE mode [16]. As an advantage of these encryption schemes, it is the possibility to perform the batching operation, that is the procedure of encoding a vector of elements at the same time into a plaintext.

If the application scenario requires it, there is also a homomorphic cryptographic scheme for approximate arithmetic operations on real values, CKKS [7]. The key difference is the need to keep track of the scale factor, that is multiplied with plaintext values during encoding, which determines the bit-precision associated with each ciphertext. The scale doubles when two ciphertexts are multiplied, which may result in overflow as the scale becomes exponentially larger. Thus, a rescaling mechanism must be invoked to preserve the original scale after multiplications, which is quite similar to modulus switching and serves a similar role to reduce ciphertext noise [16]. As with the previous schemes, the CKKS also allows batching of initial values and it has difficulties while computing in bootstrapping mode, so it tend to be used in LHE mode too.

2.2.2 PALISADE

The concrete implementation of the reported project has been made possible through the use of PALISADE, a well known lattice cryptography library that currently includes efficient implementations of all the homomorphic schemes previously presented. PALISADE is a cross-platform C++11 library supporting Linux, Windows, and macOS. The supported compilers are g++ v6.1 or later and clang++ v6.0 or later. The library also includes unit tests and sample application demos. PALISADE is available under the BSD 2-clause license [22].

As reported by Figure 2.7, the architecture has multiple layers, each with a specific purpose with respect to the operations it implements:

- Primitive Math Layer supports low-level modular arithmetic, number theoretic transforms and integer sampling. This layer is implemented to be portable to multiple hardware computation substrates.
- Lattice Operations Layer has the responsibility to sustain lattice operations, ring algebra and lattice trapdoor sampling.
- Crypto Layer contains efficient implementations of lattice cryptography schemes.
- Encoding Layer supports multiple plaintext encodings for cryptographic schemes [23].

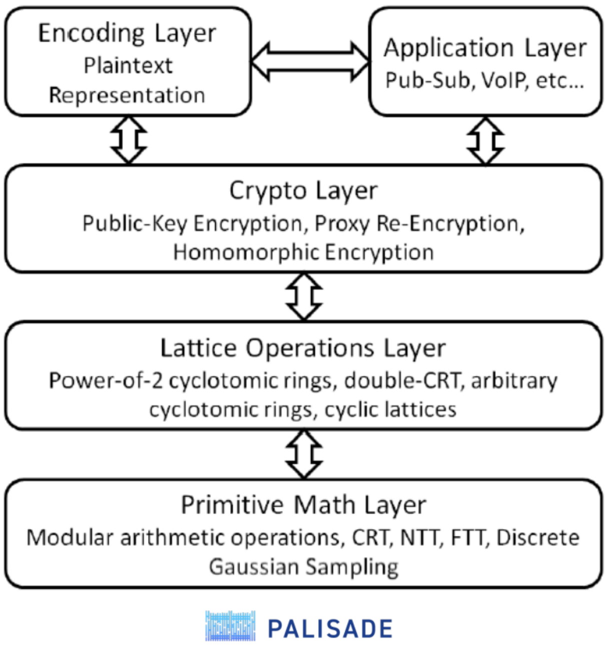
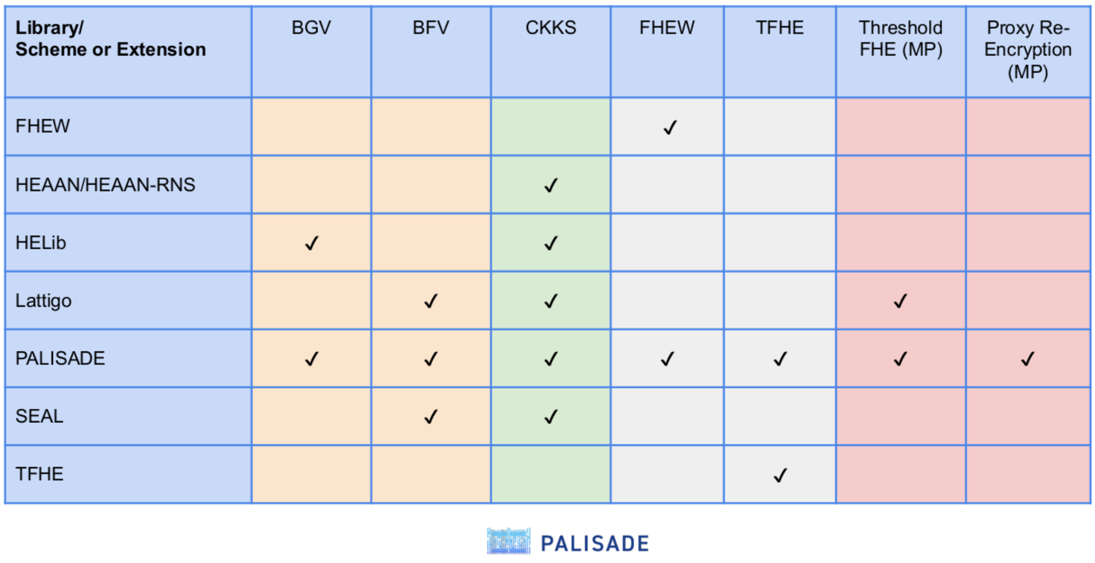

Encoding Layer
Plaintext Representation
Application Layer
Pub-Sub, VoIP, etc...
Crypto Layer
Public-Key Encryption, Proxy Re-Encryption, Homomorphic Encryption
Lattice Operations Layer
Power-of-2 cyclotomic rings, double-CRT, arbitrary cyclotomic rings, cyclic lattices
Primitive Math Layer
Modular arithmetic operations, CRT, NTT, FTT, Discrete Gaussian Sampling
PALISADE

Figure 2.7: Palisade Architecture. Taken from [23]

From a technical point of view, PALISADE stands out from other libraries by being able to implement the totality of the homomorphic schemes, present in the literature, unique in its kind as the Figure 2.8 shows. In addition to being able to perform FHE computations, it provides implementations for the building blocks of lattice cryptography capabilities along with end-to-end implementations of advance lattice cryptography protocols for public-key encryption, proxy re-encryption, program obfuscation and more. They also provide an experimental

platform for research and development as well as an implementation ready platform of known protocols that can directly be integrated into applications [6].
A further innovative aspect of this library is to have a version of each scheme with an additional RNS reduction: this modification was included for purposes strictly related to performances. One common aspect of many HE schemes is the need to manipulate large algebraic structures with multi-precision coefficients. This multi-precision arithmetic is performed for polynomials of large degrees (several thousand) with relatively large integer coefficients (several hundred bits). Implementing the necessary multi-precision modular arithmetic is computationally expensive. A way to make these operations faster is to use the Residue Number System (RNS) to decompose large coefficients into vectors of smaller, native (machine-word-size) integers. RNS allows some arithmetic operations to be performed completely in parallel using native instructions and data types, thus potentially improving the efficiency [2].
Furthermore, a cross-sectional study on the different homomorphic libraries has been performed in [16], comparing PALISADE with the most famous and recent libraries currently in use. The tested cases are the most varied: from the implementation of boolean, integer and up to floating point circuits. As a result of this study, it was found that PALISADE exhibits the fastest overall speeds for both the integer and the floating-point encodings [16], but as the authors point out, it always depends on the use case: if the application has many comparisons, it is not the most efficient, because it is based on a very large plaintext module, hence these operations quickly become prohibitively expensive in terms of noise [16].



Library/ Scheme or Extension	BGV	BFV	CKKS	FHEW	TFHE	Threshold FHE (MP)	Proxy Re-Encryption (MP)
FHEW				✓			
HEAAN/HEAAN-RNS			✓				
HELib	✓		✓				
Lattigo		✓	✓			✓	
PALISADE	✓	✓	✓	✓	✓	✓	✓
SEAL		✓	✓				
TFHE					✓		

PALISADE

Figure 2.8: Comparison between libraries. Talken from [22]

The closing detail to underline, in respect to the use of PALISADE, is the well detailed documentation that it is able to provide, especially regarding the initial phase of the cryptographic model, that is the choice of parameters. This library allows full freedom of movement to navigate and perform a finetuning of each measure that flows into the creation of a ciphertext, providing directives depending on the purpose one wants to achieve.
The authors of PALISADE reiterate that they have followed the guidelines, available to the reference documentation in [1], which dictate the main parameters to set in order to have a certain level of security guaranteed, intended as the hardship to invert the mapping without the use of the secret key. Usually such a parameter is defined at 128 bits, where it represents the fact of having to perform 2^{128} operations to break the encryption scheme. This is not the only limit on which it is possible to act: customizing q, the ciphertext modulus, which affects both performance and security; automating the maintenance of the ciphertext at runtime, with modulus switching or bootstrapping techniques and, lastly, there are choices on the level of depth of the circuits, for the LHE model, in order to obtain the best solution for the project in execution and allow PALISADE to distinguish itself from competitors.

Chapter 3

SYSTEM ARCHITECTURE

This chapter outlines an overview of the general architecture of the current project: first, a review of the components that form the model is included, analyzing their behaviors and benefits they bring to the structure and, following this, the concrete implementation using the PALISADE library is presented.

3.1 Overview

The architecture shows a multi-level and fully decentralized organization, such that it finds easy adaptation in an MCS model as the one studied. The main aspect, strongly encouraged, is to emphasize the ability to scale in each component and still have a solid framework at the base. As summarized by Figure 3.1, one can divide the main communication actors into three distinct roles: the end users, the aggregator node and the cloud platform. In this model each

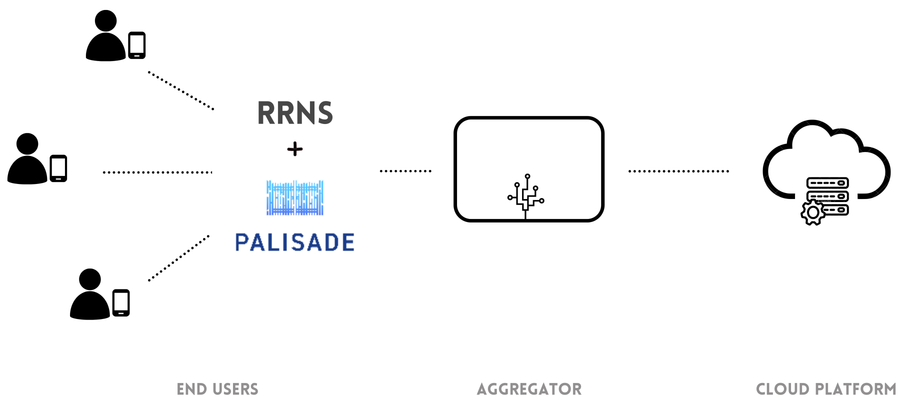


Figure 3.1: Structure Overview

component is taking an active part in the communication, as the data flow originates from the sensors belonging to the end users, it is then processed under the role of the aggregator and, ultimately, stored in the cloud platform. During this transmission, such data always moves in encrypted form: in fact, the aggregator does not possess the key to decrypt them, but it is nevertheless able to manipulate them through the use of homomorphic encryption, that allows it to perform encrypted aggregations. It then sends the encrypted results to the cloud platform.
Users are the entry point for the project: they are the direct sources from which the data are taken. In a MCS model, such as the one constructed, they are represented by the subjects taking part in the study and using their smartphone as a tool to send their information.
An additional precaution was taken when sending data: coding in redundant RNS. Each cipher is divided into its own RNS representation, with redundant modules in addition, the purpose of which is to restore the data in the event of a loss. This encoding ensures that, if at least the legitimate modules reach the next node in the network, the data can be processed and the communication can continue, without having to resort to re-transmitting it, which would cause the sensor to consume more energy. With a decentralized network based on the MCS paradigm, network problems can be very frequent, considering the extreme heterogeneity of networks, devices and intermediate communication nodes. Therefore, having a strategy that saves possible re-transmissions due to network problems can be a way to make the whole structure more robust.

Once the data have been created, they are sent to the second entity in the architecture, which may also be a set, depending on the size of the project and the computing power required, which is the aggregator node. This part of the communication focuses on the question raised by the study itself: aggregation operations, arithmetic operations on the data, are carried out in order to draw relevant or statistical conclusions on habits, behaviours and movements. It is therefore the critical point of the entire infrastructure: the complete knowledge of the dataset must travel through the aggregator node and, if it is hacked during the computation, the network attacker will be able to freely read the content of the communication. Not only that, the malicious user might also be able to trace the identities of the subjects participating in the study, using data from the sensors, and in doing so, the architecture would also have violated the principle of confidentiality previously promised to its users. For this reason, the model is structured in such a way that the aggregator is unable to remove the encryption, as it has no key. It finds itself having to manipulate ciphers, instead of plaintexts, and it is thus completely unaware of the content of its arithmetic operations. In the event of a breach, an external party will not be able to see the data directly, but it will have to act by forcing operations on the cryptographic scheme.

The last subject considered when creating the architecture is the cloud providing platform. Aggregated data here terminate their submission and they are stored to be analyzed in more detail at a later stage. These types of structures are often for public access, layered then with more complex plans for specific categories of users. An example might be public clouds like Alibaba Cloud, Microsoft Azure, Google Cloud, Amazon Web Services (AWS) and IBM Cloud. In this last node of the model, aggregated data are collected and only in this stage it is possible to work on the decrypted data, since the cloud platform will possess the private key, confident that it has traveled along the network without ever being in the clear.

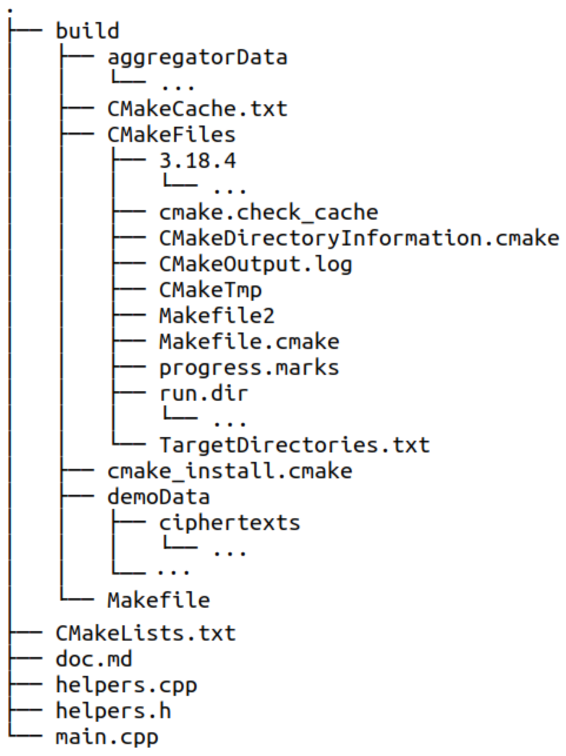
3.2 Scheme Implementation

Going more into the details of the implementation of the proposed project, a distinction must be made between the stages to execute:

- Initial phase: it consists in the data collection and consequent encrypting.
- Sending: taken the ciphertexts, these are converted into their RRNS representation, then sent to the aggregator.
- Aggregation: the RRNS representation is removed and homomorphic operations are performed on the ciphers.

The concrete case study involves the use and manipulation of data in the form of GPS coordinates and mobility tracks. They represent sensitive data as they allow, when analyzed, to trace the movement habits of users, if not also their direct identities. The proposed project aims to collect this mobility data and aggregate it, in order to calculate the sum of the distances covered by users. Through the use of homomorphic encryption, this aggregation is secure even for such sensitive data.

The MCS context is composed as follows: the sensors in the field are represented by the smartphones that the users have and, since it is only the mobility traces and GPS coordinates of the devices that are of interest, no active participation of users is required, only the activation of their position. This data is then encrypted and reduced to its RRNS representation, by the devices, and then sent to the aggregator, which will calculate the encrypted sum of the distances covered by the users, in order to obtain the total sum of the trajectories of all participants.
Data collection was simulated using the public GeoLife [13] dataset, organized by Microsoft Research Asia, which contains mobility data on 182 users, over a sampling period from April 2007 to August 2012. These data were prepared, with a previous processing step, to obtain the distances between the reported points.
The homomorphic sum of these distances was made possible through the use of PALISADE [22], which is capable of implementing lattice cryptography building blocks and leading homomorphic encryption schemes [22]. Since it is the only library to have implementations available for each cryptographic scheme, it was chosen to use it in its completeness. The value of the distances was processed as follows: in one file, the approximate distances were stored, with decimal values, optimal for the use of the CKKS [7] approximate arithmetic scheme; in a second file, instead, the distances rounded to the unity were gathered, such that they could be applied to the BGV [5] exact arithmetic scheme.
Regardless of the model used, the structure of folders and files is as shown in Figure 3.2. From such directory, it is first necessary to discern the source files, then the high-level contents of the folder, from the run-time components, defined through the `build` folder. In fact, the latter contains all the files essential for compilation, such as the `Makefile`, and the folders that are created during the execution itself. Of particular note are the `demoData` and `aggregatorData` folders, which respectively contain the binary files representing the ciphers, built with PALISADE, and intermediate temporary files needed for computation by the aggregator node. The source code, on the other hand, can be divided between the `helpers` files, which contain additional functions that improve the readability of the code, and the `main` file concerning the central activities of the program, such as RRNS encoding and methods for encryption. A complete view of the implemented code can be found in the dedicated GitHub repository in [3].



```
.
├── build
│   ├── aggregatorData
│   │   └── ...
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   │   ├── 3.18.4
│   │   │   └── ...
│   │   ├── cmake.check_cache
│   │   ├── CMakeDirectoryInformation.cmake
│   │   ├── CMakeOutput.log
│   │   ├── CMakeTmp
│   │   ├── Makefile2
│   │   ├── Makefile.cmake
│   │   ├── progress.marks
│   │   ├── run.dir
│   │   │   └── ...
│   │   └── TargetDirectories.txt
│   ├── cmake_install.cmake
│   ├── demoData
│   │   ├── ciphertexts
│   │   │   └── ...
│   │   └── ...
│   └── Makefile
├── CMakeLists.txt
├── doc.md
├── helpers.cpp
├── helpers.h
└── main.cpp
```

Figure 3.2: Scheme Directory of [3]

3.2.1 PALISADE Encryption

As the first real computation to perform on data, during the initial phase, there is the homomorphic encryption, in order to protect the sensitive information that the data contains. This task is performed immediately after the data collection phase, both of which are performed by the devices under end-user management.

PALISADE's source code structure is broken down into specific classes, in order to design the components needed to implement the various cryptographic schemes. The objects used in an encryption model can be summarized as follows:

- `CryptoContext`: it is the class that provides all PALISADE encryption functionalities [23]. Each object created must be considered dependent on the CryptoContext. It is the basic reference class, distinguished by implemented scheme and custom parameters for constructing it. PALISADE incorporates the standard security tables, developed during the Homomorphic Encryption standardization process, described at [1]. Users of the library can construct a CryptoContext by specifying the parameter sets defined in the standard [23].

- `Plaintext`: it is the element that represents the unencrypted text. It is the object that is manipulated by the CryptoContext and then it returns the Ciphertext.
- `Ciphertext`: it is the result of the encryption routine, initiated by the CryptoContext, on the Plaintext. The methods that apply encryption and decryption need `Keys`, which are also generated by the CryptoContext. Homomorphic operations, performed in the second phase of communication, occur on objects of type Ciphertext.

The Listing 3.1 shows how the CryptoContext was created for the BGV exact integer arithmetic scheme. First, it is necessary to include the interface `palisade.h` and the namespace `lbcrypto`, proper to the use of the library.
When defining the scheme, one can customize the parameters that will compose it, or leave the default ones. The `plaintextModulus` represents the size of the plaintext space over which to perform the aggregation operations, so it is necessary to choose a value such that it is not exceeded during the latter. This value must be a prime number that is compatible with the encoding method, hence the method of creating the plaintext. A convenient plaintext modulus for most cases: $p = 65.537$ [23]. `sigma` is the distribution parameter used for the underlying Learning with Errors problem and it is usually defined as a fixed value. `depth` is one of the parameters on which the size of the final cipher will be based, as it is the maximum multiplicative depth of the circuit one is trying to compute. Lastly, the `securityLevel` represents the number of 2^{bits} operations to perform in order to break into the cryptographic scheme. If, for example, a security level of 128 is defined, then it will take 2^{128} operations for a malicious user to obtain the plaintext value.
Once the parameters setting has been defined, the build method `genCryptoContextX` must be invoked in order to create the object of type CryptoContext. In the reported case a CryptoContext is created via the BGV scheme, but it should be noted that for every other PALISADE scheme it exists its own method of creation. The underlying lattice layer element is either a Poly, a NativePoly, or a DCRTPoly [23].
The final operation to perform, on the CryptoContext, is to activate the capabilities it can have later in the code: `ENCRYPTION`, so as to perform public/symmetric encryption; `SHE`, in order to apply aggregation operations, such as sums or multiplications, and `LEVELEDSHE` to have dynamic management of multiplicative levels, compress ciphers and perform rescaling operations.

```
#include "palisade.h"
using namespace lbcrypto;

// Parameters setting
int plaintextModulus = 65537;
double sigma = 3.2;
uint32_t depth = 1;
SecurityLevel securityLevel = HEStd_128_classic;

// CryptoContext creation
CryptoContext<DCRTPoly> cc;
cc = CryptoContextFactory<DCRTPoly>::genCryptoContextBGVrns(
        depth, plaintextModulus, securityLevel, sigma, depth, RLWE, BV);

// Enabling capabilities
cc->Enable(ENCRYPTION);
cc->Enable(SHE);
cc->Enable(LEVELEDSHE);
```

Listing 3.1: CryptoContext for BGV Scheme [3]

Subsequent to the creation of the CryptoContext, in the Listing 3.2 the object of type Ciphertext is implemented, starting first with an array of integers as input, then going through the creation of a plaintext in Single Instruction Multiple Data (SIMD) fashion, to the application of the keys generated by the CryptoContext. The `makeCipher` function then has a return value of an object of type `Ciphertext`, composed of polynomials of type `DCRTPoly`.

```
Ciphertext<DCRTPoly> makeCipher (CryptoContext<DCRTPoly> &cc, vector<int64_t> v) {
  LPKeyPair<DCRTPoly> keyPair = cc->KeyGen();
  Plaintext plain = cc->MakePackedPlaintext(v);
  auto cipher = cc->Encrypt(keyPair.publicKey, plain);
  return cipher;
}
```

Listing 3.2: Ciphertext Composition [3]

3.2.2 RRNS Encoding

As previously mentioned, the following encoding step is performed at the stage of actually preparing data for submission, from the sensors in the field, to the aggregator node. This auxiliary phase is added so that the system can be modeled with an additional degree of redundancy, which can provide more robustness and avoid unnecessary re-transmissions, in case of a packet loss. This operation is therefore performed on the ciphers, by the end users, and the second node in the network, the aggregator, will receive only the redundant residue vectors that represent the initial cipher.
Since the RRNS encoding and decoding are neutral operations, with respect to the different ciphers, it was possible in the second stage to reconstruct the original data, without it being sent in the first place. The use of this technique, therefore, not only brings useful redundancy in the data sending phase, but also an additional layer of security: if detected during sending, the data actually delivered contains the residues of the binary representation of the ciphers, which to an outside user has no meaning, since they do not possess the RNS base, and even in the event that an attacker succeeds in removing the RRNS encoding, they would still have the entire cryptographic scheme to decipher before seeing the data in the clear.

Once the construction of the ciphers has been done, with respect to the mobility data of the application scenario, the following actions need to be performed:

- Setting the RNS base: it represents the set of modules, relatively co-prime, with reference to which the residual coding values will be defined. This base must be consistent across the span of the entire architecture, once chosen it cannot be changed, and every value is coded according to it. The number of modules in the base depends solely on the maximum value that is to be represented, keeping in mind that the representation interval is given by the product of the modules in the set. In the application case, it is emphasized that the base contains redundant modules, thus more values than those that are needed.
- Ciphers conversion: ciphertexts need to be manipulated and transformed into a sequence of integer values, in order to apply the RRNS encoding. This stage exploits PALISADE's ability to take advantage of serialization, thus the process by which complex objects such as ciphertexts themselves can be written to binary files. The latter are interpreted and

read as sequences of integers, so 4 bytes at a time, and each cipher is then stored as a vector of integers.

- RRNS enconding: once the meaning of a ciphertext has been reformed, it is necessary to take the RNS base and compute the corresponding vector of residues. In doing so, a ciphertext is converted to a vector of integer vectors, since each value read from the binary file is replaced with a vector of residues, which are themselves integers. The intermediate node, thus the aggregator, will receive this set of integer vectors, and no longer an object of type `Ciphertext`.

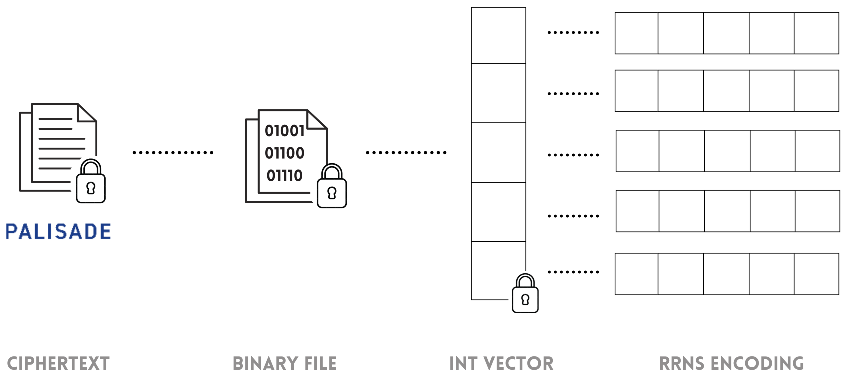


Figure 3.3: RRNS Encoding of a cipher

A concise diagram of the process of editing a ciphertext is defined in Figure 3.3, in order to obtain its RRNS representation. It is again emphasized that this encoding operation is subsequent to the application of the encryption: the objects manipulated for sending are the ciphertexts themselves.
The last operation to clarify in more detail is the serialization, which PALISADE is able to implement, for its cryptographic schemes, thus the process by which a data structure or object is translated into a format that can be stored or transmitted [23]. It will then be up to the receiver to reconstruct the original contents of the file, through the deserialization operation. Having to handle very complex data structures, PALISADE relies on Cereal, a C++ library which takes arbitrary data types and reversibly transforms them into different representations, such as compact binary encodings, XML or JSON [17]. Cereal was designed to be fast, light-weight and easy to extend: it has no external dependencies and can be easily bundled with other code or used standalone [17].

The portion of the code that deals with writing objects to binary files, through the use of Cereal, was included in the Listing 3.3. The function `Serialize` takes as parameters the file to write to and the object to be stored, along with the serialization mode, which in the specific case concerns binary writing. Cereal succeeds in its intent through the use of archive-defined objects, which characterize the type of storage to assign to its data structures. The binary archive can be used by including `<cereal/archives/binary.hpp>`. The binary archive is designed to produce compact bit level representations of data and is not human readable [17].

```
template <typename T>
void Serialize(const T& obj, std::ostream& stream, const SerType::SERBINARY& st) {
    cereal::PortableBinaryOutputArchive archive(stream);
    archive(obj);
}
```

Listing 3.3: Serialization

3.2.3 Homomorphic Operations

With RRNS encoding the work of the end-users' devices terminates, so the data is finally sent to the second component of the architecture, which is represented by the aggregator. At this stage it is necessary to perform aggregation operations, then arithmetic calculations, on the newly collected data in order to facilitate the analysis actions, that will be performed later on by the cloud platform. Depending on the information that is to be taken from the data, these calculations may change, but the innovative aspect remains that they handle a set of arithmetic operations directly on the encrypted data and will remain so until the receiver is reached.

On the reference architecture, the role of the aggregator then becomes to perform the following operations:

- RRNS decoding: when sending, this encoding was included in order to ensure redundancy in the model and avoid re-transmissions, resulting in the devices consuming less energy. When considering the aggregator, however, this redundancy may not be as advantageous, as it is reasonable to consider that the architecture uses networks that are more stable, powerful and better organized, so it is possible to remove such protection.

In the event that this assumption turns out to be too restrictive, the model can be immediately modified by reinserting RRNS encoding to the data, before sending them to the cloud platform.

- Aggregations: in order to perform meaningful operations on the data, it is necessary to reconstruct the `Ciphertext` type object, so that it can take full advantage of the PALISADE library functions, which incorporate a consolidated set of arithmetic operations on encrypted data.

To concretize the first role of the aggregator, thus RRNS decoding, it applies the Chinese Remainder Theorem [8] to reconstruct the vector of integers symbolizing the binary representation of a cipher.
Once the RRNS encoding is removed, the same conversion as in the previous step is performed, taking advantage of the fact that once an object is synthesized in binary format, then it can be read and interpreted as another data type. The aggregator writes the vector of integers to a binary file and subsequently, through the deserialization operation provided again by the Cereal [17] library, it is able to interpret the set of bytes as an object of type `Ciphertext`.
In Listing 3.4 there is the deserialization function provided by the Cereal library: it takes as input the generic type object, in which to save the content of the file, the name of the file to be read and, lastly, the format type of the serialization, where by default the binary format is considered. In Figure 3.4, on the other hand, the complete conversion process has been schematized, leading as the final step to obtaining a `Ciphertext`.
It is emphasized that these conversions remain neutral on the actual content: in fact, it is not decrypted, but merely reinterpreted according to the context of action, first seen as redundant residues of an integer representation, later then reconstructed into the form expected by PALISADE, for aggregations.

```
template <typename T>
void Deserialize(T& obj, std::istream& stream, const SerType::SERBINARY& st) {
    cereal::PortableBinaryInputArchive archive(stream);
    archive(obj);
}
```

Listing 3.4: Deserialization

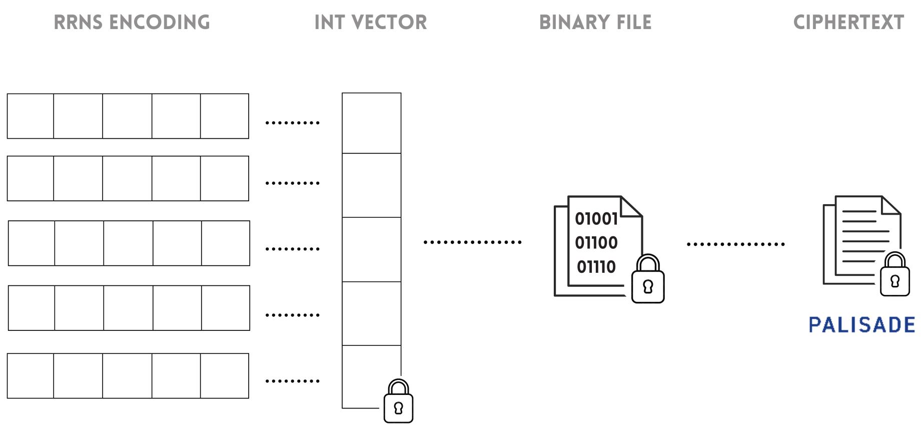


Figure 3.4: RRNS Decoding of a cipher

At this point, the aggregator node only needs to perform the arithmetic operations required by the scenario: summing distances between mobility points, saved as GPS coordinates. Once this result is obtained, it is sent to the cloud platform to be stored.
PALISADE makes it easy to act arithmetically on ciphertexts, as in the case of the above sum. In Listing 3.5, in fact there is the example of code needed in order to perform the sum between ciphertexts. Homomorphic addition of two `Ciphertexts` is performed by invoking the `EvalAdd` method of the `CryptoContext`. Both of the operands to the `EvalAdd` must have been created in the same `CryptoContext`, encrypted with the same `Key` and encoded in the same way for this operation to work [23].

```
    // homomorphic addition
    auto ciphertextResult = cryptoContext->EvalAdd(ciphertext1, ciphertext2);
```

Listing 3.5: Evaluation

In the example case of Listing 3.5, `ciphertextResult` thus contains the result of the sum between two ciphers, which, when decrypted by the cloud platform, will be equivalent to what the result would have been if the ciphers had been decrypted by the aggregator, summed, and re-encrypted for sending.
Once the data have made the journey within the network, been aggregated, they reach the end point that resides in the cloud platform, where they can be deciphered and further analyzed. Since encryption and decryption functionalities require a public key/private key pair [23], the platform has the private key with which initialize the decryption procedure. In Listing 3.6,

it is shown how the encryption can be removed, through the use of the secret key. A new object of type `Plaintext` is created, which will hold the decrypted result, and the `Decrypt` method is invoked on the `CryptoContext`.

```
// decrypt the result of the addition
Plaintext plaintextResult;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextResult, &plaintextResult);
```

Listing 3.6: Decryption

Chapter 4

EXPERIMENTS AND EVALUATIONS

This chapter will cover the simulations and actual tests carried out on the completed project, in order to evaluate its effectiveness, first and foremost, but also its execution efficiency.
The first part highlights the work done on the GeoLife [13] reference dataset, pre-processing executed in advance compared to the simulations themselves. In the second section, meanwhile, the performances of the designed program will be analyzed, including details of the execution times of the various network components and a comparison of the two encryption schemes employed.

4.1 Dataset Processing

The reference scenario concerns the management and manipulation of GPS coordinates, which delineate routes executed by users participating in the MCS application. The collection of such data was made possible through the use of the GeoLife dataset: a study sponsored by Microsoft Research Asia, on a sample of 182 users over the course of 3 years [13].

The extension devised to that project was to take the coordinates present in the dataset, preprocess them in a way that calculates the distances between the defined points, aggregating all these values and calculate the total path performed by the users.
Various technologies have been employed in order to implement such data processing: first and foremost the GeoPy library, which is a Python client for several popular geocoding web

services [14]. It turns out to be a comprehensive and extensive library for managing mobility data, allowing users to manipulate distances, points and coordinates expressed in latitude and longitude pairs or gradients. Listing 4.1 is a brief example of how it can be used to calculate the distance, given two points expressed as a coordinate pair. One imports the `distance` method from the library, defines the points of interest for calculation and simply invokes the function to obtain the space between them, with the auxiliary specification on the result format, which in the example case is in kilometers.

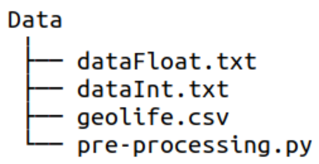
```python
from geopy import distance

newport = (41.49008, -71.312796)
cleveland = (41.499498, -81.695391)
print(distance.distance(newport, cleveland).km)

# 866.4554329098685
```

Listing 4.1: GeoPy Distance

The data pre-processing work, available at [3], was divided as shown in Figure 4.1. `geolife.csv` represents a sample portion of the GeoLife dataset, 769.040 coordinate pairs consisting of latitude and longitude were considered. A total of 765,041 distances were obtained from this set of points, seeing the coordinates based on the trajectories taken by the users surveyed. The source code for this calculation was included in the `pre-processing.py` file. Lastly, the two files `dataFloat.txt` and `dataInt.txt` contain the resulting values of the computation, then the set of distances to be sent to the network aggregator node, in floating point and integer format, respectively. This distinction was made in order to have two separate input files, which were then processed by the two main encryption schemes: the CKKS [7] for the approximate real arithmetic and the BGV [5] for the exact integer arithmetic. Having both schemes in place is interesting from a performance point of view, thus being able to compare them since they are based on the same dataset.



```
Data
├── dataFloat.txt
├── dataInt.txt
├── geolife.csv
└── pre-processing.py
```

Figure 4.1: Data Folder

Having to handle a huge collection of points, it became necessary to rely on a library suitable for the amount of work to be pre-processed: the choice fell on Pandas [24], which is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language [24]. With the help of Pandas, a dataset becomes an object of type `DataFrame`, such that it can be handled and modified at will, depending on the study application. Furthermore, Pandas supports the integration with many file formats or data sources out of the box (csv, excel, sql, json, parquet,. . .) [24], so as to adapt to any format in which one's data may be.
By combining both GeoPy and Pandas functionalities, the resulting code fragment is as in Listing 4.2, taken form `pre-processing.py`, containing the overall functions for reading and manipulating the dataset. The first step is to transform the contents of file `geolife.csv`, into an object of type `DataFrame`, using the function `read-csv`. The `header=None` specification implies not storing the names of the columns, but rather assigning increasing numeric values to them. The data reference table contains 5 columns, respectively divided into: the first two assign the latitude and longitude values of the stored point; the third is dedicated to a unique progressive numeric `user id`, to identify the subject involved; the fourth concerns an identifier for the trajectory and, as final column, there is a timestamp referring to the time and date of coordinates. Not all of this information is necessary for the purpose of the project, so the timestamp and user id columns are removed, via the `drop` function.
The idea of calculating distances, given a set of points, is performed by grouping by the trajectory identifier, which, regardless of the user, remains unique. Then, taking a trajectory, intended as a sampling moment, distance calculation is performed between pairs of its consecutive points, with values expressed in meters. The moment the trajectory changes, so its identifier is different, the calculation is resumed from the latter, forgetting the value of the previous one.

```python
from geopy.distance import distance
import pandas as pd

file = "./geolife.csv"
df = pd.read_csv(file, header=None)

# deletes uid and date_time columns
df.drop(columns=[2, 4], inplace=True)
df.drop(labels=0, axis=0, inplace=True)
# groups by trajectory column
df.groupby(3)
dist = []
dist_to_write = []

tid = df.loc[1, 3]
df = df.reset_index()

for idx, _ in df.iterrows():
    if (df.loc[idx, 3] == tid):
        lat = df.loc[idx, 0]
        lon = df.loc[idx, 1]
        dist.append(tuple((lat, lon)))
    else:
        tid = df.loc[idx, 3]
        # calculates the distances
        for i in range(0, len(dist)-1):
            d = distance(dist[i], dist[i+1]).m
            dist_to_write.append(d)
        dist.clear()
```

Listing 4.2: Dataset Processing [3]

For completeness, the function regarding writing to the respective result files, with distance values in real or approximate integer format, is inserted in Listing 4.3. `dist-to-write` represents the list of values that are to be written to the files: it is first saved to `dataFloat.txt` and then the values are approximated to the nearest integer, so that the `dataInt.txt` file can be completed. As an implementation choice, it was decided to store up to a maximum of the fifth significant digit after the decimal point, for floating point values.

```python
distance_int = "./dataInt.txt"
distance_float = "./dataFloat.txt"

def write (filename, l):
    with open(filename, 'w') as f:
        for item in l:
            f.write("%s\n" % item)

# distances float
dist_to_write = [round(num, 5) for num in dist_to_write]
write (distance_float, dist_to_write)

# distances int
dist_int = [round(num) for num in dist_to_write]
write (distance_int, dist_int)
```

Listing 4.3: Files writing [3]

4.2 Evaluations

The proposed project has several key points: focusing on a decentralized architecture, the parties involved can take participation in analysis according to very different metrics. Creating a project evaluation scheme is a complex goal: starting with the parameters from the cryptographic scheme, the number of devices involved in the network, to the very metrics by which to measure the model, each aspect may vary and thus give a different result. Based on these several outcomes, it can be concluded whether a project was successful or not.
The following analysis aims to examine the project according to two macro notions: effectiveness and efficiency, reporting accurate measurements on program execution time and highlighting whether or not the choice of parameters affected this metric.
The most important issue concerns the effectiveness of the proposed solution, i.e., being able to ensure that, despite the introduction of a over-structure, such as the one given by RRNS encoding, the system was still able to produce correct results, for aggregation operations. The aim was to ensure that the use this encoding remained a content-neutral operation with respect to the ciphers, since it was designed to insert redundancy. The first tests performed dealt with exactly this query: both encryption schemes, the approximate CKKS one and

the BGV, chosen for the exact integer arithmetic, were first run without the aid of RRNS encoding and the results were saved to a text file. By result one wants to mean the complete computation within the overall network: data are encrypted, encoded, aggregated and finally decrypted. The tests were then rerun, with the activation of the encoding through the use of a Boolean flag, storing also these outcomes.

To prove that the use of PALISADE has not been violated, those files were compared: the decrypted values of the distance sums, with and without the redundancy given by the RRNS, must be equivalent. This task concerns the two simple functions inserted in Listing 4.4: they read and compare the two files, through the help of iterators on char values. The moment the values differ, execution is immediately aborted, reporting a negative result, emphasizing that the files do not match. Such meticulous control is successful on both adopted schemes. The BGV integer arithmetic scheme presents identical files, meanwhile the floating point arithmetic one leaves a margin of error on distances that do not reach the meter, so zero values. This is a scheme that by definition contains approximations on values, so this error is possible on the n-th digit after the decimal point, depending on the approximation factor decided for the model, which is however irrelevant for the purpose of calculating distances, since the sums return the correct result.

The effectiveness control was not as naive as one might think: from a theoretical point of view, the RRNS encoding and decoding operations remain content-neutral, since the Chinese Remainder Theorem makes it possible to restore the original numerical positional value. From a more concrete point of view, however, the work had to be concentrated on reading and writing of binary files, containing the ciphers, especially on the reinterpretation of data types when reading. PALISADE allows the encoding of `Plaintext` only to 64bit integers, or 32bit double elements, so type conversions when reading binary files must be in accordance with these constraints.

```
template<typename InputIterator1, typename InputIterator2>
bool range_equal(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2) {
    while(first1 != last1 && first2 != last2) {
        if(*first1 != *first2) return false;
        ++first1;
        ++first2;
    }
    return (first1 == last1) && (first2 == last2);
}

bool compare_files(const string& filename1, const string& filename2) {
    ifstream file1(filename1);
    ifstream file2(filename2);

    istreambuf_iterator<char> begin1(file1);
    istreambuf_iterator<char> begin2(file2);
    istreambuf_iterator<char> end;

    return range_equal(begin1, end, begin2, end);
}
```

Listing 4.4: Comparing Outcomes

Having attested that the designed architecture correctly performs and executes the sum between the pre-calculated distances, whether left approximated with floating points or rounded to an integer value, the second phase of the analysis concerns the correlation between the choice of various parameters and the execution time.
Before going into the details of the choice of parameters, it is necessary to point out what is meant by execution time, in fact, there is no single metric for such calculation: there is the wall time, which is the time that elapses between computations, measurable with a stopwatch, and CPU time, which concerns the period of time the CPU remains blocked performing operations, quantifiable as the number of clock cycles. These values are to be analyzed together: wall time alone is not enough to shed light on hypothetical performance problems, because it is not clear which operation keeps the program most employed. On the other hand, CPU time alone is concerned with mere computational time, so any I/O bounded programs, slowed down by the consistent waiting for input and output operations, would not be reported.
In order to measure different execution times, two libraries were chosen which are part of the standard one for C++: Chrono [28], which is included from version 11 of the C++ compiler and allows to measure the wall time, having numerous clocks in the machine, and CTime [29], available for both Linux and Windows systems, allows to measure the CPU time of the program. The primarily used function of the latter is `clock()`, which returns the number of clock it took the processor to perform the operations.

The examples given in Listing 4.5 are part of a study on measuring execution time in C++ programs, reported in [10]. Using the libraries altogether comes very easy: both allow one to create an object to which assign the measurement start value, isolate the code to be evaluated and, lastly, run a difference to obtain the elapsed time result.

```
#include <chrono>
auto begin = std::chrono::high_resolution_clock::now();
// ...
auto end = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);

#include <time.h>
clock_t start = clock();
// ...
clock_t end = clock();
double elapsed = double(end - start)/CLOCKS_PER_SEC;
```

Listing 4.5: Example Time Libraries

Once the measurement metrics have been defined, it is necessary to specify the main parameters on which the architecture is based, which can affect execution times:

- Multiplicative depth: it represents the maximum value of multiplications supported for the scheme, to perform the correct decryption. As mentioned earlier, this value affects the size of the resulting cipher, because it increases the size of the vectors of polynomials, so it stands to reason that the computational time for them will also increase.
- Chunk size: instead of processing the entire data set at the same time, which is not likely in real applications, it is possible to divide it into sections, the size of which may vary, and process the values in the SIMD mode enabled by PALISADE. This size simulates the simultaneous set of distances that the devices send to the aggregator: this value does not change the size of the ciphers, but possibly the performance time to process them.
- Plaintext modulus: it is represented by a prime number, whose task is to define the modulus for arithmetic operations, thus intrinsically a limit on the values the integers can take. For this reason, it does not change the size of the cipher. In the case of exact integer arithmetic, the plaintext modulus p should be large enough to avoid an overflow [23]. PALISADE's authors suggest a default value of $p = 65537$. If one wants to change

it, the constraint is that, taken m, the degree of the cyclotomic polynomial, $(p-1)/m$ must be integer.

- Scale factor: used only for the approximate CKKS scheme, it represents the degree of precision required after the decimal point. Usually defined as $\triangle = 2^s$. CKKS "erases" 12-25 least significant bits (depending on the multiplicative depth), and each addition and multiplication may consume up to 1 bit. The value of s for desired precision v is something like $s \approx v + 20$. It can be adjusted as needed, depending on the multiplicative depth of the computation [23]. This value is multiplied, during the construction of the `Plaintext`, with each value of the input vector, but it does not change the size of the cipher.

- RNS base size: referring to the number of modules it contains, bearing in mind that the proposed design assumes redundant ones, i.e. more than those needed for representation. Changing the number of modules also changes the number of comparisons and decodings required for the project, when sending and receiving ciphers, but it does not affect their sizes.

The analyses were performed on Ubuntu 21.10, 64-bit, AMD® Ryzen 7 5700u processor with RADEON graphics×16, 15GB RAM, AMD® Renoir graphic card. These evaluations were recorded for both the BGV integer arithmetic scheme and the CKKS scheme, with approximate arithmetic. For interpreting data more clearly:

- Sending time: it is the time it takes the users' devices to encrypt the data, insert the RRNS encoding, then send them to the aggregator.

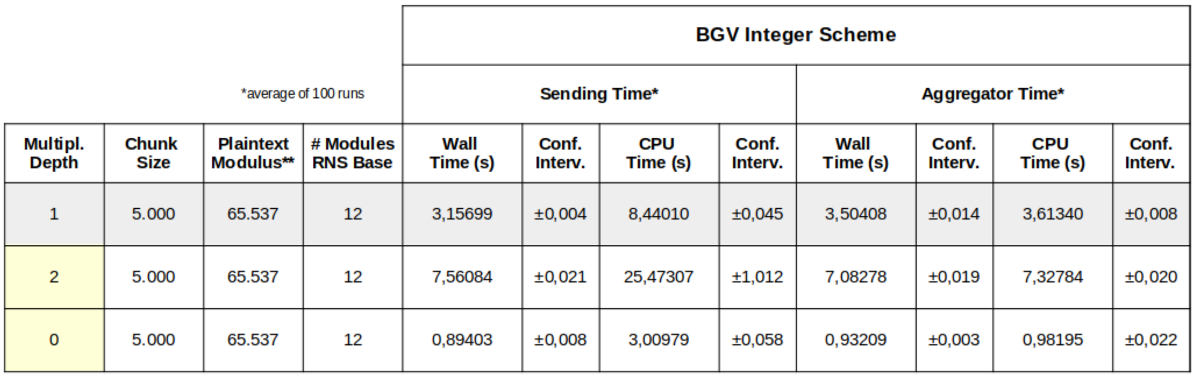
- Aggregator time: it refers to the time it takes the aggregator to remove the RRNS encoding and aggregate the data. The time taken to send data from the aggregator to the cloud platform will not be considered.

- Wall time: it is the time interval to execute the related operation. Average value of 100 runs.

- CPU time: it is the time in which the CPU was occupied. During this measurement, any auxiliary services running on the machine was disabled. Average value of 100 runs.

- Size of the dataset: 765.041 pre-calculated distances.

- Confidence Interval: it represents the estimated range of values associated with the measurements. The confidence level is the overall capture rate if the method is used many times [20]. A 95% confidence interval is assumed.
- Default run: in order to have a standard of comparison, the tests were first run with default values, considering the directions of the PALISADE's authors and the guidelines in [1]. In the figures below, it is indicated with the color gray.
- Best run: such execution refers to the test performed once the parameters assume their most efficient values. It is indicated by the color green.

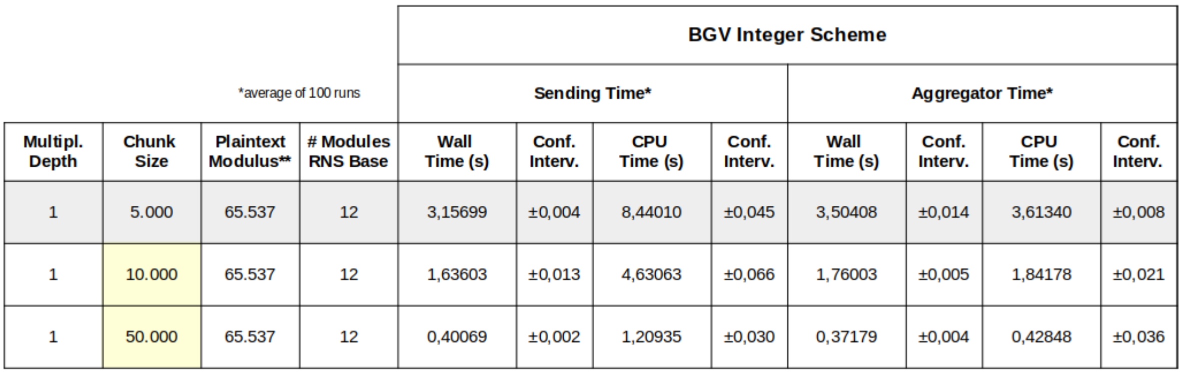


				BGV Integer Scheme							
		*average of 100 runs		**Sending Time***				**Aggregator Time***			
Multipl. Depth	**Chunk Size**	**Plaintext Modulus****	**# Modules RNS Base**	**Wall Time (s)**	**Conf. Interv.**	**CPU Time (s)**	**Conf. Interv.**	**Wall Time (s)**	**Conf. Interv.**	**CPU Time (s)**	**Conf. Interv.**
1	5.000	65.537	12	3,15699	±0,004	8,44010	±0,045	3,50408	±0,014	3,61340	±0,008
2	5.000	65.537	12	7,56084	±0,021	25,47307	±1,012	7,08278	±0,019	7,32784	±0,020
0	5.000	65.537	12	0,89403	±0,008	3,00979	±0,058	0,93209	±0,003	0,98195	±0,022

Figure 4.2: BGV Scheme and Multiplicative Depth

Figure 4.2 shows the comparison of the default run in contrast to the change in multiplicative depth, in the BGV exact integer arithmetic scheme. Processing 5.000 distances simultaneously, with a unitary multiplicative depth, a module limiting integers set to 65.537 and 12 modules in the base, 4 of which are redundant, allows a runtime on 3 seconds average. While this turns out to be a considerable finding, the most important result to note concerns the CPU execution time: it averages around 8 seconds. This result indicates a computational effort by the end-users' devices, probably caused by the reading and writing of the ciphers, performed in order to derive their integer representation from the binary files. In contrast, the aggregator in the default run has a more balanced execution: wall time and CPU time have a remarkably small margin, emphasizing less computational effort.
One immediately notices how increasing the multiplicative depth of the scheme, thus the size of each cipher, leads the entire performance plan to deteriorate dramatically. The cause of this behavior must again be found in the processing phase of the ciphers themselves: increasing their ability to perform multiplicative operations embellishes the cryptographic scheme,

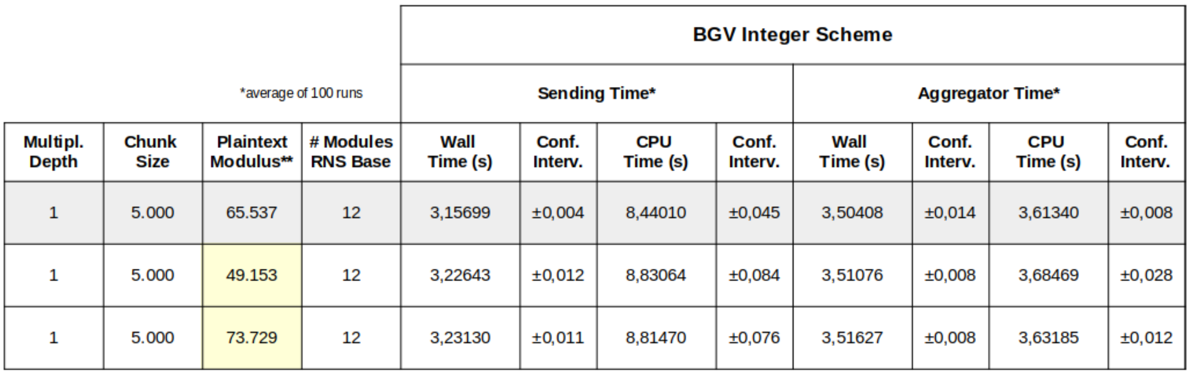
complicating their algebraic structure, but making their manipulation more expensive. However, for the scenario studied this result is not a concern: the aggregator node does not need to perform homomorphic multiplications on the data, so increasing this parameter is irrelevant. For this reason the last tests were introduced, including the behaviour of the system when the multiplicative depth is set to zero: the improvements achieved are significant, with results even more than halved.



				BGV Integer Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Plaintext Modulus**	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	65.537	12	3,15699	±0,004	8,44010	±0,045	3,50408	±0,014	3,61340	±0,008
1	10.000	65.537	12	1,63603	±0,013	4,63063	±0,066	1,76003	±0,005	1,84178	±0,021
1	50.000	65.537	12	0,40069	±0,002	1,20935	±0,030	0,37179	±0,004	0,42848	±0,036

Figure 4.3: BGV Scheme and Chunk Size

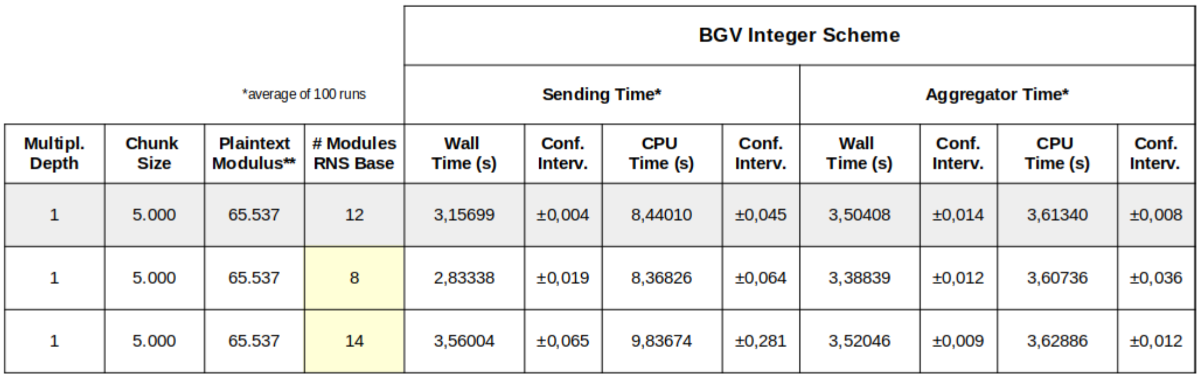
The number of distances to be packed together for encryption, i.e., the chunk size, shown in Figure 4.3, turns out to be a prominent parameter. When doubled from the default run, it leads the system to reduce the execution time by half, both for the user node and the aggregator. The reason for this success lies in PALISADE's ability to create objects of type `Plaintext` from joint data, exploiting a Single Instruction Multiple Data methodology. Having multiple grouped distances on which to perform operations turns out to be advantageous, as these are done simultaneously.



				BGV Integer Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Plaintext Modulus**	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	65.537	12	3,15699	±0,004	8,44010	±0,045	3,50408	±0,014	3,61340	±0,008
1	5.000	49.153	12	3,22643	±0,012	8,83064	±0,084	3,51076	±0,008	3,68469	±0,028
1	5.000	73.729	12	3,23130	±0,011	8,81470	±0,076	3,51627	±0,008	3,63185	±0,012

Figure 4.4: BGV Scheme and Plaintext Modulus

Figure 4.4 shows that the plaintext modulus is an almost neutral parameter in comparison with the system performance. This result should not be surprising, however, because this value affects the content of the ciphers, seen as a limit to the integer values with which it is constructed, but not its form. Thus modification operations on them are not conditioned by this parameter.

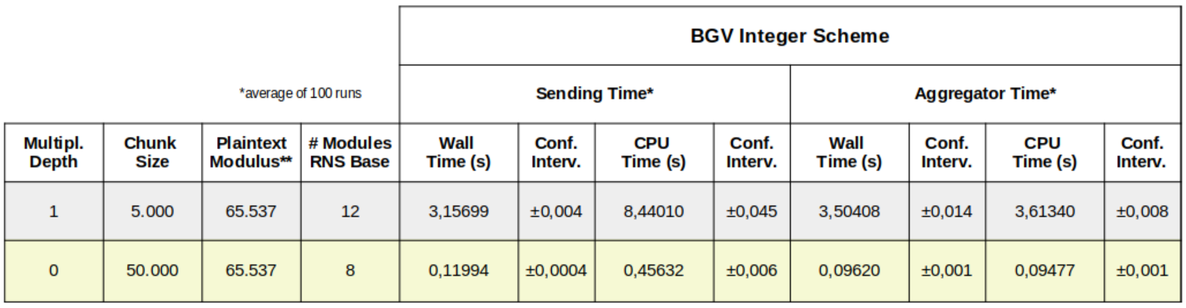


				BGV Integer Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Plaintext Modulus**	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	65.537	12	3,15699	±0,004	8,44010	±0,045	3,50408	±0,014	3,61340	±0,008
1	5.000	65.537	8	2,83338	±0,019	8,36826	±0,064	3,38839	±0,012	3,60736	±0,036
1	5.000	65.537	14	3,56004	±0,065	9,83674	±0,281	3,52046	±0,009	3,62886	±0,012

Figure 4.5: BGV Scheme and RNS Base Size

A slight improvement can be achieved, however, by decreasing the number of modules in the RNS base. In fact, as shown in Figure 4.5, performance improves by nearly half a second on all nodes in the network. Having fewer values with which to encode, and then later decode, ciphers implies having shorter cycles of operations and thus a gain in performance, while having more increases the execution time slightly.

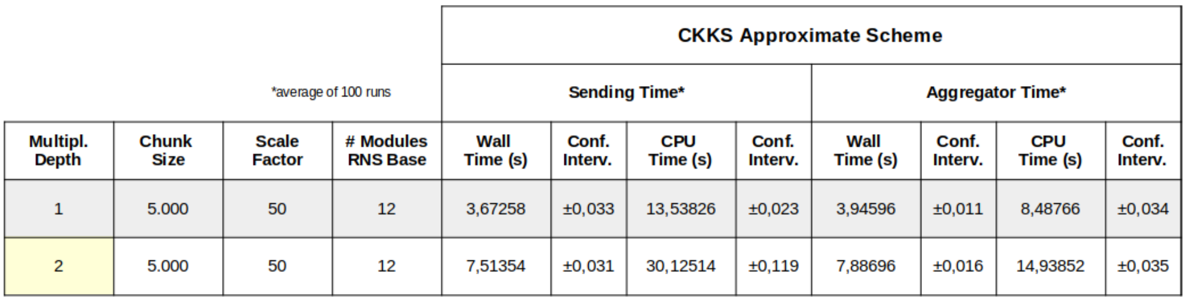
After the analysis just completed on parameters, one is able to construct a last run, defined as the best one, to understand how the system behaves in the best-case scenario. This result is stored in Figure 4.6. Comparing the default starting run, with the best processed one, the results are remarkable. The totality of execution times, along the entire network, are almost cleared, including the occupancy of the computing CPU.



				BGV Integer Scheme							
			*average of 100 runs	Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Plaintext Modulus**	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	65.537	12	3,15699	±0,004	8,44010	±0,045	3,50408	±0,014	3,61340	±0,008
0	50.000	65.537	8	0,11994	±0,0004	0,45632	±0,006	0,09620	±0,001	0,09477	±0,001

Figure 4.6: BGV Scheme Best Run

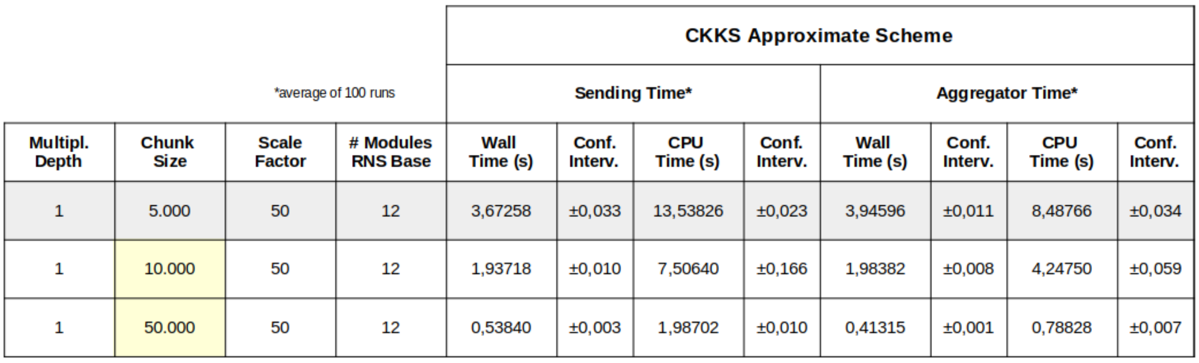
The same performance analysis carried out on the integer model is also conducted on the CKKS scheme, which is responsible for the approximate floating point arithmetic. Commenting on the default run, shown in Figure 4.7, it is composed as follows: again considering 5.000 distances at a time, with a unitary multiplicative depth, the scaling factor set to 50, indicating a saving accuracy of 1/50, 12 modules in the RNS base, it returns an execution time of approximately 3.6 seconds. This result is in line with the default run on the BGV scheme, with a small deterioration in timing, but justifiable by the introduction of a complexity factor in the scheme itself. However, a dramatic difference in CPU computational time of the nodes is obtained: about 13 seconds in terms of calculation time. The reason for this deterioration is again to be found in the use of the approximate arithmetic: with an accuracy threshold that allows for extremely precise values, but numerous digits after the decimal point have to be memorised, written into the binary files, reinterpreted and finally recomposed. In the sending phase, the computational time becomes considerably high, possibly becoming the system's bottle neck.



				CKKS Approximate Scheme							
			*average of 100 runs	Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Scale Factor	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	50	12	3,67258	±0,033	13,53826	±0,023	3,94596	±0,011	8,48766	±0,034
2	5.000	50	12	7,51354	±0,031	30,12514	±0,119	7,88696	±0,016	14,93852	±0,035

Figure 4.7: CKKS Scheme and Multiplicative Depth

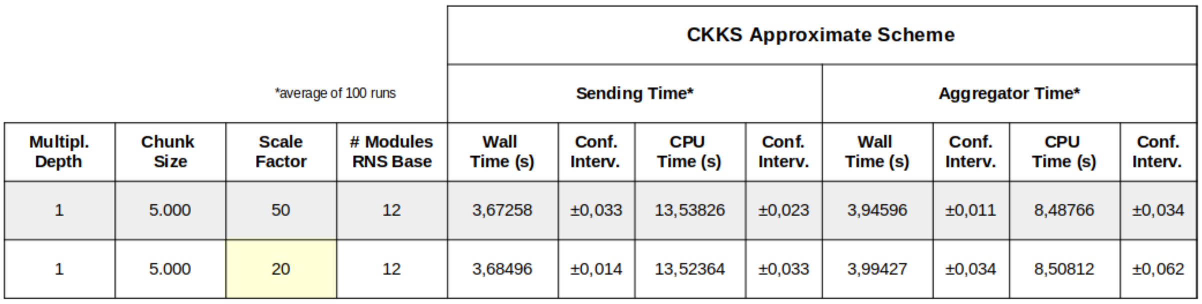
In the second run, still in Figure 4.7, when increasing the multiplicative depth, the results again are not surprising: there is a drastic deterioration in performance, in line on all the nodes of the network. The ciphers are doubled in size, through the ability to handle more homomorphic operations now, but this advantage is paid for computationally. This improvement in the scheme does not lead to a greater accuracy in calculating the sum of distances, so there is no need to change the multiplicative depth parameter. Unlike the integer BGV scheme, CKKS does not allow the multiplicative depth to be set to zero, due to problems with the representation of numbers in approximate arithmetic, therefore the minimum value remains the unit one.



				CKKS Approximate Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Scale Factor	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	50	12	3,67258	±0,033	13,53826	±0,023	3,94596	±0,011	8,48766	±0,034
1	10.000	50	12	1,93718	±0,010	7,50640	±0,166	1,98382	±0,008	4,24750	±0,059
1	50.000	50	12	0,53840	±0,003	1,98702	±0,010	0,41315	±0,001	0,78828	±0,007

Figure 4.8: CKKS Scheme and Chunk Size

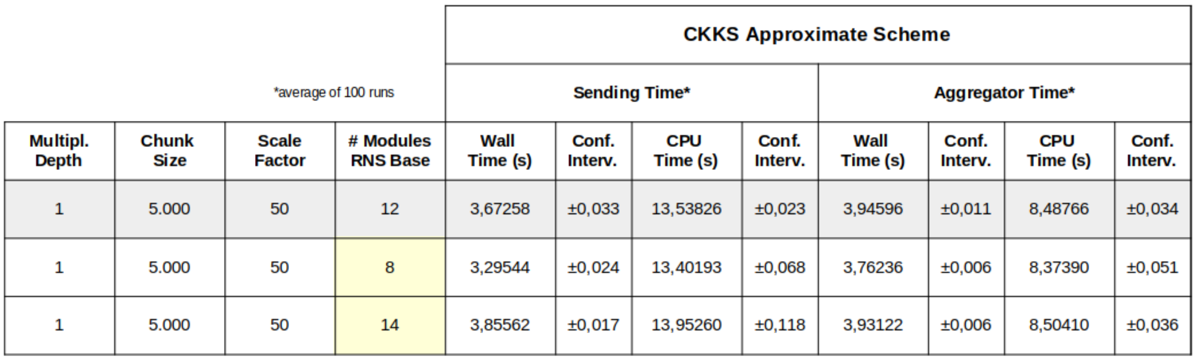
The system responded extremely positively, as the Figure 4.8 shows, to increasing the chunk size, i.e. to increasing the portion of the dataset to be packed into a single object `Plaintext`, processed simultaneously. Doubling this value, from 5.000 distances in the default run to 10.000 in the second test run, led to execution times being more than halved. With this change, it was also eventually possible to lighten the CPU's computational load, resulting in an average of 7 seconds. Clearly, when this value is increased further, up to 50.000 distances, execution times even drop below one second.



				CKKS Approximate Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Scale Factor	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	50	12	3,67258	±0,033	13,53826	±0,023	3,94596	±0,011	8,48766	±0,034
1	5.000	20	12	3,68496	±0,014	13,52364	±0,033	3,99427	±0,034	8,50812	±0,062

Figure 4.9: CKKS Scheme and Scale Factor

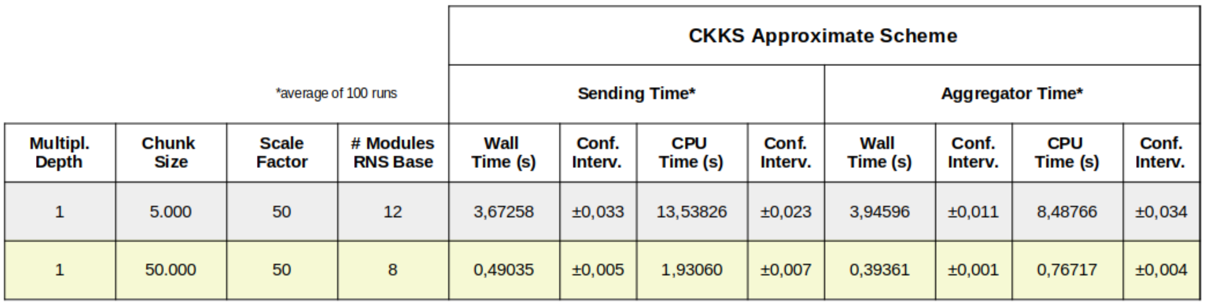
Figure 4.9 involves changing the scaling factor of the scheme. The reported values show that in reality this parameter does not significantly affect the execution time. The reason can be found in the fact that it allows the content of the ciphers to be changed, in the number of accuracy values saved, but not in their structure. Operations such as encoding and decoding in RRNS are not burdened by this.



				CKKS Approximate Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Scale Factor	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	50	12	3,67258	±0,033	13,53826	±0,023	3,94596	±0,011	8,48766	±0,034
1	5.000	50	8	3,29544	±0,024	13,40193	±0,068	3,76236	±0,006	8,37390	±0,051
1	5.000	50	14	3,85562	±0,017	13,95260	±0,118	3,93122	±0,006	8,50410	±0,036

Figure 4.10: CKKS Scheme and RNS Base Size

The number of modules in the RNS base, however, is able to provide a very limited easing on the wall time, as shown in Figure 4.10. Eliminating 4 modules, from 12 in the default run to 8 in the second test run, results in fewer comparisons during both encoding and decoding, reducing the computational cost. Trend therefore confirmed when, in the third run, this factor is increased, from 12 to 14 modules, and the times in fact increase accordingly.



				CKKS Approximate Scheme							
		*average of 100 runs		Sending Time*				Aggregator Time*			
Multipl. Depth	Chunk Size	Scale Factor	# Modules RNS Base	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.	Wall Time (s)	Conf. Interv.	CPU Time (s)	Conf. Interv.
1	5.000	50	12	3,67258	±0,033	13,53826	±0,023	3,94596	±0,011	8,48766	±0,034
1	50.000	50	8	0,49035	±0,005	1,93060	±0,007	0,39361	±0,001	0,76717	±0,004

Figure 4.11: CKKS Scheme Best Run

As a last example run, the performance with the best parameters in terms of execution time is shown in the last line, thus constructing the best run shown in Figure 4.11. These parameters continue to guarantee the correctness analysis shown earlier: the results are still consistent and gain renewed redundancy through the use of RRNS. By refining the choice of parameters, it was possible to optimise the efficiency analysis, halving the performance time. Just under half a second, 765.041 distances can be sent from user nodes to the aggregator. In this window of time, not only is the actual sending carried out, but also the ciphers' manipulation, binary transformation and RRNS encoding.

When compared, the two analyzed schemes show altogether similar performance times, despite profound implementation differences. Certainly the distinctive result lies in the CPU calculation time, as far as the CKKS scheme is concerned. In the best run it is considerably reduced, thus lightening the load on the network sensors, although it presents a higher result than the BGV scheme. This conclusion is in line with the characteristics of the two schemes and their implementations. As the CKKS has to maintain information regarding the accuracy and approximation levels of the underlying data, it presents a more articulated structure: values intrinsically have a scaling factor that must be kept correct in order to guarantee proper decryption. It must be taken into account that, during homomorphic operations, this parameter fluctuates, so there are also rescaling operations by the scheme itself, which are carried out during the computation and can therefore burden the processing time.

Chapter 5

CONCLUSIONS

The project presented was based on the idea of building an architecture model, in an MCS context, capable of aggregating data in a secure manner. The centre of interest lies in the promise of confidentiality, that can be granted to those users who want to participate in the data collection campaign. To address this need, homomorphic encryption was devised, i.e. a special type of encryption that allows the use of aggregation operations directly on encrypted data, without the need to process it in the clear. Despite it being employed mainly in data storage applications, such as secure computation on databases and data analysis on cloud, it seemed a natural extension to adapt it to a decentralised context such as MCS.

The developed structure consists of several entities: mobile devices, that collect sensitive data from users, a central node that is able to securely aggregate them and, lastly, a storage node such as the cloud platform itself. On top of this structure, an additional layer was added, which is the redundancy when sending data, from the devices to the aggregator, given by using RRNS encoding. The latter is mainly oriented towards safeguarding mobile devices, as the redundancy allows the model to recover possible lost packets in the network, without resorting to re-transmissions, which would consume both network bandwidth and energy.

The technologies studied were applied to a concrete scenario of manipulating GPS coordinates, in order to calculate the sum of the distances travelled by the participants. PALISADE was used as a homomorphic encryption library in order to perform this sum in a secure manner. It allows any current homomorphic scheme to be implemented, so in order to also have a comparison metric between models, it was decided to test the architecture with both the integer BGV scheme and the approximate CKKS one. On both implementations, it was necessary to make refinements in the choice of parameters, before achieving competitive execution times, while maintaining the same load of distances, i.e. 765.041. Despite the refinements, and the large improvements reported, the CKKS approximate scheme remains computationally heavier than the integer one, both in terms of execution time and CPU load. Having an approximate arithmetic implies having a scaling factor that fluctuates during execution, so it is necessary to periodically perform rescaling operations in order not to exceed the tolerance threshold, after which the decryption may suffer significant alterations.

Both schemes turned out to be correct in terms of calculation: the sums reported are in line with the values that would have been obtained in the clear.

However, the innovation of secure homomorphic aggregations is not computationally unpaid for: the use of this encryption is considerable, in terms of low-level load for the devices employing it. It is therefore necessary to use sensors with high computing power, such as smartphones, and it may not be possible on IoT devices with low computing performances. It is also pointed out that, depending on the choice of the multiplicative depth and security level, the size of the ciphers changes, potentially increasing the computational load as well. This result, in the state of the art, does not have to be a limitation: the use of this type of encryption is justified when it is necessary to guarantee secure aggregations of sensitive data. If the data handled, indeed need to be protected, but do not contain sensitive information, as the example with temperature sensors, there is no need to pay the high cost of homomorphic encryption at the sending stage. For the latter case, it might be sufficient to build a cryptographic architecture in the network and, only at storage point, use more advanced protection with homomorphic encryption, since the cloud platform can afford an higher computational cost.

A further observation concerns the use of cryptographic libraries: the reference scenario was implemented using PALISADE. It may be the object of future work to analyze how different libraries encrypt data, construct operations and behave computationally. This comparison would be useful at the time of implementation, in order to orientate on which library is the most suitable according to the needs of the scenario.

A final note, regarding possible extensions of the present project, may be to succeed in fully combining the use of HE with the RRNS methodology, in order to exploit the inherent parallelism of the latter, to perform aggregation operations on the encrypted fragments of the cipher, instead of having to reconstruct the encrypted data.

Bibliography

[1] Martin Albrecht et al. *Homomorphic Encryption Security Standard.* Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018. URL: https://homomorphicencryption.org/introduction/.

[2] A. A. Badawi et al. "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme". In: (2019). URL: https://eprint.iacr.org/2018/589.pdf.

[3] C. Boni. *Data aggregation using Homomorphic Encryption in Mobile CrowdSensing context.* 2022. URL: https://github.com/ChiaraBn/Master-Thesis.

[4] H. Bozduman and E. Afacan. *Simulation of a Homomorphic Encryption System.* 2016. ISBN: 2444-8656. URL: https://sciendo.com/journal/AMNS.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. "Fully Homomorphic Encryption without Bootstrapping". In: (2011). URL: https://eprint.iacr.org/2011/277.pdf.

[6] A. Carey. "On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries". In: (2020). URL: https://scholarworks.uark.edu/csceuht/77/.

[7] J. H. Cheon et al. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: (2017). URL: https://link.springer.com/chapter/10.1007/978-3-319-70694-8_15.

[8] R. C. Daileda. "The Chinese Remainder Theorem". In: (2018). URL: http://ramanujan.math.trinity.edu/rdaileda/teach/s18/m3341/CRT.pdf.

[9] J. Fan and F. Vercauteren. "Somewhat practical fully homomorphic encryption". In: Cryptology ePrint Archive, Report 2012/144 (2012). URL: http://eprint.iacr.org/2012/144.

[10] C. Ferreira. *8 Ways to Measure Execution Time in C/C++*. 2020. URL: https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9.

[11] R. K. Ganti, F. Ye, and H. Lei. "Mobile Crowdsensing: Current State and Future Challenges". In: (2011). URL: https://ieeexplore.ieee.org/document/6069707.

[12] C. Gentry. "A Fully Homomorphic Encryption Scheme". In: Ph.D. dissertation, Stanford University (2009). URL: https://crypto.stanford.edu/craig/craig-thesis.pdf.

[13] *GeoLife Dataset Microsoft*. 2012. URL: https://www.microsoft.com/en-us/download/details.aspx?id=52367.

[14] *GeoPy*. 2018. URL: https://geopy.readthedocs.io/en/stable/.

[15] M. Girolami et al. "How Mobility and Sociality Reshape the Context: A Decade of Experience in Mobile CrowdSensing". In: (2021). URL: https://www.mdpi.com/1424-8220/21/19/6397.

[16] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. *New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks*. Cryptology ePrint Archive, Paper 2022/425. https://eprint.iacr.org/2022/425. 2022. URL: https://eprint.iacr.org/2022/425.

[17] W. S. Grant and R. Voorhies. *cereal - A C++11 library for serialization*. 2017. URL: https://uscilab.github.io/cereal/index.html.

[18] S. Hardy. *A Homomorphic Encryption Illustrated Primer*. 2021. URL: https://humanata.com/blog/illustrated_primer/.

[19] C. Mahapatra et al. "A Reliable and Energy Efficient IoT Data Transmission Scheme for Smart Cities based on Redundant Residue based Error Correction Coding". In: (2015). URL: https://www.researchgate.net/publication/308546259_A_reliable_and_energy_efficient_IoT_data_transmission_scheme_for_smart_cities_based_on_redundant_residue_based_error_correction_coding.

[20] D. S. Moore, W. I Notz, and M. A. Flinger. "The basic practice of statistics (6^{th} ed.)" In: New York, NY: W. H. Freeman and Company (2013). URL: https://www.macmillanlearning.com/college/ca/product/The-Basic-Practice-of-Statistics/p/1319244378.

[21] A. Nannarelli and M. Re. "Residue Number Systems: a Survey". In: (2008). URL: https://backend.orbit.dtu.dk/ws/portalfiles/portal/3350177/tr08_04.pdf.

[22] *PALISADE Homomorphic Encryption Software Library.* 2021. URL: https://gitlab.com/palisade/palisade-release/blob/master/doc/palisade_manual.pdf.

[23] *PALISADE Manual.* 2021. URL: https://palisade-crypto.org/.

[24] *Pandas.* 2008. URL: https://pandas.pydata.org.

[25] B. Porter. "Cyclotomic Polynomials". In: (2015). URL: https://www.whitman.edu/Documents/Academics/Mathematics/2015/Final%20Project%20-%20Porter,%20Brett.pdf.

[26] O. Regev. *On lattices, learning with errors, random linear codes, and cryptography.* 2009. URL: https://dl.acm.org/doi/10.1145/1568318.1568324.

[27] D. Schoinianakis. "Residue arithmetic systems in cryptography: a survey on modern security applications". In: (2020). URL: https://link.springer.com/article/10.1007/s13389-020-00231-w.

[28] *std Chrono Library.* URL: https://www.cplusplus.com/reference/chrono/.

[29] *std Time Library.* URL: https://www.cplusplus.com/reference/ctime/.

[30] Y. Wang et al. "Privacy protection in mobile crowd sensing: a survey". In: (2019). URL: https://link.springer.com/article/10.1007/s11280-019-00745-2.

[31] L. Yang and L. Hanzo. "REDUNDANT RESIDUE NUMBER SYSTEM BASED ERROR CORRECTION CODES". In: (2001). URL: https://eprints.soton.ac.uk/257134/1/lly-lh-vtcfall-2001-2.pdf.