UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

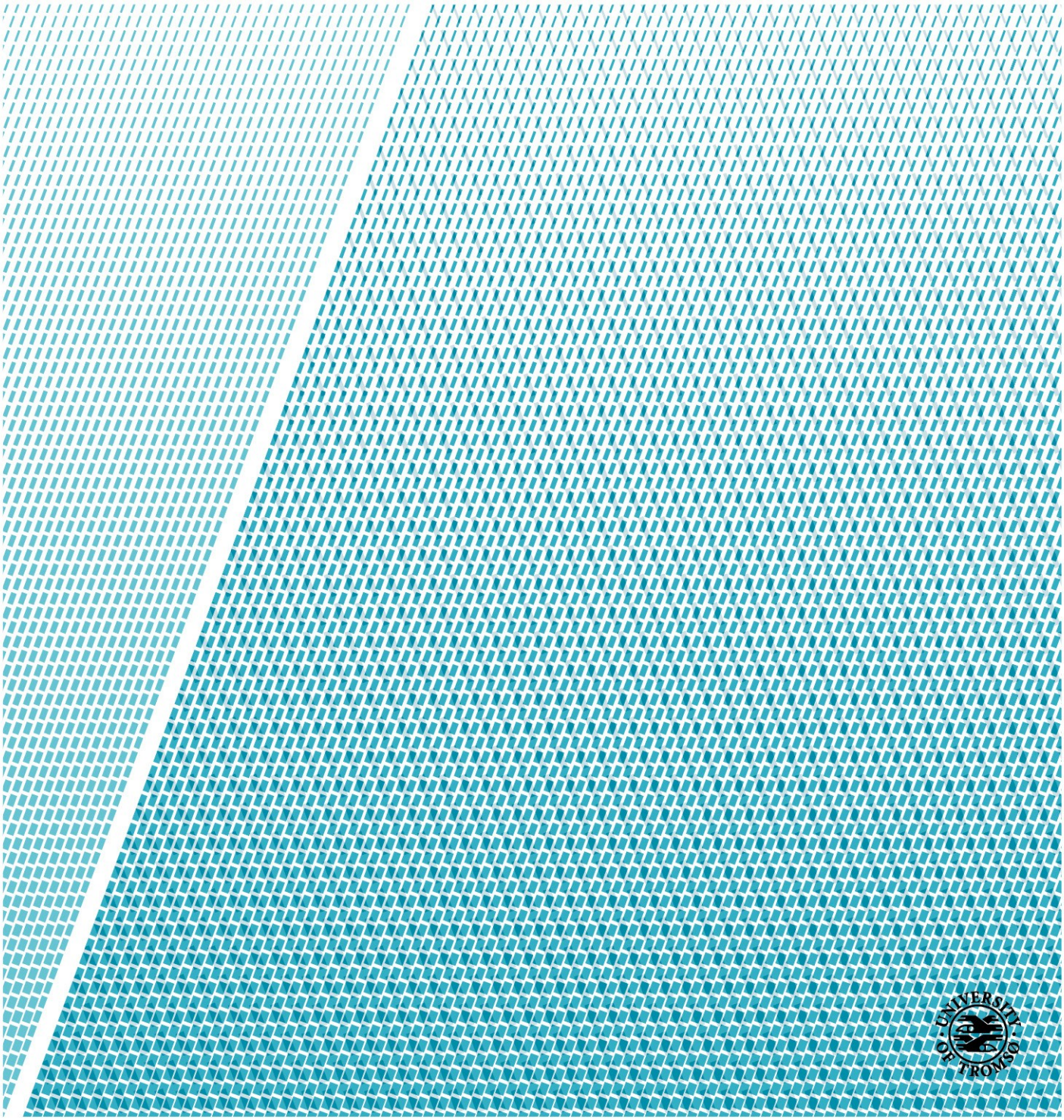
Faculty of Science and Technology, Department of Computer Science

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

Abstract i

Acknowledgements iii

Contents vi

List of Figures viii

List of Tables ix

1 Introduction 1

2 Background 3
2.1 Machine learning . . . 3
2.2 Deep learning . . . 4
2.2.1 Convolutional neural network . . . 4
2.3 Classification vs. detection . . . 5

3 Approach 6
3.1 Objectives . . . 6
3.2 Design Stage . . . 7
3.2.1 C++ wrapper for Python . . . 7
3.2.2 Create the API . . . 7
3.2.3 Find a suitable model for testing the API . . . 8
3.3 Implementation stage . . . 9
3.3.1 Choose a Fully Homomorphic Scheme implementation . . . 9
3.3.2 Wrap the code to Python . . . 10
3.3.3 Create a structured Python API . . . 13
3.3.4 Create tests to measure the performance . . . 14

4 Design 16
4.1 C++ wrapper for Python . . . 16

5 Implementation **17**
5.1 Wrapping the code 17

6 Discussion **18**
6.1 asdf 18

7 Conclusion **20**
7.1 Future Work 20

Bibliography **21**

Appendices **24**

A tttt **26**

List of Figures

2.1 Illustration of the convolutional layer . The red box is the input, and blue box is the result. The width and height of the input is preserved, but since the convolutional filter has 32 filters, the depth of the output volume is 32 pixels. 4
2.2 Vizualisation of four different image recognition tasks 5

List of Tables

A.1 asdf 26

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

This chapter describes relevant technical background and relevant work. We will first define the field of machine learning, and then describe the methods that have been used in this thesis. We will explain feature extraction for audio signals using spectrograms. We will also describe recent relevant work that uses the same machine learning methods and feature extractions methods as in this thesis.

2.1 Machine learning

Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc. [1].Mauris vel interdum nunc. Curabitur mi lectus, rhoncus venenatis ullamcorper quis, mattis volutpat dui. Proin at lectus ac metus ornare lacinia et pretium risus [2].

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{P + N}$$

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Recall

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris.

Precision

2.2 Deep learning

2.2.1 Convolutional neural network

Aenean rhoncus, nisi in suscipit aliquet, eros diam tempus nunc, non luctus turpis mauris ut lorem. Nulla sed scelerisque mauris. Figure 2.1 shows an illustration of the convolutional layer.

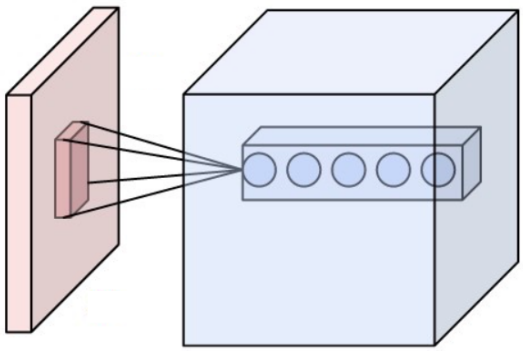
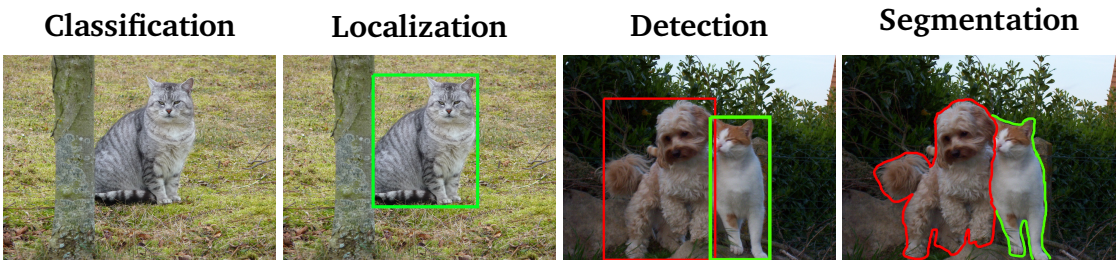


Figure 2.1: Illustration of the convolutional layer . The red box is the input, and blue box is the result. The width and height of the input is preserved, but since the convolutional filter has 32 filters, the depth of the output volume is 32 pixels.



(a) Cat **(b)** Cat **(c)** Cat, Dog **(d)** Cat, Dog

Figure 2.2: Vizualisation of four different image recognition tasks

2.3 Classification vs. detection

- **Classification:** Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc.
- **Object localization:** Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc. the size of the bounding box.
- **Object detection:**Donec id massa ac leo consequat euismod et a sem. Pellentesque sem justo, vulputate vel neque a, ultrices dapibus ipsum. Vestibulum orci orci, semper ut odio et, luctus condimentum nunc.

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) [3], Brakerski's scale-invariant cryptosystem [4], The NTRU-based cryptosystem due to Lopez-Alt, Tromer, and Vaikuntanathan (LTV)[5] and The Gentry-Sahai-Waters cryptosystem (GSW) [6].

Of the 2nd generation homomorphic cryptosystems there are three open source libraries. The HElib library [7] that implements the BGV cryptosystem with the GHS optimization, The FHEW library [8] which implements a combination of Regev's LWE cryptosystem [9] with the bootstrapping techniques of Alperin-Sheriff and Peikert [10] and the TFHE library[11] 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 [12]. 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 [13]. 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.

4.1 HElib design

Implementation

5.1 Wrapping the code

Discussion

6.1 asdf

Conclusion

7.1 Future Work

Bibliography

[1] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, pp. 115–118, Feb 2017. Letter.

[2] "Homesite quote conversion." `https://www.kaggle.com/c/homesite-quote-conversion`. Accessed: 2017-30-05.

[3] 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`.

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

[5] 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`.

[6] 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`.

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

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

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

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

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

[12] 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.

[13] A. Ibarrondo, "Pyfhel," *Retrieved from Pyfhel: https://github.com/ibarrond/Pyfhel*, 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