Faculty of Science and Technology, Department of Computer Science

Detection of Wheezes and Breathing Phases using Deep Convolutional Neural Networks

—

Johan Fredrik Eggen Ravn

INF-3981, Master's thesis in computer science, June 2017

Abstract

A Fully Homomorphic Encryption (FHE) scheme allows you to perform an arbitrary number of computation on encrypted data without decrypting it first. This is a very useful feature as it allows us to perform computations on data without knowing the data itself. Computations on sensitive data can be done in an untrusted environment without compromising privacy. Even though this may sound like the holy grail of cryptography which could solve the worlds IT problems when it comes to security and trust, it have some issues. Unfortunately Fully Homomorphic Encryption(FHE) libraries are very limited in Python. In this paper we have implemented a Python API based of a FHE C++ library called HElib and we will take a look at potential usages and limitations to the library.

Acknowledgements

Morbi sit amet diam condimentum, posuere tellus ut, fringilla massa. Nulla viverra dolor leo, vel cursus erat ornare non. Phasellus vehicula velit eget posuere rutrum. Integer at egestas enim, sed vulputate sapien. Nunc odio metus, mattis in erat at, rhoncus venenatis tortor. Vivamus porttitor molestie risus, vel gravida leo lacinia ac. Phasellus efficitur vehicula risus ut ornare. Pellentesque tincidunt libero ac massa finibus, sed hendrerit nisl pellentesque. Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Contents

List of Figures

List of Tables

Introduction

When dealing with sensitive data we need to make sure that the data is secure. This means, if the data is leaked due to a security breach, error in the system or any other reasons, we need to make sure the data are unrecognizable for anyone who is not authorized. This is commonly solved by using an encryption algorithm to encrypt the data, making it unreadable to anyone who do not know the private key used during the encryption. As more and more businesses and organizations are moving away from having a their own dedicated server to renting server space in large server parks, like a cloud, security are becoming a bigger concern than ever before. Traditionally businesses and organizations would have their own private server which would handle computation and storing of the data. The only people having direct access to the server would be authorized personnel hired by the business or the organization. Meaning the only real threat from a potential attacker would come from outside the business/organization, as any attacks from inside the business/organization would be met with prosecution and lawsuit.

As the trend of cloud services have exploded in the last years this have changed. We no longer have control over the physical location of the server, and we now have to trust the cloud service to uphold access control agreed upon. When storing data on cloud servers we are faced with not only the same security threats as on the private server, but also additional threats specific for cloud computing, like side channel attacks[12], virtualization vulnerabilities and abuse of cloud services. Cloud administrators or unauthorized users might access the data and obtain sensitive information for various motivations. Because of the increased threat when storing data in a cloud environment the data need to be encrypted at all times with our own encryption key, as even the cloud service cannot be trusted.

If we use the cloud service to only store data, then this is pretty simple, but what if we want to do some computation on the data stored? One way to do this is to encrypt the sensitive data and store this on the cloud. To perform a computation the data would be downloaded to a private server and decrypted. The computation is performed and the data is encrypted and uploaded back to the cloud. The problem with this approach, it's inefficient, especially when a lot of computation is done and it also requires a private server to perform the computation. Another approach is to use a Homomorphic Encryption(HE) scheme which enables us to perform computation on encrypted data without decrypting it first. This means, we can perform computation on ciphertexts generating an encrypted result which, when decrypted, matches the result of operations performed on the plaintext. By using Homomorphic Encryption we can perform the computation without ever exposing the data which means we can ensure that the data is secure at all times during the computation.

Why do we want Homomorphic Encryption? As shown in the previous example it allows us to do computation on our data without ever exposing the data during the computation. This essentially means that we can now export the computation work to a third party like a cloud service. We no longer have to pay for expensive servers to perform heavy computations, we can export the job an external server which is much cheaper in term of power consumption, server maintenance and bandwidth cost. This also allows us to pay for exactly the amount of storage and computational power needed, which can dynamically increased when needed. But is also have other benefits like hospital record can be examined without compromising patient privacy and financial data can be analyzed without opening it up to theft.

If Homomorphic Encryption is so great why are not everyone using it already? Even though there are many benefits which comes with using Homomorphic Encryption, there are some issues. First of all, performance. The first Fully Homomorphic Encryption scheme by Craig Gentry[1] was taking 100 trillion times as long to perform computations on encrypted data than plaintext computation. In this paper we will take a look at a FHE library called HElib which is an open source library in C++. With Boost Python we have created a Python API of this library and will use this to see how far the FHE schemes have come and look at practical usages and/or which challenges still remain for FHE schemes.

Background

2.1 Homomorphic encryption

"The method used in the calculation of encrypted information without the need for decryption is known as homomorphic encryption" [1]

For instance, a search engine that is homomorphically encrypted can take encrypted search terms and compare it with an encrypted index. A financial database that is homomorphically encrypted can allow the users to inquire the amount of money that an employee earned in the past. The input of employee name would be encrypted and so would the result given. This avoids any problems in privacy that is seen in online services that deal with sensitive data. Notably, many encryption schemes enable homomorphic encryption that is partial like RSA, Paillier, Benaloh, ElGamal. Partial Homomorphic Encryption implies that the scheme have some homomorphic properties like addition or multiplication but not both.

Homomorphic in mathematics explains the shift of one data set into another and at the same time, there is the preservation of relationships between the elements in both sets [2]. In modern cryptography, fully homomorphic encryption is among the holy grails. It is a form of encryption that enables the assessment of arbitrarily complex programs on encrypted data[3]. Rivest Adleman together with Dertouzos are the ones who suggested this problem in 1978 [4]. Late on thirty years later there was the first plausible fully homomorphic encryption candidate. This finding or rather discovery was made possible by Gentry's work [5] [6].

The main building block in Gentry's solution was based on the complexity of the problems on ideal lattices [3]. Lattices have become standard fare when it comes to cryptography. Very little is known about the ideal lattices although the lattices problems have been well researched and studied. In order to build fully homomorphic encryption, the ideals are a mathematical object to be utilized. This is due to the fact that they have support for multiplication and addition. It is significant to note that all the fully homomorphic encryptions rely on the ideals found in various rings. Additionally, looking at the discovery of Gentry in the attainment of full homomorphism he had to perform building through a destroying step that made him add more complex assumption. Notably, all the subsequent solutions faced similar difficulty as Gentry when attaining fully homomorphic encryption. They depended on the assumption of sparse subset-sum hence making them counter the difficulty.The added assumption is seen to be the main limitation to the solution of Gentry. The removal of this assumption has been the open problem in fully homomorphic encryption schemes' design [3]. As mentioned previously, RSA have some homomorphic properties, namely multiplication. For instance given RSA public key $pk = (N, e)$ and ciphertexts $\{\psi \leftarrow \pi_i^e modN\}$, one is able to compute $(\prod_i \pi i)^e \; modN$ which a ciphertext that encrypts the original plaintexts products[5].

According to Rivest, one can compute arbitrarily on data that is encrypted when dealing with an encryption scheme that is homomorphic fully[5]. This means that one can process encrypted data through querying, writing into it and anything else that can be considered to be or rather that can be represented as a circuit without the requirement of a decryption key[5]. A suggestion of private data banks was made whereby there is storage of data on the untrusted server in encryption form by a user while at the same time allowing the server to process and give a response. The server is responsible for sending all the data that is encrypted back to the user for processing. It is essential to note that a public encryption scheme E that is homomorphic contains 4 algorithms which include keyGen, Encrypt, Evaluate and Decrypt. Fully homomorphic encryption is at times known as crypto-computing whereby it is seen as a method of capturing the imagination. Fully homomorphic encryption is seen as a technology that would completely reduce the weaknesses found in the security systems [7].

The lack of requirement for decryption before the processing of data makes it an asset when it comes to privacy concerns. However, there is a problem in the implementation of this idea. In the 80's and 90's, the researchers introduced many schemes. Most of these schemes had support for addition and multiplication. However, they were not able to handle both operations simultaneously as there were limitations. This inability was a frustrating thing to the researchers. In order to attain arbitrary computation, they needed to come up with this doubly homomorphic scheme. A doubly homomorphic encryption scheme can be said to be a scheme that encrypts bits.

Table 2.1: NAND gate truth table

A	B	:	1+A*B
0	0		1
0	1		1
1	0		1
1	1		0

This means that the plaintext is either 1 or 0. For instance we can have a ciphertext that encrypts bits X and Y(Green 2012).This scheme can be used in the computation of $1 + X * Y$.The product of this function would the following truth table 2.1. The truth table describes a NAND gate which is a big deal to any computer engineer as one can derive all other useful boolean logic gates: AND, OR, NOT, XOR and XNOR [7].

It is important to note that homomorphic schemes that are not secure semantically such as RSA have stronger attacks. There is the statement by Boneh and Lipton that an algebraic privacy homomorphism that is over ring has the ability to be broken in sub-exponential time under an assumption that is number theoretic [5]. This is based on whether the scheme is deterministic or has the ability to provide an equal oracle. It is also seen that both Micalli and Goldwasser had a proposal for a homomorphic encryption that is semantically secure [5]. This scheme is additively homomorphic over Z2. By this, it means that XOR gates are contained in the permitted circuits set CE. Additionally, a number of homomorphic encryption schemes utilize linear codes or lattices. Boneh-Goh-Nissim and Polly cracker are some of the examples schemes that are semantically secure.

Sanders and Young are some of the researchers that utilize homomorphic encryption that is circuit private additively on the building of a circuit –private scheme that has the ability to handle NC1 circuits [5]. On the other hand, Paskin and Ishai perform this for purposes of branching programs that cover the NC1 circuits. Notably, their schemes have shorter ciphertexts' that are proportional to the branching program length. An observation from Gentry's work is if C- homomorphic scheme's decryption circuit exists in C then it is likely that the scheme can be changed into a scheme that is fully homomorphic [3]. A C-homomorphic scheme is a scheme that enables assessment of any circuit contained in class C. It is hard to come across an encryption scheme that has the ability to assess an addition non-trivial number and multiplication operations.

The solution by Gentry to this problem depended on the ideals algebraic notion in rings. In a situation that is high level, the message is said to be a ring element and there is masking of the ciphertext with some noise. Therefore the ciphertext form is m+xI, for some x in the ring where m is the message and the noise belongs to an ideal I. The ideal l contains two major properties which are first whereby the element is said to mask the message and the second is the possibility of generating a secret trap door that destroys the ideal. Importantly, the first property ensures there is security and the second allows the decryption of the ciphertexts.For instance in a case where F1 and F2 are encryptions belonging to n1 and n2 respectively then $F1F2 = (n1 + xl)(n2 + yl) = n1x + xyl)l - n1n2 + zl$ [3].

The ideal is destroyed during the decryption process and there is survival of product n1n2. Therefore F1F2 is an encryption to n1n2 as needed. The mentioned solution is required based on the first property which is an assumption of complexity on ideals in various rings. Hence one can say that the original work of Gentry depended on the complexity assumptions on ideal lattices. Moreover, there is also a homomorphic scheme that depends on the complexity of learning with errors problem which is abbreviated as LWE. In its assumption, it states that if sεZ_q^n is a dimensional vector for n then any polynomial number of random linear combinations of coefficients of s are undifferentiated computationally from the random elements in Zq that are uniform [3]. There is no reference to ideals in the LWE assumption. The LWE problem is similarly complex to the finding short vectors in any lattice. Earlier on, there was an observation of better comprehension of the lattice problems' complexity in comparison to the corresponding problems found on ideal lattices. In the building of efficient and cryptographic schemes that are highly expressive, the LWE assumption is seen to be agreeable [3].

The utilization of LWE in the building of a fully homomorphic encryption scheme is a show of the elegance and power of LWE. Encryption schemes building whereby its security depends on LWE assumption is seen to be straightforward. For instance in the encryption of a bit m utilizing secret keys sεZ_q^n, there is a selection of a random vector aεZ_q^n. An observation of the decryption is that the 2 masks do not interrupt each other. This means that through destroying the two masks, one can decrypt this ciphertext [3]. Notably, decryption algorithm first calculates the mask (a,s) and there is a subtraction from b. Despite this scheme being homomorphic, that is naturally addictive, there is a presentation of a problem. Basing on the work of Gentry and Halevi, there is shown that the scheme supports only one single homomorphic multiplication with the risk of causing a big destruction to the ciphertext. In order to comprehend the scheme's homomorphic properties, one can also look at the decryption algorithm [3].

2.2 Bootstrapping and Full Homomorphism

In the building of fully homomorphic encryption, Gentry talked of a number of steps that are involved. First, there is the building of a somewhat homomorphic scheme which has support for low degree polynomial assessment when it comes to the encrypted data (Sen 2013). The next thing is the squashing of the procedure of decryption in order to express it as a low degree polynomial supported by the scheme. The final step is the application of the bootstrapping transformation so as to attain a homomorphic scheme that is full. A bootstrappable scheme is a moment that the polynomial degree able to be assessed by the scheme surpasses the decryption polynomial degree by a two factor [8].

The designing of a bootstrappable scheme is said to be done through a somewhat homomorphic scheme that is a GGH-type scheme over ideal lattices. With the use of a key generation procedure that is appropriate, the scheme's security made to be a number of lattice problems' worst case complexity. The lack of the somewhat homomorphic scheme being bootstrappable, led to Gentry explaining a transformation needed to crush the process of decryption [8]. In return, the decryption polynomial degree is reduced. This scenario is made possible by the addition of a public key and an extra secret key hint in a sparse subset-sum problem form. There is an augmentation of the public key with a huge set of vectors. Post-processing of the ciphertext can be done by utilization of the additional hint. Low degree polynomial decryption of the ciphertext that is post-processed takes place resulting in a bootstrappable scheme. Even in the scheme's private key version, the secret key is a short support of random ideal lattice.There is a complication in the generation of public pairs using the appropriate distributions needed for reduction of the worst case to average case.

NTRU cryptosystem is the origin for the use of ideal lattice in cryptography [8]. Ideal lattices are utilized in this approach for cryptographic buildings that are efficient. Compared to the normal lattices, the ideal lattices' added structure cause their representation to be more powerful and allows the computation to be faster. There is the raising of the question whether ideal lattices and LWE can be used together to attain the benefits of the two at the same time. This means that the efficiency and simplicity in LWE and the powerfulness in ideal lattices are achieved at the same period. A research by Brakerski showed that this can be achieved. There is the building of the somewhat homomorphic scheme depending on the LWE. An inheriting of the competence and ease together with worst case association in ideal lattices takes place. Additionally, there is the enjoyment of the security of key-dependent message. This is due to the fact that there is secure encryption of its own secret key polynomial functions.

2.3 HElib

The HElib is a homomorphic encryption library which is implemented using the Brakerski-Gentry-Vaikuntanatham(BGV) scheme. HElib together with other optimizations allows the assessment of homomorphic to be faster concentrating mainly on the utilization of smart-vercauteren ciphertext. The BGV variant implemented is defined over polynomial rings of the form $A = Z\left[X\right]/\phi_m(X)$ where m is a parameter and $\phi_m(X)$ is the m'th cyclotomic polynomial[9]. In this scheme the "native" plaintext space is usually the ring $A_2 = A/2A$, which is namely binary polynomials modulo $\phi_m(X)$. With the use of the Smart-Vercauteren CRT-based encoding technique the vectors of bits in a binary polynomial are "packed" so a polynomial arithmetic in A_2 translates to entry-wise arithmetic on the packed bits. In the scheme the ciphertext space consists of vectors over $A_q = A/qA$ where q is an odd modulus that evolves with the homomorphic evaluation. The system is parametrized by a "chain" of moduli of decreasing size, $q0 < q1... < qL$ and freshly encrypted ciphertext are defined over R_{qL}[9]. By switching to smaller and smaller moduli during homomorphic evaluation until we cannot compute anymore, we get the ciphertexts over A_{q0}

The present state of HElib allows only for use by researchers that work on homomorphic encryption and it utilizations. It is considered to be a homomorphic encryption's assembly language. This means that HElib offers low-level procedures with plenty optimizations access. It is expected that in future it will be able to offer high-level procedures.

2.4 Application of Homomorphic

Homomorphic encryption is utilized in many areas. One of the applications of homomorphic encryption is in private medical records. It is noted that the providers of healthcare upload medical records to the cloud service in form of public key encryption [10]. By doing this, they can control the access of data and perform an encrypted search on the data. Under this also, there is the ability of different monitoring devices to stream data that is encrypted. An example of this is the blood sugar level or the heart rate. There is the computation of the statistical functions by the clouds service on the data. Additionally, there is the determination of the risks and the sending of alerts to the patients and doctors. The computation of functions on the encrypted data is performed with homomorphic encryption [10]. It is important to note that most of the Cryptosystems contain the homomorphic properties. This includes Elgamal and RSA which only offer multiplicative or additive homomorphism [10].

Another application of homomorphic encryption is in the mobile agents' protection. A homomorphic scheme that is on a special non-abelian group, causes a cryptosystem that is algebraically homomorphic on the finite filed F2[8]. Cryptosystem has the possibility to encrypt the entire program so that it becomes executable since all the conventional architectures of the computer depend on binary strings[8]. Therefore from this, it could be utilized in the protection of mobile agents by encryption from malicious hosts. There are two ways in the protection of mobile agents. One of them is using encrypted functions in computation and the other is the use of encrypted data in computing. Using the encrypted function in computing is seen as a special scenario of mobile agents' protection. In this scenario, there is an assessment of a secret function publicly ensuring that the function is secretive. The privacy is ensured through the utilization of the homomorphic cryptosystems. Considering the other way which is the use of encrypted data, the homomorphic schemes work on encrypted data in order to execute publicly while at the same time ensuring privacy. The way in which this is done is by first off encrypting the data and then an exploitation of the homomorphic property is done so as to execute with encrypted data.

Multiparty computation is the other application of homomorphic encryption. In this scheme, there is the interest of computation of a common public function on the inputs while still maintaining the privacy of their individual inputs [8]. The problem, in this case, belongs to the computing area with encrypted data. In the protocols of multiparty computations, there is a set of $n <= 2$ while in the computation with encrypted data there is $n = 2$[8]. Additionally, the protocols of multiparty computation have their functions publicly known before computation while in encrypted data computation scenario there is the private input of a single party. The other homomorphic scheme is the secret sharing scheme. In this scheme, there is sharing of a secret between parties ensuring no individual party can reconstruct the secret using the information available. Conversely, the cooperation of some of the parties may lead to the rebuilding of the secret [8]. Under this scenario, the homomorphic property states that shares of the secret composition are equal to the composition of secrets shares. Multiparty schemes and secret sharing schemes are instances of threshold schemes [8].

There is implementation of threshold schemes using the techniques of homomorphic encryption as a zero-knowledge proof is a cryptographic protocol primitive that is essential. It serves as an instance of homomorphic cryptosystems' theoretical application. There is the utilization of zero-knowledge proofs in proving some private information knowledge. An example is consideration of the scenario whereby a user is required to prove his identity to a host by use of private password and account. In this case, there is the need for the user to maintain the privacy of their information during the protocol operation. The use of zero-knowledge ensures that there is communication by the protocol the knowledge that was intended and no additional knowledge [8]. Oblivious transfer can be said to be cryptographic primitive that is interesting. There is sending off a bit to the second party by the first party in an oblivious transfer protocol that is two-party 1 out of 2.This occurs in such a way that there is receiving of the bit by the second party with a ½ probability without the knowledge of delivery by the first party [8].Digital fingerprinting and watermarking schemes are entrenched information that is added to the digital data.

There use of the homomorphic property in the addition of a mark to the initially encrypted data. Watermarks are helpful in ensuring that the work is not stolen through identification of the original owner. Under the fingerprinting schemes, there is the identification of the buyer to make sure the data is not distributed illegally [8]. Lottery protocols involve cryptographic lottery whereby the pointing number to the winning ticket is supposed to be randomly selected by the participants. The homomorphic scheme is used in this case in the following way: Every player selects a random number and encrypts it([8]. By use of the homomorphic property, there is the computation of the encryption of the random values sum. The addition of the threshold decryption scheme and thus results to the needed functionality.

There is also the commitment scheme that is a cryptographic primitive that is essential. A commitment is made by the player in this scheme. The player is able to select a value out of a set and the choice is committed permanently. The selection is made private although later on is made public. There is an efficient implementation of commitment schemes utilizing homomorphic property [8]. There are security issues when it comes to the aggregation of data in wireless sensor networks. The use of the WSNs technique which at the intermediate nodes it brings together partial results minimizes the communication overhead.

Utilization of this technique is the cause of security and privacy concerns. For instance in areas such as military surveillance, the private data sensitivity is high and therefore the conduction of aggregation should be in a privacy maintaining way.The aggregator has to be barred from seeing the sensitive data. The schemes, in this case, are not capable of performing aggregation function that is multiplicative. The homomorphic encryption scheme can be utilized in the protecting privacy during the computation of arbitrary aggregation in a wireless sensor network [8].

2.5 Related work

This section gives a brief overview of the existing open source implementations of fully homomorphic encryption schemes.

2.5.1 Implementation of Gentry's Fully Homomorphic Encryption Scheme

In 2010 the first attempt to implement Gentry's first plausible construction of a fully homomorphic encryption scheme [5] was made by Smart and Veracauteren [11]. They chose to implement the scheme using a variant of "principal-ideal lattice" of prime determinant. The lattices can be represented by two integers and they described a method where the secret key is represented by a single integer. The underlying somewhat homomorphic scheme they implemented were not able to support large enough parameters to make Gentry's squashing technique go through, which resulted in the scheme not being "bootstrappable" and therefore not a fully homomorphic scheme. The Gentry-Halevi implementation continues in the same direction of the Smart-Vercauteren implementation using optimalization to make the implementation "bootstrappable" and fully homomorphic. The Gentry-Halevi implementation of the Gentry's fully homomorphic encryption scheme is considered as the first reported implementation of a fully homomorphic encryption.

2.5.2 FHEW library

The FHEW library is implemented by Leo Ducas and Daniele Micciancio and uses a combination of Regev's LWE cryptosystem with the bootstrapping techniques of Aplerin-Sheriff and Peikert[12]. The implementation is made as a response to solve the main bottleneck of Gentry's FHE schemes bootstrapping procedure which requires refreshing noisy ciphertexts to keep computing on encrypted data [13]. When doing operations on "noisy" encryption schemes, each homomorphic operation increases the noise which can lead to the noise growing to a level where the ciphertext can no longer be decrypted. Gentry's bootstrapping technique solve this problem homomorphically computing the decryption function on encrypted secret key which brings the noise of the ciphertext back to acceptable levels. This bootstrapping technique is the bottleneck of Gentry's FHE implementations due to the complexity of homomorphic decryption. The FHEW implements a new bootstrapping technique which allows for computations to be performed in less than a second. The implementation is simpler than HElib as it performs only a single bit operation before bootstrapping, while HElib allows more complex operations[13].

2.5.3 TFHE library

The TFHE implementation revisits the fully homomorphic encryption based on GSW and its ring variants. The implementation improves the FHEW bootstrapping procedure by replacing the internal product of the GSW with a simpler external product between GSW and an LWE ciphertext. A formalization of the LWE/ringLWE is provided on numbers or polynomials over the torus, which is obtained by combing the Scale-Invariant-LWE problem of or the LWE normal form of with the General-LWE problem Brakerski-Gentry-Vaikutanathan[14]. They call the representation of LWE ciphertexts which encode polynomials over the Torus, TLWE.

Approach

This chapter outlines the methodology and actions which will be taken to create a Fully Homomorphic Encryption API in Python. We will also take a look at how we can test the Python API to measure the difference in performance between the C++ library and the Python library. We will explain the approach of choosing a Python wrapper for the C++ code. And we will look at how we can determine possible usages and limitations of the library.

3.1 Objectives

The main objective of this thesis is to create a Fully Homomorphic Encryption API in Python and measure the performance to get a sense of how far we have come in the development and implementation of FHE schemes. The Python API will be based on an already existing FHE library. The Python API should support a multitude of functions in order to classify it as a FHE API. Ideally the Python API should not be significantly slower than the original library in terms of compilation time and runtime. Lastly we want to highlight the benefits/drawbacks of having a FHE scheme in Python and find potential usages of the API.

The thesis work is structured into the following three stages

1. A design stage

2. An implementation stage

3. Measurement, analysis and comparison stage

3.2 Design Stage

In the design stage we will discuss the necessary steps to design a solution which satisfies the objectives mentioned above. For simplicity sake the design stage is divided into several steps needed to fulfill the requirements of the objectives, these are outlined below.

1. Research and choose a suiting C++ wrapper for python
2. Create an API which is easily understood
3. Find a suitable model for testing the API

3.2.1 C++ wrapper for Python

In this section we will examine different C++ wrappers to find one that is suitable to wrap the FHE scheme to python. Before choosing the wrapper the FHE scheme implementation should be examined to see if there are some special functionality the wrapper should support. If the FHE scheme have a lot of user defined objects or any other functionality, the wrapper chosen should be able to handle this in a somewhat simple matter. The main goal of this exercise is to find a wrapper to create a Python API from an already existing FHE scheme. The wrapper should ideally have a good performance i terms of runtime and the API created with the wrapper should be well structured and easily organizable.

3.2.2 Create the API

There are currently few options when it comes to FHE libraries in Python. When the Python API is created it is important that the API is well structured, well documented and it should include examples for potential usages. Ideally the Python API should be as similar to the underlying library as possible, making it easier for the user to find valuable documentation on how to use the library. The library being wrapped to Python is most likely an extensive library which means the user should be able to use the documentation for the API combined with the underlying library documentation for a more thorough understanding on how to use the API to the full extent. Any differences between the API and the underlying library should be highlighted making it easy for the user to see the parallels between the API and the library. As the underlying library is most likely an extensive library it will consist of many files. When creating the Python API this needs to be taken into consideration as the API should be presented in a lucid manner corresponding to the underlying library. As the underlying library will most likely be large it should also be taken into consideration how much of the library should be wrapped to Python to support the functionality required by an FHE scheme.

3.2.3 Find a suitable model for testing the API

Once the API is created the performance of the Python API should be measured with regards to the existing library in C++. A benchmark test should be created on the existing C++ library which can be used in comparison to the Python API. The tests created should include most of the functions wrapped to Python to see if there are some operations or functions which are slower than others in regards to the benchmarks. If a function or an operation is slower than others it should be investigated to provide an explanation why this is the case and possible solutions to the result. The tests implemented should not only serve as a measurement in performance but also as examples to potential usages of the library. The tests should show how the library can be used in different scenarios and also show the limitations to the library. It's important that the tests highlight the benefits and drawbacks of having the FHE scheme implemented in Python compared to the C++ implementation. If there are other implementations of a FHE scheme in Python this could be used in the testing phase to measure the difference in performance in terms of runtime and functionality compared to the API implemented.

During the research stage of the assignment an implementation of the FHE scheme in Python was found. This implementation does have some limited functionality compared to the existing C++ library, but it can be used as a comparison to our implementation. We should look at this library to compare the difference in performance and functionality to our implementation of the HElib library.

3.3 Implementation stage

The implementation stage is divided into a few objectives to satisfy the goals outlined in the design stage.

1. Choose an underlying FHE scheme implementation

2. Wrap the code to Python

3. Create a structured Python API

3. Create tests to measure the performance

3.3.1 Choose a Fully Homomorphic Scheme implementation

The first task of the implementation stage is to choose a suiting FHE scheme which will serve as the underlying library of the Python API. The FHE scheme chosen should be up-to date and therefore be based on one of the 2nd generation of homomorphic cryptosystems. These includes The Brakerski-Gentry-Vaikuntanathan cryptosystem (BGV) [15], Brakerski's scale-invariant cryptosystem [16], The NTRU-based cryptosystem due to Lopez-Alt, Tromer, and Vaikuntanathan (LTV)[17] and The Gentry-Sahai-Waters cryptosystem (GSW) [18].

Of the 2nd generation homomorphic cryptosystems there are three open source libraries. The HElib library [19] that implements the BGV cryptosystem with the GHS optimization, The FHEW library [13] which implements a combination of Regev's LWE cryptosystem [20] with the bootstrapping techniques of Alperin-Sheriff and Peikert [12] and the TFHE library[21] which proposes a faster variant over the Torus. Of the three libraries the HElib library was chosen as a basis for the Python API. HElib offers some interesting features like ciphertext packing and recryption.

3.3.2 Wrap the code to Python

The HElib library is chosen as the underlying library for the Python API. Before wrapping the code to Python and create the Python API a suiting wrapper should be chosen. The HElib library uses the NTL mathematical library which means it most likely will use user defined objects and types. The wrapper chosen should be able to handle this in a good manner. The main goal when exporting the library to Python is to create a shared library which can be imported as a module in Python. This module should be directly and easily accessible from Python and it should be complete in the sense that the user should be able to use all the functionality in the module without the need specify return values or create any form of conversions of C++ types.

In the Python module of the HElib library the main functionality of the library will be exported, this means as the library is quite extensive, that there will still be parts of the library which will not be exported. As the module might be extended in the future to support more functionality it should be structured in such a way to make it easy for anyone not familiar with the project to understand how it works. Also the HElib library uses a technique called ciphertext packing to create a bulk of data to which an instruction is performed, called SIMD(Single Instruction Multiple Data). These ciphertexts are vectors. This is something to keep in mind when choosing the wrapper as vectors are not a familiar concept in Python.

Another thing to keep in mind, the HElib library consists of multiple functions which have the same function name, so called overloaded functions. These functions have the same name, but with different input types. When the function is called the interpreter recognized the input types and makes a function call to the correct function. This is also a concept which is not familiar in Python. Before choosing the wrapper we should consider the following requirements for which the wrapper should handle.

- C++ Structures
- User defined objects/types
- Vectors
- Overloaded functions
- Pointers

The wrapper should not only support the requirements mentioned above. Ideally the wrapper chosen should fast, in terms of compilation and runtime. It should be simple Among the more known wrappers from C++ to Python we have Boost, Ctypes, Swig and Python C-extensions.

C-extenstions

Python C-extensions is a feature which is implemented within Python and does not require any extra software. It is a good software if the objective is to wrap only small parts of code. The main problem with Python C-extension is that it is low level, making it time consuming to expose functions over to Python. Other drawback of the software is that it is prone to memory leaks and is Python version dependent. As the HElib library consists of a rather large amount of code which is going to be exposed to Python, this is not an option.

SWIG

On the other side of the spectrum we have SWIG. SWIG stands for Simplified Wrapper and Interface Generator and is a software designed to automate the process of wrapping code to Python seamlessly. Automatically wrap the code from C++ to Python would be ideal if it can handle all the requirements mentioned above. The problem is, it doesn't. Although it does handle standard vectors and pointers fairly well, it only have partial support for overloaded functions and custom vectors have to be wrapped manually. Another key point about SWIG is that, if you do not wish to wrap a whole file one must specify the desired functions to be wrapped. The HElib library is quite large and the majority of the code does not have to be wrapped to Python to be able to support the desired functionality. This means by using SWIG one have to specify exactly which functions to be wrapper combined with the manual wrapping required the user defined objects makes the automation process obsolete.

Ctypes

So, the C-extensions are too low level making the code wrapping slow and tedious. SWIG automates this process, but since the HElib consists of many overloaded functions and user defined objects there will not be much code left to be automatically wrapped to Python. Another option is Ctypes. Ctypes is a standard library module in Python, which adds the standard C types to python. At runtime it can load a shared library and import it's symbols, allowing a Python application to make functions calls into the library without any special preparations. It offers extensive facilities to create, access and manipulate simple and complicated C data types in Python.

With Ctypes the HElib library can be compiled as a shared library and then imported into Python. The problem with this approach is that one still have to specify the return value of the different functions. For any special objects like a vector or overloaded functions which is not a standard in Python, a converter would need to be written in C++ to be exported to Python. This would result in a combination of C++ and Python code to make the library being useful, which would make the API unstructured and messy. This also defeats the purpose of the goal which is to create a shared library which can be imported as a Python module and be directly accessible in a simple matter.

Boost

The last wrapper considered is Boost Python. The main goal with Boost is to enable the user to export C++ functions and classes to Python using nothing more than a C++ compiler [22]. Boost is a C++ library where the syntax is similar to C++ and it's designed to be minimally intrusive to the existing C++ code. Having the syntax similar to C++ is beneficial at it does not require the user to learn an additional language as it would with other wrappers like SWIG or SIP. With Boost the HElib library can be exported to python with rewriting the original code, this is a desired feature, as the library is complex making it hard to rewrite. Boost also supports structures, vectors, pointers, user defined objects and overloaded functions.

3.3.3 Create a structured Python API

This might sound like it should not be a big deal, but using a wrapper which allows one to export functions while maintaining them in a structured manner is important, especially when the project grows large. A large project will consist of multiple files, many functions and classes. Exposing the project code without a proper structure does not only make the work for the one exporting the code more difficult, but even more so for the ones who are not familiar with the project. When dealing with a small project one could gather all the exported code within a single file and it would be no problem. But when dealing with a large project which consists of multiple files, this would be very unorganized and hard for anyone to see how the code is structured.

Ideally the structure of the exported code should be similar to the structure of the original code making it easy to see what file it corresponds to in the original code. For example when dealing with overloaded functions which is functions having the same name but with different input parameters. If an overloaded function is spread over multiple files and the exported code is gathered in a single file, which means all the overloaded functions are gathered in the same file. This would make it very hard to see what context each of the overloaded function relates to. Now, you are probably asking yourself, why does this matter as long as all the functions are exported? For starters, the concept of overloaded functions are not relatable to Python. When a C++ compile the code it uses a technique called name mangling which encode functions and variable names into unique names so that linkers can separate common names in the language making it possible to user overloaded functions, namespaces and templates.

Although Boost Python do have support for exporting overloaded functions it means that the functions exported will have to be given a unique name. One could organize the different function by giving them a unique name which corresponds to which context the function are being used, but this will result in rather large function names. By having the functions separated in files which corresponds to the original code, it is easy to see where and in what context the functions belongs to. Another important issue is when the original library is undergoing changes and updates. During the change or the update only parts of the code will be updated. If there is no structure to the exported library it would be hard to find out what part of the exported code which needs to be changed.

3.3.4 Create tests to measure the performance

Once the code has been exported and the projects is organized in a structured manner it is time to test the exported library functionality. With the tests we want to see the difference in performance between the original library and the exported module. There will be created a test which involves the majority of the functions with a set of fixed parameters which will run in a loop where the goal of the test is to measure the difference in performance when multiple functions are called multiple times. The amount of rounds in the loop will be the variable to see if there are more/less difference in performance depending on how many times an exported function is called. A timer will be used to measure the runtime of the different functions to see if there are some functions that are slower than others. The runtime of the encryption/decryption and total runtime will also be measured.

The second test will compare the implementation of the exported library with another project by Ibarrondo [23]. In the project an abstraction of HElib (Afhel) is created in C++ which includes the most important HElib operations. This abstraction is wrapped using Cython, creating what he calles Python for HElib (Pyfel). The goal of the test is to measure the difference in performance and functionality between the implementations.

In the last test and implementation of a server and a client will be implemented. This should simulate a client and a cloud server. The client will encrypt some data, which are sent over to the server. The server will perform some computation and send the data back to the client. The client will then decrypt the data making sure the data is correct. With this implementation we want to measure the difference in performance between different scenarios.

1. A baseline where the client sends the data in plaintext to the server which performs the computation and send it back.

2. The encrypted data is stored on the server. The client fetches the data, decrypts it, performs the computation, encrypts it again and stores in on the server.

3. Use the HE implementation to perform the computation on the server.

Design

In this chapter we will display the results of the tasks outlined in the approach section. In order to export the HElib library and make a well structured project we will first look at how HElib is designed. This is important as we want the exported library have the same structure as the HElib library. As the HElib library consists of many modules, many of which we might not need to export to Python, we should look at the structure of the modules and what they contain to figure out which of the modules we need to export to make the Python API functional.

4.1 HElib design

> “The HElib library consists of four layers where the bottom layer contains modules for implementing mathematical structures and other various utilities. The second layer implements the Double-CRT representation of polynomials, the third layer implements the cryptosystem itself, and the top layer provides the interfaces for using the cryptosystem to operate on arrays of plaintext values[9]. The two bottom layers are though of as the "math layers", and the two top layers are thought of as the "crypto layers" ”.

In the bottom layer we have the modules Numbth, timing, bluestein, PAlgebra, PAlgebraMod, Cmodulus, Indexset, and IndexMap. In the second layer we have FHEcontext, SingleCRT and DoubleCRT. In the third layer FHE, Ctxt and KeySwitching, and Encrypted Array in the top layer. To figure out which of these modules we need to export to Python in order to create a functional API, we have to take a look at what they do. Keep in mind that we only have to export the modules of which we have to call the functions directly.

Timing The timing module contains some functions for measuring the time that methods use to execute. This module allows us to have a timer per function, measuring the time for each function, which can be useful when testing the exported library, but it is not a necessity.

Numbth The numbth module consist of many different little utility functions including CRT-reconstruction of polynomials in coefficient representation, conversion functions between types, procedures to sample at random from various distributions[9].

Bluestein "The bluestein module implements a non-power-of-two FFT over a prime fiels Z_p, using the Bluestein FFT algorithm"[9] The module contains two functions which compute the length-n FFT of the coefficient-vector of x and put the result back in x.

Cmodulus and CModulus These are two classes which provides and interface layer for the FFT routines of bluestein relative to a single prime. They also keep the NTL "current modulus" for that prime

PAlgebra The PAlgebra class contains the structure of $(Z/mZ)*/(p)$ and the quotient group $Z_m^*/\langle 2\rangle$. In the implemented scenarios/tests the PAlgebra class is mainly used to get the factorization of $Phi_m(X) mod p^r$

PAlgebraMod The PAlgebraMod class implements the structure of the plaintext spaces, which is of the form $A_p r = A/P^r A$ for a small value of r[9]. The r value is typically 1 for ordinary homomorphic computation but are expected to be larger than 1 when used for bootstrapping. The plaintext slots are determined by a factorization and once this is done, an arbitrary factor is chosen which corresponds to the first input slot. The class provides functionality like encoding/decoding and storing of "mask tables" which is used by EncryptedArray to facilitate rotating slots of encrypting data.

FHEcontext The objects in the higher layers of the library are defined by some parameters, like a sequence of small primes that determine the modulus chain. The FHEcontext class is used to allow convenient access to these parameters and provides access methods and some utility functions. The FHEcontext also contains IndexSet object which are partitions used when generating the key-switching matrices in the public key and key-switching on ciphertext. It also contains some functions for chosing parameters and adding moduli to the chain. In the FHEcontext class we find the FindM(long k, long L, long c, long d, long s, long chosen_m, bool verbose=false) a helper class which is used in all of the examples. This helper function is used to select the values for the parameter m, defining the m'th cyclotomic ring. The FHEcontext also implements the public and secret keys, and key-switching matricies.

DoubleCRT A DoubleCRT object is tied to a specific FHEcontext and is used to manipulate the polynomials in the Double-CRT representation. The FHEcontext must remain fixed, but the set of primes can change dynamically, so the matrix can add or remove rows as we go. These rows are kept as a dynamic IndexMap data member. The DoubleCRT objects supports the usual arithmetic operations like addition, multiplication, etc.

Ctxt The Ctxt module implements the ciphertext and the ciphertext arithmetic. The library deals with both "canonical" and "non-canonical" ciphertexts. This is supported by using ciphertexts which consists of an arbitrary-lengt vector of ciphertext parts, where each part is a polynomial with a "handle" that points to the secret key that should multiply this part at decryption[9].

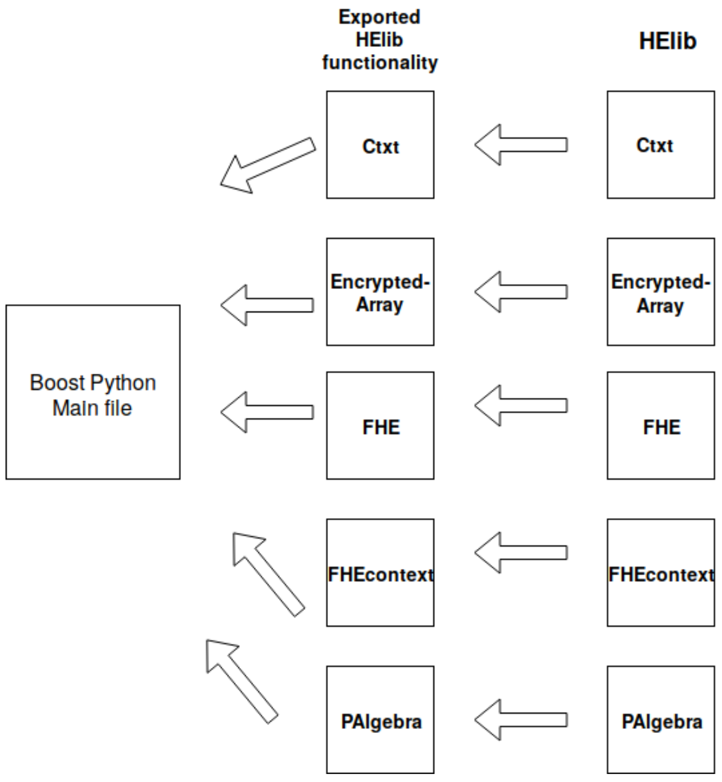
EncryptedArray The EncryptedArray class is used to present the plaintext values as either a linear array or as a multi-dimensional array. It also provides methods which can handle both encoding and homomorphic operations in one shot.

4.2 HElib Python design

The design of the HElib Python library is made purposely similar to the HElib library in C++. There are several reasons for this including having a structured and easily understood API, making it easier for anyone who is not familiar with the project to understand how is it structured. In Boost Python, when exporting a lot of classes the compile time can be substantial and the memory consumption can easily become high. This can be solved by splitting the class definitions over multiple files[24].

For each of the header files in the HElib library which contains functions or functionality used in the exported python library a corresponding file has been made which contains the functions and classes which are exported. All of the files with exported functions and classes have a corresponding header file which are used in order to call the functions in the main file, see figure below.

Figure 4.1: Presentation of the HELib Python structure



4.2.1 Exported functionality

All the files which contains the exported functionality are named the same as the corresponding file in the HElib library with a _python abbreviation for simplicity sake. In addition to the files shown in the figure above an additional file is contained in the library which contains the conversion of different vectors, which allows for the handling of vectors in python. We will now take a look at the exported functionality in the HElib python library.

Ctxt

As previously mentioned the Ctxt module handles the ciphertexts and the arithmetic operations done on the ciphertext. In the Ctxt module we find three classes: Ctxt, CtxtPart and SHandle. The Ctxt class holds a single ciphertext which is a vector of ciphertexts parts where each part have a “handle” describing the secret-key polynomial. The CtxtPart class can also be used as a DoubleCRT object as CtxtPart is derived from DoubleCRT. When exporting the Ctxt class we only have to export the public variables and functions which we will be called directly, this means all functions and variables under the public section of the class. The SKHandle class is handled by the CtxtPart class which are handled by the Ctxt class. We wont do any direct calls to neither CtxtPart or the SKHandle class, meaning only the Ctxt class are exported. This also includes the functions which are declared outside the classes, which are mainly operations on the ciphertext like innerproduct and totalproduct.

EncryptedArray

In the EncryptedArray module the information about an FHEcontext are stored. It also supports encoding/decoding and encryption/decryption. In the module we find the classes EncryptedArrayBase, EncryptedArrayDerived, EncryptedArray, NewPlaintextArrayBase and NewPlaintextArray. The EncryptedArray class works as a wrapper around a smart pointer object to the EncryptedArrayBase object, where the EncryptedArrayBase is a virtual class and EncryptedArrayDerived is a derived class of EncryptedArrayBase. Of the three classes mentioned the EncryptedArray class is the only one exported as this works as the interface including the functions which will be called directly. The NewPlaintextArray class is mainly used to simplify testing, where the NewPlaintextArrayBase class is a virutal class. Functions which are used to perform operations on the NewPlaintextArray are exported along with functions used to encrypt/decrypt, encode/decode on the EncryptedArray.

FHE

The FHE module is used to handle the secret key and the public key. In the module we find the classes: KeySwitch, FHEPubkey and FHESecKey. The KeySwitch class is used to convert a ciphertext part with the respect to the secret-key's polynomials into a canonical ciphertext. In the examples/tests this class is never called directly and in the FHEPubKey we find keySwitching matrices represented as vectors in the private section. The FHEPubKey class handles the public-key while the FHESecKey which is a derived class of the FHEPubKey class handles the secret-key. Both of the classes are mainly used to generate and handle the secret/public - keys. The classes also contains key-switching and encrypt/decrypt functions which have not been exported, or used in the examples implemented.

FHEContext

The FHEContext class is used to handle the parameters of the objects in the higher levels of the library. The file only consists of a single class FHEContext with multiple functions to modify the context/parameters. A helper function which is used to set the parameters are also located in this file, which is called FindM(long k, long L, long c, long p, long s, long chosen_m, bool verbose=false). This is the most interesting function in this file as it helps us used set the values for the parameters m(defining the m'th cyclotomic ring) and is used in all of the examples. k is the security parameter, L is the number of levels, c is the number of columns, p is the characteristic of the plaintext space, d is the embedding degree, s is at least that many plaintext slots, chosen_m is the preselected value of m.

PAlgebra

The PAlgebra class contains the structure of $(Z/mZ)*/(p)$ and the quotient group $Z_m^*/\langle 2\rangle$. In the implemented examples the PAlgebra class is mainly used to get the factorization of $Phi_m(X)modp^r$, which is used when initializing the EncryptedArray class. In the module we also find PAlgebraMod which is a wrapper for the PAlgebraModBase and provides direct access to the functions in the class.

Python vectors

As python are not familiar with the concept of vectors a converter has been created to inform python how these objects should be handled. In this file we find the converter for two different vectors, a standard long vector and a NTL:ZZX vector, which is a vector defined in the mathematical library which the HElib is using, NTL.

HElib Python mainfile

All of the sections describes above have their functionality exported in their own separate file making it easy to see how it corresponds to the original library and to minimize the memory consumption which occurs when exporting many classes. Each file have their own boost python module where the classes and functions are exported. Each file also have a corresponding header file declaring the export function located in the file. The mainfile consist of some helper functions which include a debug, timer, and some arithmetic functions. Within the boost python module in the mainfile all other modules located in the separate files are called.

Implementation

5.1 Building the HElibrary

The original HElib library compiles the library into a private library called FHE.a. Having a private library made it troublesome when trying to export the functions with Boost Python which requires to have a link to all original files during the compilation. It should work in theory to compile the Boost python code with a link to a shared library, but an easier method was to compile the HElib library as a shared library. By compiling the HElib library as a shared library one only have to provide the link “-lfhe” during the compilation of the Boost python code to let the compiler know where the original code is located. During the compilation of the HElib library the files are first compiled as intermediate files which are then linked to create the shared library. The makefile used to create the shared library is borrowed from (link)

5.2 Exported functionality

Each of the files which are created with the same functionality and with a name corresponding to the original files have the exported functionality located in one function which corresponds with the filename. In this function all the classes and functions which are exported to the python library are located. The function is declared in a header file and is called within the Boost python module in the mainfile.

5.2.1 Classes

The exported classes are defined by class_<name_of_class>("python class name"), where the name_of_class is the original name of the class in C++ and "python class name" is the name of the class when exported. If the class have a standard constructor, there is no need to define this as it will initialize automatically. If the constructor is not standard however, it has to be initialized. For example, the EncryptedArray class does not have a standard constructor. In fact it has two constructors, one which takes a FHEcontext , ZZX, and one which takes FHEcontext, PAlgebraMod. Both of the constructors have been defined within the class using the .def(init) function. Its important to note that since the class does not have a default constructor we are telling the compiler to not initialize the class by passing in the no_init argument when exporting the class.

Some of the classes exported are derived from other classes. To export a class which have a base class we have to tell the compiler about the base class during the export of the derived class. This is done by passing in the bases<"class"> argument when exporting the class. To export a derived class we also have to export the base class. When declaring the variables in a function we use the .addproperty("python name", class_name::variable) where "python name" is the name the variable will recieve in the shared library in python, class_name is the class name of the C++ class and variable is the name of the variable in C++. Functions within a class is exported using the .def("python name", class_name::function_name) where the input is the same as when exporting variables and the function name is the name of the function is the original library.

5.2.2 Overloaded functions

In C++ there is a concept where one can have many functions with the same name, but with different inputs. This is done during the compilation of the code when the compilator uses a technique called "name mangling". During the compilation of C++ the functions are given a unique id/name which enables the compiler to differientiate between the overloaded functions. Python does not compile the code, but uses an interpreter which mean this concept does not work in python.

To handle this when exporting overloaded functions in Boost python, each of the overloaded functions are given a unique name when exported. For example, the EncryptedArray class consists of many overloaded functions. One of these are the encrypt function, which are repeated three times with different input parameters.

When exporting the encrypt functions we first declare the function and bind it to a unique name. Void (EncryptedArray::*e1)(Ctxt, const FHEPubKey, const NewPlaintextArray) const = EncryptedArray::encrypt;

The first parameter (EncryptedArray::*e1), “EncryptedArray” refers to the name of the class in C++ while “e1” is the temporary name which are later used when exporting the function. The (Ctxt, const FHEPubKey, const NewPlaintextArray) is the input paramteres to the overloaded function we are exporting and the EncryptedArray::encrypt; is the name of the class and the name of the function. The function are now declared with the unique identifier “e1”, with the input parameters and the name of the overloaded function. To export the function we use the .def(“python_name”, unique identifier) method, where the python name will be the new name of the function in the shared library and the unique identifier is “e1” in this example.

Another concept of the overloaded functions is when a function have a standard input parameter. For example void Test(int a, int b, in c=2), where the input c equals to 2 if the nothing is passed in the c argument when the function is called. Boost python handles this with the use of BOOST_PYTHON_FUNCTION_OVERLOADS (new_name_of_function, name_of_function, min input, max input). The “new_name_of_function” is a temporary name which is used later when exporting the function. “Name_of_function” is the name of the function in C++ and min input is the minimum number of input arguments that can be passed to the function, this means all the arguments which have to be passed in to make the function work properly. The max input argument is the maximum number of arguments which can be passed in the function including the overloaded function arguments.

5.2.3 Vectors

Vectors is another concept which are not familiar in python. In order to handle vectors in python a converter has been created. In the exported library there are two different vectors being used. This is a standard long vector and an NTL vector, NTL is the underlying mathematical library. To create an converter for the vectors the class method is used. Example: class_<std::vector<long»(“pyvector”), which is the standard long vector and “pyvector” is the name of the vector in python. To export the vector as an actual vector and not a class we are defining the vector within the class using the vector_indexing_suite parameter which creates an indexable container. It emulates a python container which enables us to use the vector as a list with some limited functionality in python.

5.2.4 Mainfile

In the main file we find the actual Boost python module. Within this module all the exported functions from the different files are called. In this file some helper functions has also been implemented. This includes functions to get the product/adding two ciphertexts. The operator in the Ctxt class have not been exported, which means in order to modify the ciphertexts these helper functions have been created and exported. In the mainfile we also find a function which are used to compare two plaintext. It takes an EncryptedArray, secret key, plaintext and a ciphertext as an input. The ciphertext are decrypted using the secrey key and compared to the plaintext. This function is used extensively when testing as the operation(s) are performed on a ciphertext and a plaintext which are later compare to make sure the result is correct.

5.3 Test cases

5.3.1 Test case one

The first test case is made with the intention to measure the difference in performance between the exported library and the original HElib library. In the test case four plaintexts are made which are filled with random data. Four ciphertexts are created by encrypting the plaintext using the encrypt function in the EncryptedArray module. Note that the plaintexts are not overwritten or modified as these are later used to compare the results when the ciphertext are decrypted to make sure the result of the operations are correct. Within are loop the different arithmetic operations are performed on the different ciphertexts and the plaintexts, which includes adding the ciphertext to a random constant, multiply two ciphertexts and multiply by a random constant. The operation is done on the ciphertext and the corresponding plaintext. Once the operation is done the ciphertext is decrypted and compared to the plaintext.

5.3.2 Test case two

Measurement, analysis and comparison

6.1 testing

Discussion

7.1 asdf

Conclusion

8.1 Future Work

Bibliography

[1] M. Rouse, "What is homomorphic encryption?," *Retrived from Wired: https://www.wired.com/2014/11/hacker-lexicon-homomorphic-encryption/*, 2011.

[2] A. Greenberg, "What is homomorphic encryption?," *Retrived from Techtarget: http://searchsecurity.techtarget.com/definition/homomorphic-encryption*, 2014.

[3] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Annual cryptology conference*, pp. 505–524, Springer, 2011.

[4] M. D. R. Rivest, L. Adleman, "On data banks and privacy homomorphisms," *Foundations of Secure Computation*, pp. 169–177, 1978.

[5] C. Gentry, "Fully homomorphic encryption using ideal lattices," *STOC*, pp. 169–178, 2009.

[6] C. Gentry, "Toward basing fully homomorphic encryption on worst-case hardness," *CRYPTO*, pp. 116–137, 2010.

[7] M. Green, "A very casual introduction to fully homomorphic encryption," *Retrieved from https://blog.cryptographyengineering.com/2012/01/02/very-casual-introduction-to-fully/*, 2012.

[8] J. Sen, "Homomorphic encryption: theory & applications," *arXiv preprint arXiv:1305.5886*, 2013.

[9] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," *IBM Research (Manuscript)*, vol. 6, pp. 12–15, 2013.

[10] M. Naehrig, K. Lauter, and V. Vaikuntanathan, “Can homomorphic encryption be practical?,” in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pp. 113–124, ACM, 2011.

[11] N. P. Smart and F. Vercauteren., “Fully homomorphic encryption with relatively small key and ciphertext size,” *Public Key Cryptography - PKC*, vol. 6056 of Lecture Notes in Computer Science, pp. 420–443, 2010.

[12] J. Alperin-Sheriff and C. Peikert, “Faster bootstrapping with polynomial error.” Cryptology ePrint Archive, Report 2014/094, 2014. `https://eprint.iacr.org/2014/094`.

[13] L. Ducas and D. Micciancio, “Fhew: Bootstrapping homomorphic encryption in less than a second.,” *EUROCRYPT (1)*, vol. 9056, pp. 617–640, 2015.

[14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,” in *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pp. 3–33, Springer, 2016.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “Fully homomorphic encryption without bootstrapping.” Cryptology ePrint Archive, Report 2011/277, 2011. `https://eprint.iacr.org/2011/277`.

[16] Z. Brakerski, “Fully homomorphic encryption without modulus switching from classical gapsvp.” Cryptology ePrint Archive, Report 2012/078, 2012. `https://eprint.iacr.org/2012/078`.

[17] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan, “On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption.” Cryptology ePrint Archive, Report 2013/094, 2013. `https://eprint.iacr.org/2013/094`.

[18] C. Gentry, A. Sahai, and B. Waters, “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based.” Cryptology ePrint Archive, Report 2013/340, 2013. `https://eprint.iacr.org/2013/340`.

[19] S. Halevi and V. Shoup, “Helib,” *Retrieved from HELib: https://github. com. shaih/HElib*, 2014.

[20] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” *Journal of the ACM (JACM)*, vol. 56, no. 6, p. 34, 2009.

[21] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Tfhe: Fast fully homomorphic encryption library over the torus,” 2016.

[22] D. Abrahams, R. W. Grosse-Kunstleve, and O. Overloading, “Building hybrid systems with boost. python,” *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[23] A. Ibarrondo, “Pyfhel,” *Retrieved from Pyfhel: https://github.com/ibarrond/Pyfhel*, 2017.

[24] D. A. Joel de Guzman, “General techniques,” *Retrived from Boost Python docs: http://www.boost.org/doc/libs/1_65_1/libs*, 2017.

Appendices

tttt

Experiment	d	f	f	f
x	x	d	figure	f
x	x	d	f	f
x	x	d	f	f
x	x	d	f	f
x	x	d	f	f

Table A.1: asdf