Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2016

Project Title:	**Homomorphic encryption: Cryptography for Cloud computing**
Student:	**Q.D. MCGAW**
CID:	**00746622**
Course:	**4EM**
Project Supervisor:	**Dr Wei Dai & Dr Cong Ling**
Second Marker:	**Professor Athanassios Manikas**

Abstract

This project concerns the research and development of a real-use application of homomorphic encryption for cloud computing. The application takes advantage of the various possibilities and limitations of present homomorphic encryption schemes and programming libraries to remain usable in terms of time. The foundations of the application rely on the design of binary operations using homomorphic encryption. All the binary logic gates and various binary blocks were developed and adapted to provide enough functionalities to the application. The project focuses on providing features to cloud computing such as calculating averages on large amounts of encrypted numbers in a relatively short and decent time. The result is an application program interface written in C++ allowing to perform various operations on integers. It thus shows homomorphic encryption can be used today for simple operations if the security is more important than the speed of execution.

Acknowledgements

The idea of this project was suggested to me by Dr. Wei Dai after we met for a presentation I was giving about my industrial placement. Not only he proposed me this fantastic topic, but he also offered me a book on *homomorphic encryption and applications*. I can't thank him enough for introducing me to homomorphic encryption, which allowed me to develop this project. I would also like to point out he successfully recommended me to New York University for a Computer Science postgraduate degree, and I will always be grateful for this. I would also like to thank Dr. Cong Ling for endorsing my project.

My industrial placement supervisor, Christian Bodt, introduced me to cryptography and security, and taught me many programming techniques which were useful in this project. I would hence like to thank him for his patience and generosity.

Several useful email and forum conversations also took place during the development of this project, more specifically with Shai Halevi from IBM and his Github repository HElib issues section, and Wei Dai from the Worcester Polytechnic Institute who updated me about the status of his software library cuHE.

Alexandra Rouhana, a fellow student in electrical and electronic engineering at my university, also helped me overcome several design restrictions for my project through many discussions.

Finally, I would like to thank my father, Didier McGaw, my mother, Sylvie Portenseigne, and my step-mother Pascale McGaw, who is no longer with us, for their encouragements.

Contents

Chapter 1

Introduction

Powerful, scalable, cheap and easily accessible, cloud computing is becoming increasingly popular to manage data. Companies and individuals tend more and more to outsource their data and its management to cloud computing. This type of remote computing is quite recent and really took off in 2006 with the introduction of the Elastic Cloud Computing by Amazon.

There is however a strong security and privacy flaw in the cloud computing concept. Whereas in cloud storage, a client can encrypt its data before uploading it to the remove server, or "cloud", and will only need to decrypt it once he or she needs it, this is not the case for cloud computing. Indeed, as its name indicates, cloud computing performs some operations on the data. The security protocol today is thus the following: the client encrypts the data, sends it to the cloud computers which then decrypt it in order to process the data received. The result is then encrypted and sent back to the client which in turn can decrypt this one. The communication channel is thus secured, but the cloud computers have access to all of the client's data, query and corresponding result. If the data is sensitive, this raises many security and privacy concerns.

There is not a concrete solution yet, but there exists a technology in a booming evolution which can address satisfactorily these concerns. It is called fully homomorphic encryption (FHE), and conceptually consists in allowing to perform arbitrary computations on encrypted data, also called ciphertext(s), to produce an encrypted result which, once decrypted, matches the result of the operations performed on the original data, also called plaintext(s). This technology became a reality in September 2009 when Craig Gentry, a doctoral student at Stanford University, released a dissertation on the first realisable FHE scheme (Reference 1).

Any computation can be expressed as a combination of logic gates, which can be derived from the addition (XOR) and multiplication operations (AND). As these two operations are supported by the FHE scheme, theoretically any function could be performed on encrypted data. This would revolutionize cloud computing and solve all the privacy and security concerns previously described. The cloud computer using FHE would not access the plaintext data at all, and only the client would be able to decrypt it.

But the reality is sadly different. Even with the many advances and new FHE schemes, the processing time needed by the latest FHE implementations is still millions of times greater than the time required for usual plaintext processing. This limitation often categorises FHE as a non-realistic method, which should not be used until it becomes much faster.

One of the aim of this project is to show present FHE implementations can already be used for some cloud computing scenarios. Another aspect is to expose the limits of FHE calculations. In order to do so, various operations such as the Euclidean division or the multiplication were implemented in binary homomorphically, to concretely test possibilities and limitations of current FHE. Overall, this depicts what can be done homomorphically in a cloud computing context today.

The report is structured as follows. The next section is the background of the project, covering the working of FHE, the relevant external work and the choices made for this project. The chapter 3 summarises the aim of this project and its deliverables. The next chapter contains the design aspect of the project whilst the chapter 5 explains the implementation of the designs. The chapter 6 highlights the testing methods used as well as how the implementation was functionally verified. Results and their analysis obtained from testing the program implemented are shown in chapter 7. The comparison between what has been achieved and what was aimed to be done will then be carried out with explanations in the evaluation chapter. The chapter 9 covers further work which can be carried out from the final state of this project, and is followed by the conclusion chapter. Finally, the chapter 11 is a user guide for setting up and using the program developed.

Chapter 2

Background

In order to develop the application for this project, several software libraries implementing the latest schemes of FHE were found. These actually serve as a proof of practical use of their respective scheme. Indeed, some schemes such as Gentry's original FHE scheme (reference 1) could never be implemented due to its space complexity. First of all, we will start with a quick explanation of this initial FHE scheme designed by Craig Gentry in 2009, to better understand the more recent optimisations. The main advances made in the field will then be highlighted. The libraries found will be discussed regarding their advantages and drawbacks.

2.1 Craig Gentry's initial FHE scheme

Craig Gentry started with a somewhat homomorphic encryption, abbreviated by SwHE, in which ciphertexts have some "noise", which comes from randomness added for security purposes. The noise of a ciphertext grows with each additional operation performed on this one. The ciphertext can't be decrypted anymore once the noise grows above a certain threshold. To make of his SwHE scheme a FHE scheme, Gentry developed a technique called bootstrapping which decrypts and recrypts the ciphertext at each operation to reduce its associated noise. This allows an infinite amount of operations to be performed on ciphertexts. However, each bootstrapping is slow so performing an addition or multiplication will always be slow. Since then, many improvements have been made to make a more efficient, usable FHE scheme.

2.2 FHE new schemes and improvements

Apart from several small enhancements such as a reduction of the size of the FHE keys, there are today three main improvements since Gentry's first FHE scheme (reference 1). The first one is the support of SIMD operations (reference 2). Abbreviation of Single Instruction Multiple Data, SIMD is usually implemented in hardware chips to process several vectors of multiple elements with a common instruction component-wisely. The following figure illustrates its working.

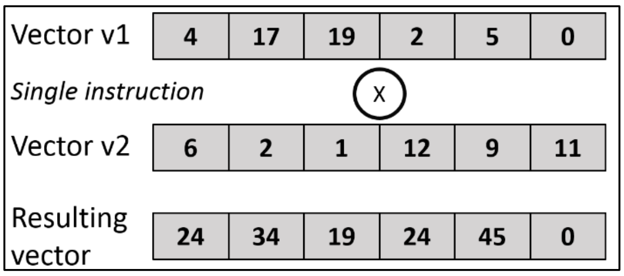


Figure 2.2.a: *SIMD operation of multiplication on vectors v1 and v2*

In the FHE case, the SIMD operations are actually performed on the ciphertexts in the plaintext space. Before being encrypted, a plaintext is not a single value but a vector of plaintext elements. The encrypted vector then results in a "packed" ciphertext. The SIMD operations are executed as shown in the example figure below.

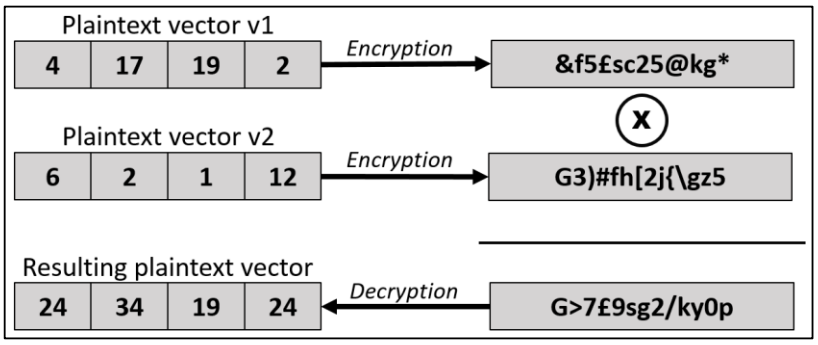


Figure 2.2.b: *SIMD operation of multiplication on two ciphertexts with FHE*

This improvement allows to perform an operation on multiple entries at once and thus reduces either the time cost associated with bootstrapping or the number of levels needed in the case of a leveled FHE (see part 4.5).

The next enhancement is tightly related to the introduction of SIMD operations support for FHE. SIMD for FHE initially only supported the addition and multiplication operations. The limitation was that if one needs to perform an operation with the element of one plaintext vector at the position 2 and the element of another plaintext vector at the position 3, the ciphertexts had to be unpacked and then repacked, taking a significant amount of time. The solution brought was a permutation network allowing to permute elements without needing to unpack and repack everything (reference 3). Not only this provides great flexibility and efficiency, but also make the FHE SIMD very usable.

The modulus switching was introduced with the BGV scheme (reference 4), aiming to provide an alternative to the slow bootstrapping. The modulus switching is the key element behind a new FHE scheme called the BGV scheme, or RLWE (Ring learning with error) scheme, or also leveled FHE. Whilst bootstrapping allows to perform an infinite number of operations on a ciphertext, the leveled FHE sets an arbitrary limit to the number of operations which can be performed on a ciphertext. This may first appear as a drawback, but is actually a lot faster than bootstrapping in some cases. In this scheme, there is a predefined number of levels which is proportional to the maximum number of operations to be performed on the ciphertext. This one is actually proportional to the accumulated noise added with each operation. Note that addition adds very little noise whilst multiplication adds significant noise to the ciphertext. Therefore, the number of multiplication operations to be performed on a ciphertext should determine the number of levels. Note that for a high number of levels, bootstrapping becomes faster than the BGV scheme.

All these improvements added to the various speed and space enhancements made during the last six years gave birth to more usable FHE schemes. Many of them are implemented in open-source libraries and software packages and hence available for further development.

2.3 FHE software libraries

The most complete, portable and well maintained library is called HElib (reference 5). It is written in C and C++ and implements a FHE scheme including both the BGV scheme with modulus switching (reference 4) and bootstrapping, SIMD operations (reference 2) and the permutation network optimisation (reference 3). It also supports several other speed enhancements such as multi-threading and propose an easy access to tweaking the many optimisations. It is also well documented (references 6, 7) and is maintained by Victor Shoup (NYU, creator of NTL library), Shai Halevi (IBM Watson research centre, and one of the pioneers of FHE) and several other participants on GitHub. HElib requires the NTL mathematical library (reference 8) as well as the GMP library (GNU Multiple Precision Arithmetic)(reference 9).

As the HElib library is quite low-level, the PhD student Grant Frame released in June 2015 an open-source Integrated Development Environment (IDE) for HElib called HEIDE (reference 10). It uses Python for its interface layer and links it to a compiled wrapper written in C++ for the main HElib functions. On one hand, the described C++ wrapper served as a starting point to the application programming interface developed in the project. On the other hand, this wrapper was quite limited and was not fitting the purpose of the project. The python program using it was thus not used at all in this project.

Another software package "An R package for fully homomorphic encryption", released by Louis Aslett (Oxford) in 2015 (reference 11), implements the Fan and Vercauteren scheme (reference 12). It can thus perform the addition and multiplication operations, as does HElib, but not the subtraction. Its source code is high performance C and C++ tailored to multi-core multi-threading computations, making it quite attractive. It also supports SIMD operations. However, this library was not chosen because I had no experience in R language and the downloadable source code was too complex to understand. HElib complemented with the C++ wrapper of HEIDE was thus preferred over this library.

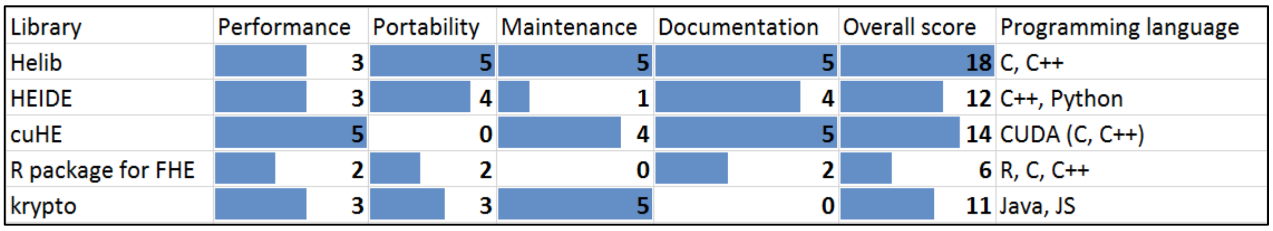
Wei Dai from the Vernam Group at the Worcester Polytechnic Institute developed and released in January 2016 an open-source FHE library called cuHE (reference 13). It implements the Doroz-Hu-Sunar (DHS) SwHE scheme (reference 14) based on the Lopez-Tromer-Vaikuntanathan (LTV) scheme (reference 15). The main difference with the previous implementations is that it uses Nvidia CUDA-enabled graphic processing units (GPU) to parallelise and accelerate the homomorphic operations even further. It is 10 to 100 times faster than the other implementations, hence widening the range of applications for FHE. It requires the NTL library and the OpenMP API to run. However, this library is not very portable as it requires an Nvidia GPU. As some of the project's work was already carried using HElib, cuHE was not used. It however remains a very interesting implementation for further work.

There is another open-source library, krypto (reference 16), released in November 2015 by the company kryptnostic. It is actively developed and might incorporate an interesting FHE scheme. However, no research paper or documentation was found, making it undesirable.

Leo Ducas and Daniele Micciancio released the open-source FHEW library in January 2015 (reference 17) which achieves a 1-bit NAND gate homomorphic operation in less than a second with bootstrapping. However, this library is very narrowly focused on performing a fast NAND gate operation only. It does not provide many settings or options and is thus not suitable for the project.

The last library found was released by Jean-Sebastien Coron in 2012 and implements the DGHV FHE scheme in Python with the SAGE mathematical library (reference 18). However, it has not been modified since then and does not incorporate the latest enhancements such as the ones present in HElib. It was therefore not appropriate for the project.

The following table summarises the libraries with some of their characteristics such as the programming language they use.



Library	Performance	Portability	Maintenance	Documentation	Overall score	Programming language
Helib	3	5	5	5	18	C, C++
HEIDE	3	4	1	4	12	C++, Python
cuHE	5	0	4	5	14	CUDA (C, C++)
R package for FHE	2	2	0	2	6	R, C, C++
krypto	3	3	5	0	11	Java, JS

Figure 2.3.a: *Summary table of FHE software libraries of interest*

In figure 2.3.a, each library is assigned a score ranging from 0 to 5 on different criteria such as performance. The overall score column represents the cumulative score for each library. HElib was thus chosen as the base of the project because of its highest score. Moreover, HEIDE was also used as a starting point to the application programming interface (API) developed. cuHE is also a really great library and should be explored in future works.

Chapter 3

Requirements capture

The project should show homomorphic encryption can be used for simple operations today. It should provide simple but useful operations for cloud computing willing to use FHE, such as the average operation. The average is relatively simple and only requires a sequence of additions and a division. The project will also focus on showing the possibilities and limitations of current FHE, especially through testing, measurements and plots. This should hence determine what feature can be used in a decent time with current technology.

Theoretically, averaging numbers is simple. For homomorphic applications, it is not at all. First of all, the basic operations provided by HElib are very limited as it will be shown in chapter 4. All the traditional logic gates and binary circuits necessary to reach the addition and the division operations will have to be developed and adapted to the homomorphic encryption scheme.

This project should also provide a good starting point for many horizons for future work, regarding the addition of features, the increase in performance, and a better ease of use.

Chapter 4

Analysis and Design

4.1 The limits of the software library HElib

The software library HElib, as all the other homomorphic encryption libraries, provide the addition and multiplication operations on ciphertexts in the plaintext space. The homomorphic subtraction and negation operations are also supported by HElib. However, all these operations are done in a field F_{p^d} where p is a prime number, and their results are modulo p. Homomorphic additions and multiplications are only meaningful if the result satisfies $r < p$, condition which can't be checked homomorphically. The subtraction and negation operations are meaningless as their result is also modulo p, which is undefined. Furthermore, all these homomorphic operations are pointless if used in their raw form. In addition to these limits, numbers can't be compared, conditions statements can't exist, and more complex operations such as division can't be achieved.

4.2 Unlocking the potential of HElib

The solution to unlock the potential from these operations was to implement binary logic compatible with FHE from the ground up. In order to do so, p is set to $p = 2$, and $d = 0$, hence allowing values to be either 0 or 1. In the binary context, subtraction is equivalent do addition, and negation does not change the binary value, as shown in the binary equations below.

$$\begin{cases} 0+0=0-0=1+1=1-1=0 \\ 1+0=1-0=0+1=0-1=1 \\ \quad -0=0 \quad and \quad -1=1 \end{cases}$$

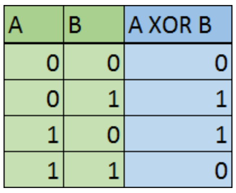
Furthermore, only the addition and multiplication homomorphic operations are useful in binary.

Now, the addition operation and the multiplication operation actually implement a XOR (exclusive OR) and an AND logic gate, as shown in the equations and truth tables below.

- *Addition*

  $0+0=0$
  $0+1=1$
  $1+0=1$
  $1+1=0$

XOR truth table



A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

- *Multiplication*

AND truth table

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

This makes a good starting point, but other logic gates such as the OR gate are required. To be able to derive all the other logic gates, the NOT logic gate is also needed. The NOT logic gates takes a binary input and switches it. In order to implement it, the only solution is to add the input to a constant 1, so that a 0 becomes a 1 and 1 becomes a 0.

- *Addition with constant* 1

$$0 + 1 = 1$$
$$1 + 1 = 0$$

NOT truth table

A	NOT A
0	1
1	0

This raises some security concerns as it means the user will have to provide a ciphertext of reference for 1. However, the risk is mitigated thanks to the SIMD support of HElib as explained later.

From these three logic gates, all other logic gates can be derived: NAND, OR, NOR and XNOR gates.

4.3 Level parameter and complexity

The next stage is to develop larger binary blocks from these logic gates. However, it is important to highlight the main limitation of FHE now. As explained in Chapter 2, the homomorphic multiplication operation adds a significant amount of noise to the ciphertext in comparison with the homomorphic addition operation. Indirectly, the number of homomorphic multiplications is thus proportional to the time complexity of a homomorphic function, and is thus called *complexity* in the following.

To prevent the noise from growing too big, HElib proposes a levelled homomorphic encryption, which is more flexible and usually faster than the classic FHE scheme with bootstrapping. It consists in setting a level parameter L at the key generation stage. This parameter is proportional to the maximum number of homomorphic operations to be performed on a ciphertext. Also, the lower L is, the faster the homomorphic operations will be. This thus gives a great compromise between short and quick operations, and long and slow operations.

To highlight the relationship between the level parameter, the number of multiplications (*complexity*) and the time needed for a function to execute, the following graph was plotted from real time measurements.

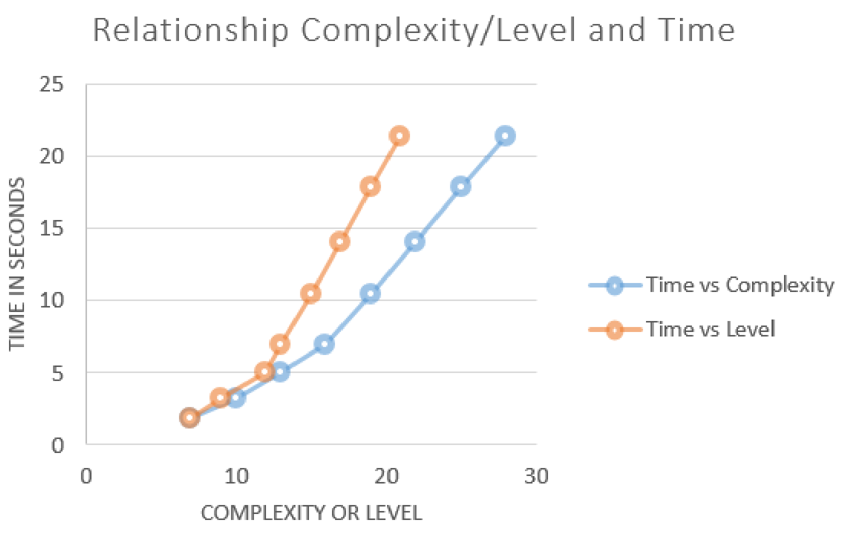


Figure 4.3.a: *Execution time plotted against the complexity and the level L*

The graph shows that the complexity is usually a good indicator to determine the minimum level required. This also shows that the time required for a homomorphic function to complete is almost linearly proportional to its complexity.

4.4 Logic gates and complexity

The complexity and its relationship with the level parameter previously described implies that some logic gates using multiplications should be avoided as much as possible. The following table describes the number of homomorphic multiplications needed for each of the logic gates.

Logic Gate	XOR	AND	NOT	NAND	OR	NOR	XNOR
Multiplications	0	1	0	1	1	1	0

Figure 4.4.a: *Homomorphic logic gates and associated required multiplications*

XOR, NOT and XNOR gates only uses the addition operation and do not affect significantly the noise of the ciphertext. Hence they do not require a higher level *L* usually. The other gates however all require 1 homomorphic multiplication and add a lot of noise to the ciphertext; these should thus be carefully used.

An important design was for the OR gate, which was originally implemented with 3 NAND gates so requiring a complexity of 3. It was first found that a NOR gate could be implemented with the following circuit.

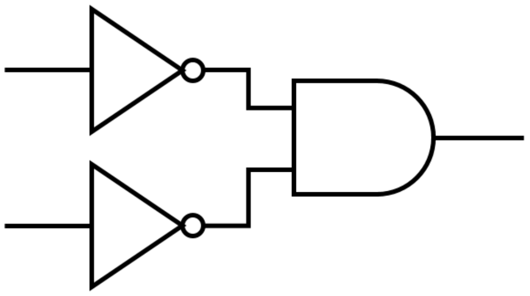


Figure 4.4.b: *Digital circuit design for the NOR gate*

This only requires 2 NOT gates and 1 AND gate, resulting in only 1 homomorphic multiplication. The OR gate can then be designed from this circuit by adding a NOT gate to the output of this NOR gate, resulting in a much more efficient homomorphic logic gate with a complexity of 1 only.

4.5 SIMD operations

As described previously, HElib supports SIMD operations, where several values can be packed into a single ciphertext, and these can undergo a common instruction with another packed ciphertext. This has several advantages:

- No waste of space in the ciphertexts
- Ciphertexts are shorter
- More data can be processed homomorphically and in parallel

This enhancement is optional in HElib but was decided to be used to make full profits of expensive homomorphic operations.

The SIMD mode of operation requires N integer values -0 or 1 here- to be encrypted and packed into a ciphertext, where the minimum value of the number of plaintext slots N depends on the level parameter L and the security parameter k. This is due to the fact that ciphertexts grow larger as the level parameter or the security parameter increases. A lower bound s for N can also be set at the generation of the key pairs. Setting s higher than the minimum value of N is useful only if the time per operation is more important than the overall time. Indeed, for example, performing an OR gate on 100 slots ciphertexts would take 10 units of time, whilst performing it on 200 slots would take 17 units of time, a faster approach *per operation.* This will be investigated more deeply in part 7.3.

The packed ciphertexts are used such that each slot is independent from the other ones. The following diagram illustrates how an AND operation is performed with SIMD.

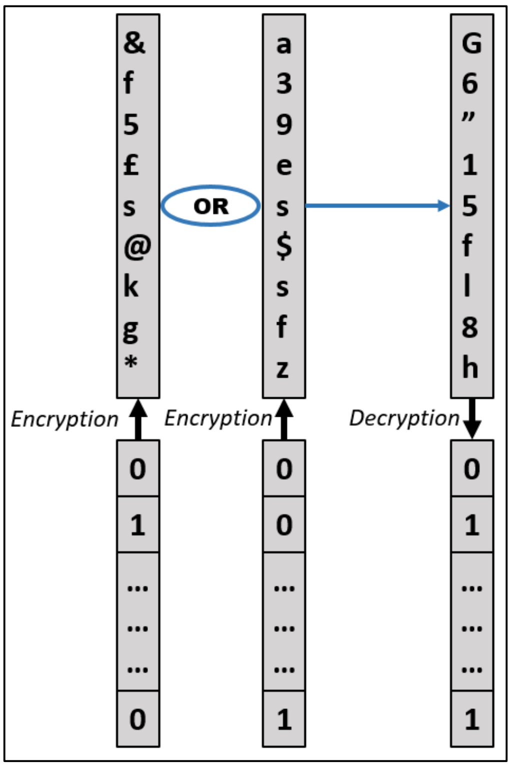


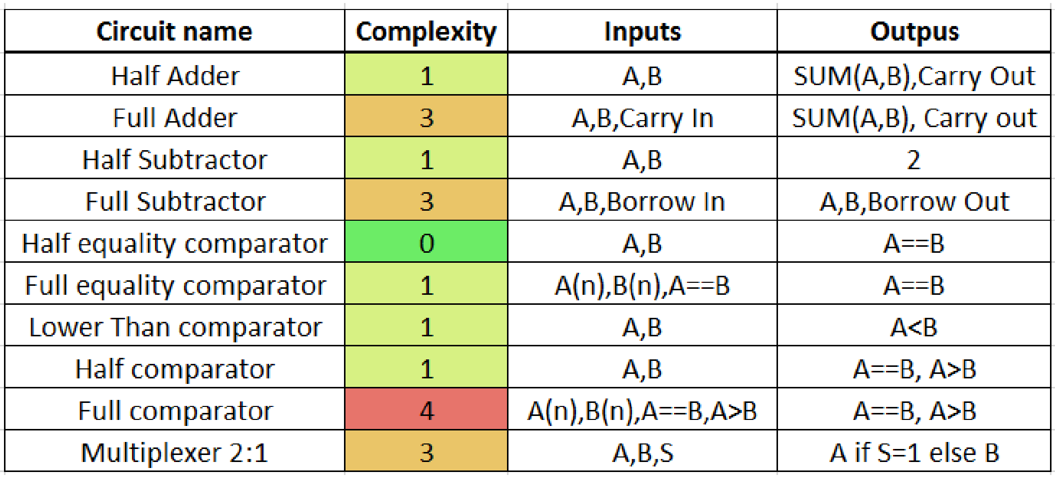
Figure 4.5.a: *Conceptual working of homomorphic OR gate using SIMD*

For instance, this independence was maintained in all the operations developed. The first “row”, or first plaintext slot, of a ciphertext will be part of an operation with other first rows of ciphertexts and will never interact with other rows.

4.6 Designing homomorphic combinational circuits

The next stage was to use the logic gates to build combinational circuits taking up to 3 bits as input and producing up to 2 bits of output. This followed the same SIMD method as before, where the M inputs are actually represented by M ciphertexts containing *N* independent bits. The output ciphertexts (or *N* independent output bits) replace some or all of the M input ciphertexts.

The circuits hence designed are listed in the table below, together with their respective complexity, inputs and outputs.



Circuit name	Complexity	Inputs	Outpus
Half Adder	1	A,B	SUM(A,B),Carry Out
Full Adder	3	A,B,Carry In	SUM(A,B), Carry out
Half Subtractor	1	A,B	2
Full Subtractor	3	A,B,Borrow In	A,B,Borrow Out
Half equality comparator	0	A,B	A==B
Full equality comparator	1	A(n),B(n),A==B	A==B
Lower Than comparator	1	A,B	A<B
Half comparator	1	A,B	A==B, A>B
Full comparator	4	A(n),B(n),A==B,A>B	A==B, A>B
Multiplexer 2:1	3	A,B,S	A if S=1 else B

Figure 4.6.a: *Descriptive table of the combinational circuits designed*

All these combinational circuits were designed to be used at a higher level for n bits numbers instead of single bits. They allow to add, subtract and compare n bits numbers if configured together properly. The multiplexer 2:1 is also a key element for the division as it actually allows to do an if-else statement.

As shown in the table, the full comparator has the highest complexity of 4 and should be used minimally. This is because it uses the *Lower Than comparator*, the *full equality comparator*, and logic gates, aggregating to 4 homomorphic multiplications. Note that a *Lower Than comparator* was designed instead of a *Greater Than comparator* because this one requires a non-necessary complexity of 4, instead of 1.

The full adder and full subtractor are essentially the same except they operate in reverse. This is why they have the same complexity of 3, as they both use two of their respective half combinational circuit with an OR gate.

The multiplexer requires a complexity of 3 as it uses 3 NAND gates, and the full equality comparator is quite cheap in terms of complexity because it only uses an AND gate on top of a XOR gate.

4.7 Designing homomorphic sequential circuits

Once all the necessary combinational blocks are designed, sequential circuits can be developed. The objective now is to be able to process variable sized binary numbers, such as adding a 4 bit number with a 3 bit number.

For the homomorphic sequential circuits, the clock is replaced by plain software code (C++) which calls a homomorphic combinational circuit in a for loop for example. To represent an n bit number, n ciphertexts are needed. The following diagram shows the plaintext representation of N numbers of n bits.

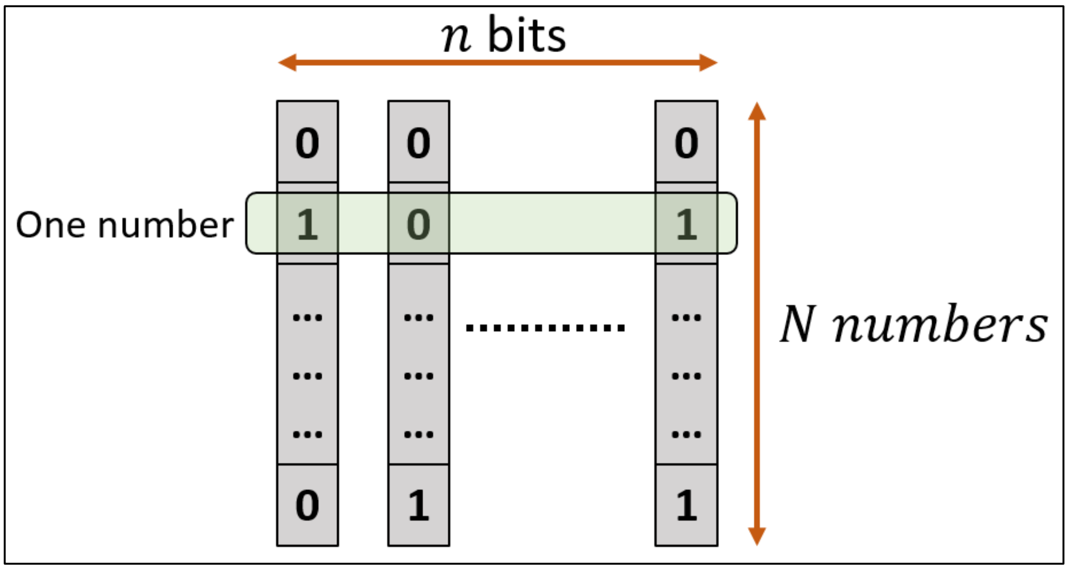


Figure 4.7.a: *Representation of n bits numbers packed in ciphertexts*

Each of the designed sequential homomorphic circuits hence takes n ciphertexts for each number (or more precisely N numbers packed).

The following table highlights the sequential circuits developed with their complexity and purpose. Here again, n represents the number of bits of a binary number, as shown on figure 4.7.a.

Circuit name	Complexity	Purpose
Ripple Carry Adder	3n + 1	Adds two n bits unsigned integers
Ripple Borrow Subtractor	3n + 1	Subtracts two n bits integers
Ripple comparator	4n - 3	Compares two n bit unsigned integers
Multiplexer N:1	3n	Selects one of two n bit integers
Ripple equality tester	n - 1	Tests if two n bit integers are equal
Right Shifter	0	Right shift an n bit number
Left Shifter	0	Left shift an n bit number

Figure 4.7.b: *Table of sequential homomorphic circuits designed*

As before, the ripple carry adder and ripple borrow subtractor act similarly. They both use their respective half combinational circuit, with a complexity of 1, followed by n iterations using their respective full combinational circuit, requiring each time a complexity of 3. They hence require a complexity of $3n + 1$.

Note also that the ripple carry adder and the ripple borrow subtractor add one extra most significant bit to the resulting binary number, respectively to store the eventual overflow bit and to store the sign bit.

Floating numbers support was not designed due to the already quite high time needed for these sequential circuits to complete for numbers of 8 bits for example.

Similarly, the ripple comparator uses the half comparator once and then the full comparator for the remaining $n-1$ bits, hence requiring a complexity of $1+4(n-1)$. This is the highest complexity and this circuit should thus be avoided as much as possible.

The multiplexer N:1 consists in a single select line (1 bit) determining if the n bits number A or the other number, B, should be outputted. It is based on a simple iterative loop using the combinational multiplexer 2:1 previously described for each bit of A and B. Its complexity is thus the number of bits –or the number of iterations- times the complexity of the 2:1 multiplexer, hence $3n$.

The ripple equality tester uses the half equality comparator once and then the full equality comparator combinational circuit in an iterative loop, similarly to the other ripple sequential circuits. Its complexity is only $n-1$ because the full combinational circuit requires 1 multiplication and is operated $n-1$ times.

Finally, the right and left shifter only switch around, copy and erase ciphertexts. As shown in figure 4.7.a, it is clear that shifts can be easily done without any homomorphic operations at all. These are thus some complexity free operations.

It is important to note all the sequential circuits were designed with the aim to use them for various size of binary numbers, for flexibility and testing purposes. The following figure plots and summaries the complexity of all the sequential circuits designed as a function of the number of bits n of the input number(s).

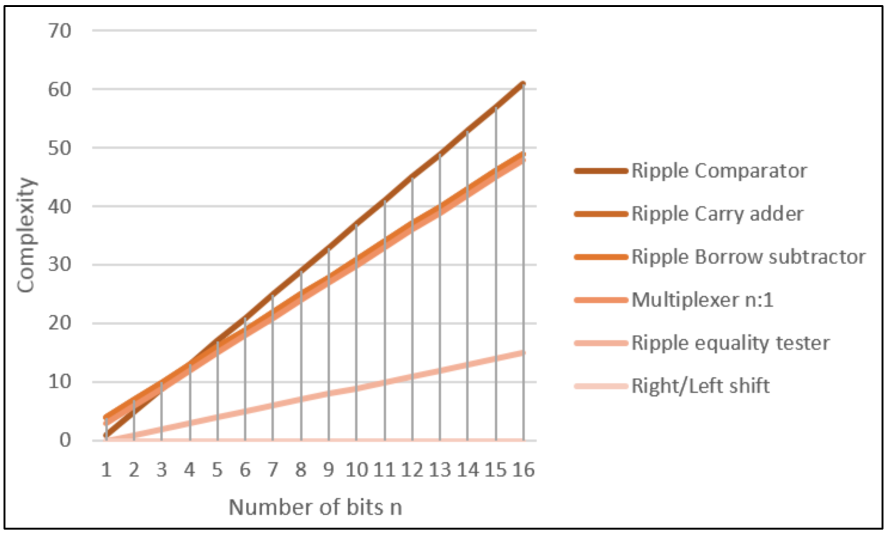


Figure 4.7.c: *Complexity vs Number of bits, for all sequential circuits designed*

The figure 4.7.c is useful to have a quick idea of the complexity of a certain circuit for a specific number of bits. This also helps to determine a reasonable value for the level parameter L. Note also the ripple carry adder, ripple borrow subtractor and 2:1 multiplexer circuits are actually worse than the ripple comparator in terms of complexity for $n < 4$.

4.8 Determining the level parameter

Homomorphic sequential circuits have a complexity varying with the number of bits of the input number(s). Even if the level parameter grows with the complexity, it is not totally proportional. Various other factors such as the number of plaintext slots can highly influence the minimum level parameter.

For instance, the only solution to determine the minimum level parameter was the trial and error method. For each sequential circuits and for each number of bits ranging from 1 to $n = 16$, the minimum level parameter yielding correct results was recorded. An online linear regression calculator from graphpad software (reference) was used with the data recorded for each sequential circuit. This provided a best fit function yielding a level parameter for a given number of bits. These various functions can then be combined to give a specific function for a more complex homomorphic circuit using various sequential circuits, for different number of bits. However, the trial and error method always yields slightly better results and should thus be used for a particular case.

4.9 Designing the binary multiplication operation

The homomorphic binary multiplication circuit takes n ciphertexts per number, so a total of $2 \times n$ ciphertexts for two n bits numbers. It uses a rather simple algorithm called the *shift and add* or *long multiplication*. An example of it is shown in the figure below.

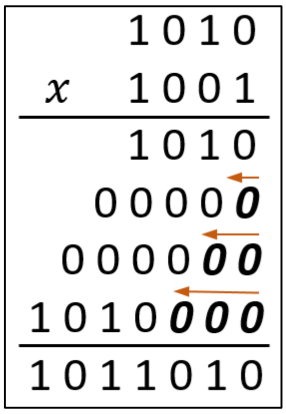


Figure 4.8.a: *Example of the long multiplication algorithm*

It consists in a combination of left binary shifts and bitwise AND between the two numbers, followed by a final addition using the ripple carry adder previously designed. The algorithm was slightly modified to have the lowest homomorphic complexity, as shown in the pseudo code below.

```
Let A and B be two n bits binary numbers.
Let Cn be an array of n binary numbers each made of n bits.
for each number C in Cn do
    Set C to A ;
for each binary number with index row in the range 1 to n do
    for each bit with index bit in the range 1 to n do
        Set Cn[row][bit] to Cn[row][bit] AND B[b];
    Binary shift left Cn[row] by row bits;
Set Cn[row][1] to 0 ;
Set A to Cn[0] (to avoid the first addition operation);
Append a Most significant bit 0 to A ;
for each binary number index row in the range 2 to n do
    Add Cn[row] to A with the ripple carry adder ;
A contains A x B
```

Figure 4.8.b: *Long multiplication algorithm adapted pseudo-code*

In this pseudo code, all the memory operations such as appending bits or setting a binary number to another binary number are not done homomorphically and thus have a complexity of 0. The overall complexity of the multiplication is only due to the ripple carry adder and to the AND logic gate used once on each ciphertext. The complexity can thus be calculated as follows:

$$\begin{aligned} Complexity &= Complexity\ of\ AND\ gate + Iterative\ complexity\ of\ Ripple\ carry\ adder \\ &= 1 + Iterative\ complexity\ of\ Ripple\ carry\ adder \end{aligned}$$

Now the complexity of the ripple carry adder is $3n + 1$, but here, because the ripple carry adder extends the size n in bits of the accumulator at each iteration, from $n + 1$ to $2n - 1$, the iterative complexity is $(3(n + 1) + 1) + (3(n + 2) + 1) + \cdots + (3(2n - 1) + 1)$.

This is a sum of an arithmetic progression which can be expressed as
$\frac{n-1}{2}\big((3(n + 1) + 1) + (3(2n - 1) + 1)\big) = 4.5n^2 - 3.5n - 1$

The overall binary multiplication complexity is thus $4.5n^2 - 3.5n$, where n is the number of bits of the binary numbers.

In terms of time, this design requires $5.5n^2 - 3.5n - 1$ multiplications as n^2 multiplications have to be performed for the AND gate operations. The following figure shows the homomorphic complexity and the total number of multiplications required varying as functions of the number of bits n of the binary numbers.

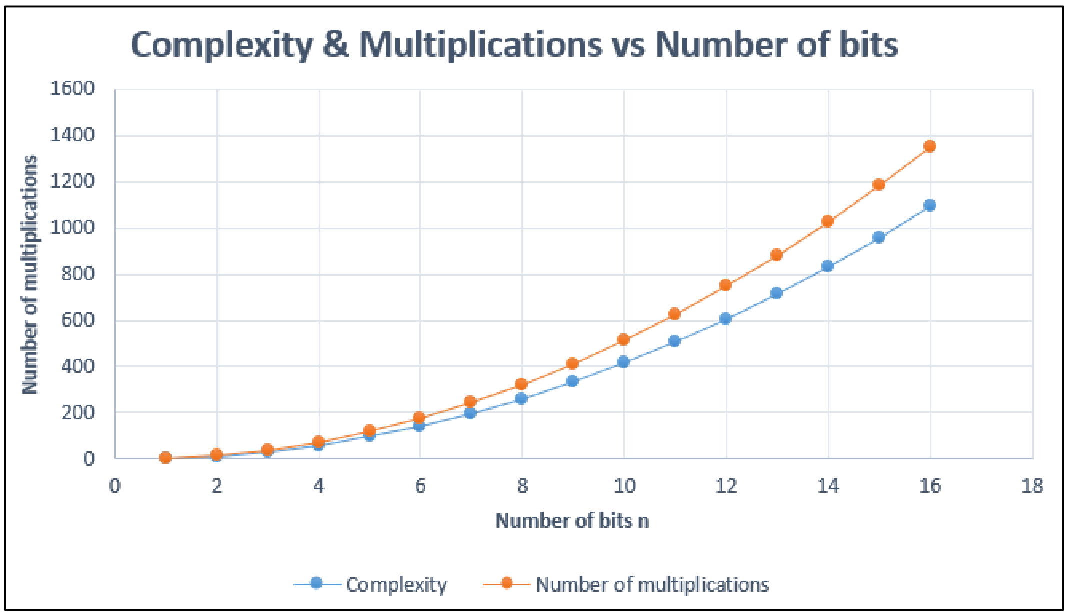


Figure 4.7.c: *Complexity and total number of homomorphic multiplications performed vs number of bits n, for the multiplication circuit*

4.10 Designing the Euclidean division

The homomorphic binary Euclidean division circuit is the most complex circuit. As for the multiplication, it takes n ciphertexts per number, so a total of $2 \times n$ ciphertexts for two n bits numbers. It is based on an enhanced version of the restoring division algorithm (reference 20). The following pseudo code explains its operation.

```
Let A be the n bits binary dividend and B the n bits binary divisor ;
Let X be the concatenation of n zeros and A, where A occupies the lower
half of X ;
Left shift X by 1 bit and set the LSB of X to 0 ;
for $i$ in the range from 1 to $n$ do
    Set R to the n most significant bits of X minus B;
    if $R$ is strictly negative then
        Left shift X by 1 bit and set the LSB of X to 0;
    else
        Set the n most significant bits of X to R ;
        Left shift X by 1 bit and set the LSB of X to 1 ;
Set A to the n most significant bits of X;
Set B to the n least significant bits of X;
Right shift A by 1 bit;
A contains the remainder and B contains the quotient;
```

Figure 4.8.b: *Long multiplication algorithm adapted pseudo-code*

For a more concrete explanation, the following figure explains the steps needed to divide 5 by 3 using this algorithm. It also highlights the homomorphically complex parts and the memory and shift operations.

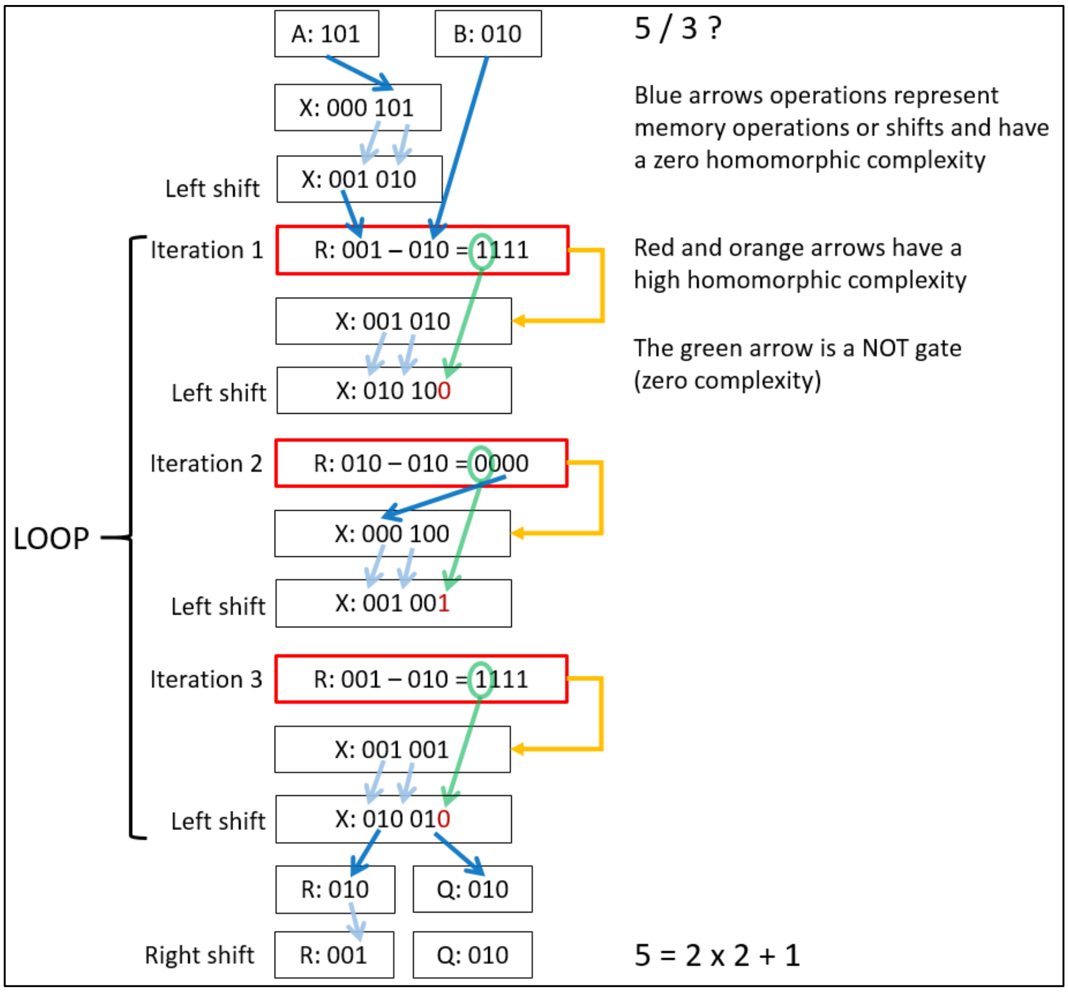


Figure 4.10.a: *Step-by-step example of the division algorithm used*

The red squares are performed with the ripple borrow subtractor, hence with a complexity of $1 + 3n$ multiplications. The orange squares represent the conditional replacements of the upper half of X by R, which is done with the n:1 sequential multiplexer, requiring $3n$ homomorphic multiplications. These two operations are actually the only ones affecting the level parameter. The overall complexity of this circuit is thus $n(1 + 3n + 3n) = 6n^2 + n$ homomorphic multiplications.

The remaining of the operations are memory operations such as binary shifts, copying and erasing ciphertexts and do not affect the performance of the homomorphic computations. The advantage of this circuit is that it accepts binary numbers of any length n, which is great for testing purposes.

The following figure plots the complexity of the multiplication operation together with the one of the Euclidean division operation for different values of n.

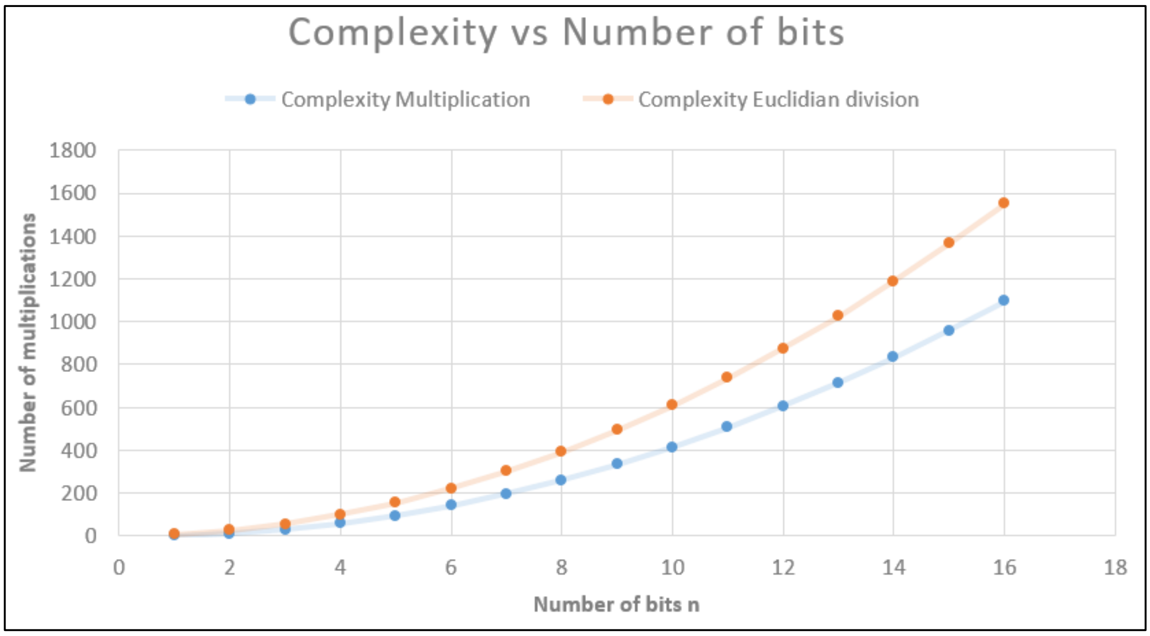


Figure 4.10.b: *Complexities of the multiplication and Euclidean division operations, as a function of the number of bits n*

In a microprocessor, the Euclidean division requires many extra clock cycles in comparison with the multiplication. The situation is similar here, the Euclidean division is much more demanding in term of homomorphic multiplications than the multiplication circuit. The larger the number of bits, the larger the complexity will be and the slower the operation will be unfortunately.

4.11 Designing the average operation

The average operation is simply an addition of N binary numbers of n bits, followed by the Euclidean division circuit. Floating numbers are not supported and should not be because of the already very high complexity. The average would thus be given to the client as a quotient and a remainder. The client could then eventually convert this to a floating point number.

The complexity of this circuit is quite big. First, the addition process takes $(1+3n)+\big(1+3(n+1)\big)+\cdots+\big(1+3(n+N-1)\big)$

$$= \frac{N}{2}\big(1+3n+1+3(n+N-1)\big)$$
$$= \frac{N}{2}(6n+3N-1)$$
$$= 3Nn + 1.5N^2 - 0.5N$$

The Euclidean division process then has a complexity of

$$6(n+N-1)^2 + n + N - 1$$

The addition of these two complexities gives a complexity which grows very quickly as soon as the amount of binary numbers N or their number of bits n raise.

To mitigate this complexity, another design was done. It is called *Fast average* and is limited to N numbers provided such that N is a power of 2. This is due to the fact that the homomorphic Euclidean division circuit is replaced by a simple right shift. This hence allows to keep the complexity to the addition part's complexity. This is obviously rarely accurate but it can be useful for approximations of averages.

Chapter 5

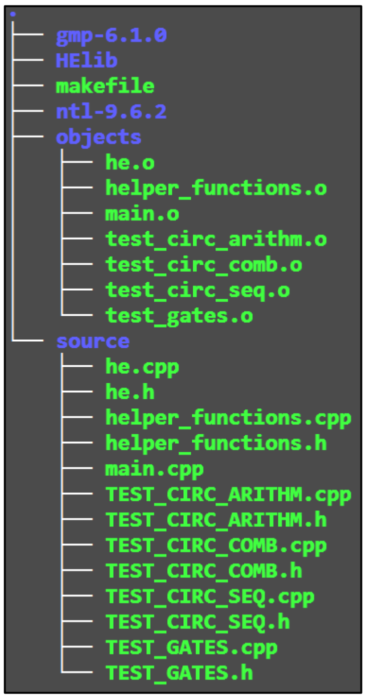
Implementation

5.1 Programming language, libraries and source code

The programming language was limited to C++ for the project, as any other languages would not had brought any added value. The development of the project was mainly about handling binary data and matrices so any programming language would meet the requirements. As HElib is written in C and C++, the best choice was to use C++ for the best compatibility and debugging.

HElib requires the NTL and GMP libraries as explained in chapter 2. It is not complicated but long to install these two and HElib. This is why a single Makefile was implemented to automatically download, compile and link the necessary libraries. This is explained in more detail in Chapter 11.

The source code of the project is organised as in the figure below.



```
.
├── gmp-6.1.0
├── HElib
├── makefile
├── ntl-9.6.2
├── objects
│   ├── he.o
│   ├── helper_functions.o
│   ├── main.o
│   ├── test_circ_arithm.o
│   ├── test_circ_comb.o
│   ├── test_circ_seq.o
│   └── test_gates.o
└── source
    ├── he.cpp
    ├── he.h
    ├── helper_functions.cpp
    ├── helper_functions.h
    ├── main.cpp
    ├── TEST_CIRC_ARITHM.cpp
    ├── TEST_CIRC_ARITHM.h
    ├── TEST_CIRC_COMB.cpp
    ├── TEST_CIRC_COMB.h
    ├── TEST_CIRC_SEQ.cpp
    ├── TEST_CIRC_SEQ.h
    ├── TEST_GATES.cpp
    └── TEST_GATES.h
```

Figure 5.1.a: *Tree representation of the source code structure*

This tree structure representation of the code shows different elements. The folders *gmp-6.1.0*, *ntl-9.6.2* and *HElib* represent respectively the downloaded and compiled GMP library, NTL library and HElib. The subfolders and source files of these directories are not shown are they are not very relevant in the project. The *source* directory contains the C++ source files and headers developed for the project. The corresponding compiled objects are stored in the *objects* directory. This allows a faster compilation of the executable *HEapp.exe* (Windows version).

In order to compile only the necessary modified files in a precise order, the makefile was designed to also include this feature. It uses gcc-g++ or g++ to compile the necessary objects independently, it solves the dependencies required for building HEapp and can automatically compile or re-compile it, hence accelerating the development productivity.

Note that the file *main.cpp* is the code which is run when launching HEapp. Other code can easily be written there.

5.2 The core API: ciphertexts management

The core API developed is implemented in the source file *he.cpp*. It is first a C++ wrapper for HElib. Indeed, HElib proposes very complex and tweakable features which are not always great when building programs on top of it. This is why several functions such as the generation of the key pairs, the encryption or decryption are incorporated in the object class HE implemented in *he.cpp*.

The class HE also contains several new methods to handle ciphertexts. To simplify and make the code more modular, ciphertexts are stored in a dictionary or *unordered map*. Practically, each ciphertext is mapped to a 16-character random and unique alphabetic key. This was implemented for two reasons. First, it greatly simplifies the code and the general productivity of development. Secondly, it was implemented with the storage of ciphertexts in mind, for an eventual server/client communication using this project. The next step would be to serialize and store the ciphertexts with their associated map keys, which is already possible in HElib (further work: refer to *HElib/source/Test_IO.cpp*).

The type for the unordered map key to a ciphertext is defined as **mkt** for *map key type*. An mkt variable is created only when a plaintext vector of binary values is encrypted. The encryption is implemented with the following function.

```
mkt HE::encrypt(vector<long> ptxt_vect) {
/*  Encrypts a plaintext vector of long values. Stores the ciphertext in
    the unordered map. Returns the map key variable associated with
    the new ciphertext */
    Ctxt ctxt(*publicKey, 0);
    ea->encrypt(ctxt, *publicKey, ptxt_vect);
    return storeCtxt(&ctxt);
}
```

Figure 5.2.a: *Encryption function defined in he.cpp*

It uses the public key generated `publicKey` to encrypt the plaintext vector `ptxt_vect` and saves the resulting ciphertext in `ctxt`. The ciphertext is then passed to the `storeCtxt(Ctxt* ctxt)` function which as its name shows stores the ciphertext in the unordered map. Its implementation is shown below as well.

```
mkt HE::storeCtxt(Ctxt* ctxt) {
/*  Takes a pointer to a ciphertext object and stores this one with a
    unique and random key generated in the unordered map map_ctxts.
    The key generated is then returned. */
    mkt k = generate_map_key();
    map_ctxts.insert(make_pair(k, *ctxt));
    return k;
}
```

Figure 5.2.b: *StoreCtxt function defined in he.cpp*

This function is internal to the class or 'private' and is only used to add new ciphertexts or copy ciphertexts. The `generate_map_key()` creates a random 16 characters alphabetic (uppercase) key which is not already present in the unordered map, so which is unique.

Similarly to the encryption function, the decryption function performs the inverse operation with the private key of the homomorphic scheme, called `secretKey` here. The following shows the implementation of the decryption function.

```
vector<long> HE::decrypt(mkt k) {
/*  Decrypts a packed ciphertext and stores the results in a vector of
    long values. If the map key or the ciphertext does not exist,
    the resulting vector is filled with 500 '9's, digit clearly
    distinguishable from 0s and 1s in the binary system. */
    vector<long> ptxt_vect;
    if (ctxt_exists(k)){
        ea->decrypt(map_ctxts.at(k), *secretKey, ptxt_vect);
    } else {
        for (unsigned i = 0; i < 500; i++){
            ptxt_vect.push_back(9); //Could raise an exception instead
        }
    }
    return ptxt_vect;
}
```

Figure 5.2.c: *Decryption function defined in he.cpp*

As explained in the comments, the function stores the decrypted values in a vector of long values. This should contain only 0s and 1s. If the ciphertext or its associated map key can't be found, the resulting vector is filled with 9s which highlights a decryption error in the upper layers of the code.

There are three main other methods to handle ciphertexts: copy, erase and replace. The copy function copies a ciphertext and returns a new map key generated for the copy. Its implementation is in the following code.

```
mkt HE::copy(mkt k){
/*  Copies the ciphertext associated with the map key k, stores the copy
    with a new random and unique map key and returns this key.
    If the map key provided does not exist, an empty key is returned. */
    mkt new_map_key = "";
    if (ctxt_exists(k)){
        Ctxt ctxt = map_ctxts.at(k);
        new_map_key = storeCtxt(&ctxt);
    } else {
        new_map_key = ""; //empty key is an error
    }
    return new_map_key;
}
```

Figure 5.2.d: *Copy function defined in he.cpp*

As ciphertexts can quickly become quite large, it is essential to have the necessary tools for a good memory management. This is why the erase function was implemented, to remove unnecessary ciphertexts and map keys from the unordered map. Its implementation is written below.

```
void HE::erase(mkt k){
/*  Checks if the map key provided and its associated ciphertext exist.
    If they don't, it displays an error message.
    Otherwise, the map key and the ciphertext are erased from the
    unordered map.*/
    if (ctxt_exists(k)){
        map_ctxts.erase(k);
        map_keys.erase(find(map_keys.begin(),map_keys.end(),k));
    } else {
        cout << className() << ": " << __FUNCTION__ << ": Error!" << endl;
    }
}
```

Figure 5.2.e: *Erase function defined in he.cpp*

To conclude this part on ciphertexts management, the replace function was implemented as shown in the following figure. It is used to shift binary numbers for example, as it will be shown in part 5.7.

```
void HE::replace(mkt k, mkt k_replacement){
/*  Checks if both map keys and ciphertexts exist before doing anything.
    It then replaces the ciphertext at map key k by a copy of the
    ciphertext at map key k_replacement. If one of the map keys or
    ciphertexts do not exist, an error message is displayed.*/
    if (ctxt_exists(k) && ctxt_exists(k_replacement)){
        Ctxt ctxt = map_ctxts.at(k_replacement);
        map_ctxts.at(k) = ctxt;
    } else {
        cout << className() << ": " << __FUNCTION__ << ": Error!" << endl;
    }
}
```

Figure 5.2.f: *Replace function defined in he.cpp*

5.3 The core API: Client keys generation

The library HElib proposes a public key infrastructure (PKI) to encrypt and decrypt homomorphic ciphertexts, which is often more desirable than a symmetrical encryption scheme. For this project, a key pair hence contains a private key and its associated public key. This key pair is generated to encrypt and decrypt the homomorphic ciphertexts.

First of all, several parameters have to be set to generate the key pair. One of these is the level parameter for example. The following C type structure was implemented to contain all the necessary parameters.

```
typedef struct key_params{
    long m;
    long p; // Defines the native plaintext space. Computations are 'mod p'
    long r; // Defines the native plaintext space
    long d; // Defines the plaintext space F(p^d) for packing ciphertexts
    long k; // Security parameter (bits)
    long L; // Level parameter or Circuit depth
    long c; // Columns in key switching matrix (usually 2 or 3)
    long w; // Hamming weight of a secret key, 64 recommended
    long slb; // Slots Lower bound - to force ciphertexts to pack more slots.
} key_params;
```

Figure 5.3.a: *Key generation parameters structure, defined in helper_functions.h*

The first three parameter, `m`, `p` and `r` define the native plaintext space as $\mathbb{Z}[X]/(\phi_m(X), p^r)$. In our case, we always set `p=2` and `r=1` to support binary logic. `m` is found using the `FindM` function from HElib, which uses an optimised look-up table explained later.

The `d` parameter defines the plaintext space of slots for packed ciphertexts as F_{p^d}. It is set to 0 for this project. The security parameter `k` is the security level in bits of the private key to be generated, and is set to 256 here. The number of columns in the key switching matrix (used for SIMD purposes) is defined by `c` and is recommended to be set to 2 or 3. Setting it to 3 showed a speed gain between 5% and 60% for complex operations such as the addition operation, so c is set to 3. Similarly, the hamming weight of the private key is set to the recommended value of 64.

The two remaining parameters are the most variable and important in terms of performance: the level parameter `L` and the lower bound for the available plaintext slots `slb`. As it has now been explained many times, the level parameter should be the lowest possible. It is usually calculated as a function obtained from linear regression calculations or with a look-up table. The lower bound for the plaintext slots is not always used but can be quite useful in some cases. Indeed, as explained in chapter 4, this one can be raised above the strict automatic minimum number of slots in order to shorten the time per operation at the expense of the overall time for all the operations to complete. As an analogy, this parameter slb acts similarly to a microprocessor pipeline.

As a summary the following code shows how these parameters are set by default and specifically for an example function.

```
//Common configuration
key_params params;
params.p = 2;
params.r = 1;
params.d = 0;
params.k = 256;
params.c = 3;
params.w = 64;

//Specific parameters
params.L = 2.076*n + 1.672; //Linear regression
params.slb = 400; //minimum of 400 plaintext slots
params.m = FindM(params.k,params.L,params.c,params.p,params.d,params.slb,0);
```

Figure 5.3.b: *Key parameters initialization procedure example*

Once these parameters are set, they are passed to the generation function `keyGen(key_params params)` which then creates the key pair as well as several other elements necessary to the encryption and decryption of the ciphertexts. The function is long and complex and is thus only included in the source code, in *source/he.cpp*. When done, the function returns the number of available plaintext slots so that the client knows how much data can be inserted in each ciphertext.

5.4 The core API: Adapting HElib arithmetic operations

As the project uses an unordered map with map keys pointing to ciphertexts, all the arithmetic operations provided by HElib were adapted to the unordered map system. The following code was used to do so.

```
void HE::add(mkt k1, mkt k2){
    map_ctxts.at(k1) += map_ctxts.at(k2);
}
void HE::sub(mkt k1, mkt k2){
    map_ctxts.at(k1) -= map_ctxts.at(k2);
}
void HE::mul(mkt k1, mkt k2){
    map_ctxts.at(k1).multiplyBy(map_ctxts.at(k2));
}
void HE::mul(mkt k1, mkt k2, mkt k3){
    map_ctxts.at(k1).multiplyBy2(map_ctxts.at(k2), map_ctxts.at(k3));
}
void HE::neg(mkt k1){
    map_ctxts.at(k1).negate();
}
bool HE::eq(mkt k1, mkt k2){
    bool comparePubkeys = true;
    return map_ctxts.at(k1).equalsTo(map_ctxts.at(k2), comparePubkeys);
}
```

Figure 5.4.a: *Wrapper to HElib's arithmetic operators*

An important trick to understand for the following is that all the operations except *eq* overwrites the first ciphertext by the result of the operation. This means that for example *add("AJGDSLVMSDSGSOWH"," FHKSTYICHHDXGMBL")* will overwrite the ciphertext associated with the key "*AJGDSLVMSDSGSOWH*" by the addition resulting ciphertext. This is also where the *copy* function becomes handy.

As it has been explained in the analysis and design chapter, only the *add* and *mul* operators were required to implement homomorphic logic gates. All the others are not useful in the binary domain.

For an eventual future work based on the present project, note that the *multiplyBy* operation provided by HElib was chosen instead of the *= shortcut as it performs a relinearization on the ciphertext after the multiplication, and, from a practical point of view, greatly reduces the memory usage of the program.

5.5 Homomorphic logic gates

This part as well as the many following parts regarding circuits implementations should be used by the cloud computer or server in a client-server communication scheme.

As it has been extensively described in the analysis and design chapter, logic gates were implemented using the *add* and *mul* operators implemented in 5.4. In the source code, each gate definition contains comments regarding its purpose, its inputs, its outputs and its homomorphic complexity. The following shows how the AND logic gate was implemented as an example.

```
void HE::AND(mkt k1, mkt k2){
    /*  Purpose: Binary AND of k1 and k2
        Inputs: Bits of k1 and bits of k2
        Outputs: k1 = k1 AND k2
        Complexity: 1 multiplication */
    mul(k1, k2);
}
```

Figure 5.5.a: *Implementation of the homomorphic AND gate*

As it has been explained in the analysis and design chapter, a ciphertext filled of 1s has to be provided to the cloud computer. In order to do so, the pattern illustrated below has to be followed.

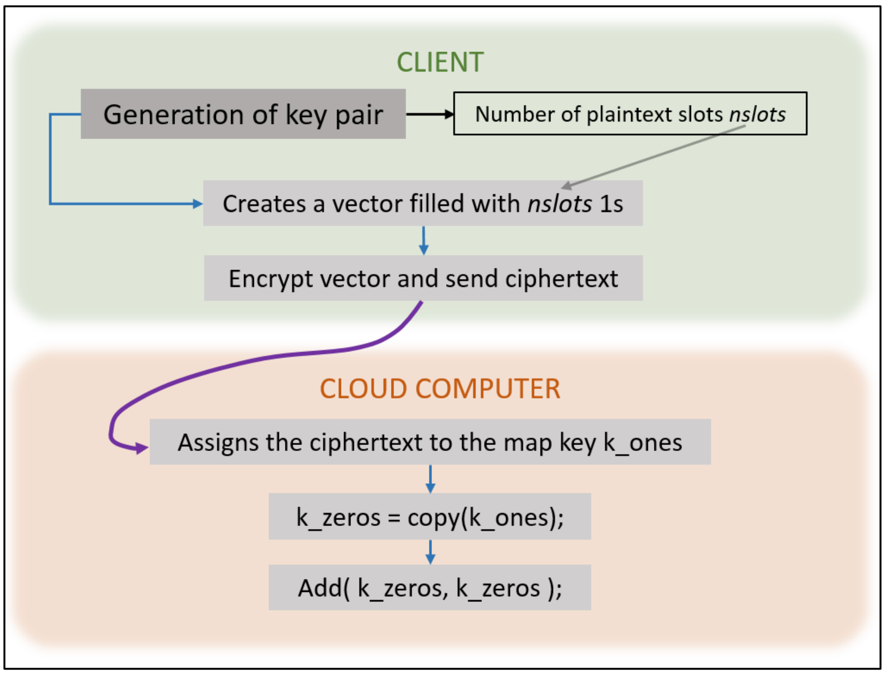

CLIENT
Generation of key pair
Number of plaintext slots *nslots*
Creates a vector filled with *nslots* 1s
Encrypt vector and send ciphertext
CLOUD COMPUTER
Assigns the ciphertext to the map key k_ones
k_zeros = copy(k_ones);
Add(k_zeros, k_zeros);

Figure 5.5.b: *Pattern to provide the Ones ciphertext to the cloud computer*

The client has to simply generate a vector filled with 1s, encrypt it and send the ciphertext to the cloud computer. This is implemented in the function `mkt HE::setOnes(long n)` defined in he.cpp. The cloud computer would then receive the ciphertext and assign it to a map key named *k_ones* so that I can be used as a permanent reference to 1 for the client. It also deduces *k_zeros* by adding *k_ones* to *k_ones*. Again, this is implemented in the method *set01* defined in *he.cpp*.

This is not very dangerous in terms of security as the cloud computer can't deduce anything from the received ciphertext except if all the SIMD results are equal to 1s or are equal to 0s across all the plaintext slots. This is very unlikely to happen and because it is the only solution found to implement a NOT gate and to go forward, it will still be used. This should also answer the security question raised about the NOT gate in chapter 4. The NOT gate is hence implemented with the following code.

```
void HE::NOT(mkt k1){
    /*  Purpose: Binary NOT of k1
        Inputs: Bits of k1
        Outputs: k1 = NOT k1
        Complexity: negligible */
    add(k1,k_ones);
}
```

Figure 5.5.c: *Implementation of the homomorphic NOT gate*

Finally, as a more complex example, the implementation of the OR gate is shown below.

```
void HE::OR(mkt k1, mkt k2){
    /*  Purpose: Binary OR of k1 and k2
        Inputs: Bits of k1 and bits of k2
        Outputs: k1 = k1 OR k2
        Complexity: 3 multiplication */
    mkt k2c = copy(k2);
    NAND(k1,k1);
    NAND(k2c,k2c);
    NAND(k1,k2c);
    erase(k2c);
}
```

Figure 5.5.d: *Implementation of the homomorphic OR gate*

As a short explanation, the OR gate requires 3 NAND gates. As an AND gate needs one homomorphic multiplication, the OR gate requires 3 of them and thus has a complexity of 3. The *copy* and *erase* functions are called to make a temporary local copy necessary for this gate to execute.

The remaining logic gates are implemented similarly in *he.cpp*. The next part covers combination circuits built on top of the logic gates previously implemented.

5.6 Homomorphic combinational circuits

The combinational circuits designed in chapter 4 are now implemented using the homomorphic logic gates previously implemented. Without going into the many details of the source code, these circuits take between 2 and 4 input bits and output between 1 and 2 bits. More details about their complexity can be found in the chapter 4. Their implementation is very similar to the implementation of the logic gates. As an example, the implementation of the half adder binary circuit is shown below.

```
void HE::HFADDER(mkt k1, mkt k2){
    /*  Purpose: Binary half addition of k1 and k2 (with carry out only)
        Inputs: Bit k1 and Bit k2
        Outputs: k1 = SUM and k2 = CARRY_OUT
        Complexity: 1 multiplication */
    mkt k_Cout = copy(k1);
    XOR(k1,k2); //k1 is SUM
    AND(k2,k_Cout); //k2 is CARRY OUT
    erase(k_Cout);
}
```

Figure 5.6.a: *Implementation of the Half Adder combinational circuit*

The following combinational circuits were hence implemented similarly:

- Half adder
- Full adder
- Half subtractor
- Full subtractor
- Half equal circuit
- Full equal circuit
- Smaller Than circuit
- Half comparator
- Full comparator
- 2:1 multiplexer

As for the logic gate, these circuits process in parallel the plaintext slots of the ciphertext, and overwrite the first and eventually second input ciphertexts with the output ciphertexts.

Please refer to the source code *he.cpp* from line 300 to line 402 for detailed explanations of each combinational circuit implemented.

5.7 Homomorphic sequential circuits

The sequential circuits are implemented on top of the combinational circuits. They are focused on performing operations on n bit numbers, instead of single bits.

In a standard computer architecture, these require a clock and eventually memory operations in some cases. For this project, the clock is replaced by iterative loops such as a C++ for loop, and memory operations do not require any clock cycles or homomorphic operations as they are simply ciphertexts handling operations such as *copy* or *replace*. In comparison with a microprocessor architecture, this is great as the memory operations are not time consuming in comparison with the homomorphic operations.

As it is explained in part 4.7, an n bit binary number is represented by n ciphertexts in a certain order. It is essential to understand this to grasp how the sequential circuits work. In the source code, a binary number is passed as a vector of mkt (map key type).

The function *PAD_BITS* defined in *he.cpp* allow to feed binary numbers of different size into the sequential circuits. It is called in all the 2-inputs sequential circuits at the beginning and basically pads the binary number with the smallest number of bits with leading zeros using the ciphertext associated with the map key *k_zeros*. The function is implemented with the following code.

```cpp
unsigned HE::PAD_BITS(vector<mkt> &k1_bit, vector<mkt> &k2_bit){
    unsigned nbits_1 = k1_bit.size();
    unsigned nbits_2 = k2_bit.size();
    if (nbits_1 < nbits_2){
        for (unsigned i = nbits_1; i < nbits_2; i++){
            k1_bit.push_back(copy(k_zeros));
        }
        return nbits_2;
    } else if (nbits_1 > nbits_2){ //just resize k2 with leading zeros
        for (unsigned i = nbits_2; i < nbits_1; i++){
            k2_bit.push_back(copy(k_zeros));
        }
        return nbits_1;
    }
}
```

Figure 5.7.a: *Implementation of the PAD_BITS function*

The return value of this function is then used as the maximum number of iterations to be performed in the iterative for loops for example.

The ripple carry adder and ripple borrow subtractor follow the same pattern of implementation: they use once their respective half combination circuit and then n − 1 times their full combinational circuit. The ripple borrow subtractor code will thus be the only code shown here. As usual, deeper information on the sequential circuits can be found in the comments in *he.cpp*.

```cpp
void HE::RBSUBER(vector<mkt> &k1_bit, vector<mkt> k2_bit){
    /*  Purpose: Ripple Borrow Subtractor nbits x nslots
        Inputs: Bit k1[0],..., bit k1[nbits - 1] and
                bit k2[0],..., bit k2[nbits - 1]
                Note that nbits can be different for k1 and k2 but these
                would then be padded if it was the case
        Outputs:    k1 = (nbits + 1) bits DIFFERENCE (unsigned and signed)
                    and k2 = untouched
        Complexity: 1 + nbits*5 multiplications */
    unsigned nbits = PAD_BITS(k1_bit, k2_bit);
    vector<mkt> kc_bit(nbits);
    for(unsigned i = 0; i < nbits; i++){
        kc_bit[i] = copy(k2_bit[i]);
    }
    k1_bit.push_back(copy(k_zeros)); //for sign
    kc_bit.push_back(copy(k_zeros)); //for sign
    nbits++;

    HFSUBER(k1_bit[0], kc_bit[0]); //1 multiplication
    for (unsigned i = 1; i < nbits; i++){ //this is nbits+1 actually
        FLSUBER(k1_bit[i], kc_bit[i], kc_bit[i-1]); //5 multiplications
        erase(kc_bit[i-1]);
    }
    erase(kc_bit[nbits - 1]);
}
```

Figure 5.7.b: *Implementation of the ripple borrow subtractor for n bit numbers*

First the function *PAD_BITS* is executed. A full copy of the binary number `k2_bit` is then performed and stored in `kc_bit` in order not to modify the binary number `k2_bit` as it is not necessary. Only `k1_bit` will be overwritten by the final result.

For the ripple borrow subtractor, an extra bit is added to both numbers to support the sign bit. For the ripple carry adder, the extra bit is also added to support the eventual overflow. The size of the vector `k1_bit` is thus extended by one.

The subtraction then begins with the half subtractor circuit followed by an n-1 for loop executing the full subtractor circuit with the correct bits or ciphertexts here. The *erase* function removes the unnecessary ciphertexts from the unordered map.
It is also important to note that a ripple borrow subtractor could be implemented with full adders, but this would require the initial carry in to be forced to 1. This would be a waste of homomorphic multiplications as well as an added security risk. This is why the half and full combinational subtractors were previously implemented and used in this sequential circuit.

Regarding the ripple comparator circuit, its implementation is similar and is shown below.

```
void HE::RCMP(vector<mkt> &k1_bit, vector<mkt> &k2_bit){
    /*  Purpose: Ripple Comparator
                 (equality and lower than, for unsigned numbers)
        Inputs: Bit k1[0],..., bit k1[nbits - 1] and
                bit k2[0],..., bit k2[nbits - 1]
                Note that nbits can be different for k1 and k2 but these
                would then be padded if it was the case
        Outputs: k1[0] = (k1==k2) and k2[0] = (k1>k2)
                 The rest of k1 and k2 is erased
        Complexity: 1 + (nbits - 1)*6 multiplications */
    unsigned nbits = PAD_BITS(k1_bit, k2_bit);

    //starts with MSB, 1 multiplication
    HFCMP(k1_bit[nbits - 1], k2_bit[nbits - 1]);
    for (int i = nbits - 2; i >= 0; i--){
        //Full comparator requires 6 multiplications
        FLCMP(k1_bit[i], k2_bit[i], k1_bit[i+1], k2_bit[i+1]);
        erase(k1_bit[i+1]);
        k1_bit.pop_back();
        erase(k2_bit[i+1]);
        k2_bit.pop_back();
    }
}
```

Figure 5.7.c: *Implementation of the Ripple comparator*

It uses the *PAD_BITS* function and its half and full combination circuits the same way as the ripple borrow subtractor. However, the result is different. This circuit outputs two single bits, where one is HIGH if the two binary numbers are equal, and the second one is HIGH if the first number is strictly greater than the other number. The equality bit and the greater than bit actually replace the bit representing the least significant bit (LSB) for the first and second binary numbers, respectively. In addition, all the ciphertexts other than the "LSB ciphertext" are erased, for both resulting binary numbers. The final result are thus only two ciphertexts, where the first one has the equality bits and the second one has the greater than bits.

Concerning the ripple equality tester, it is implemented the same way as the ripple comparator but without the greater than feature. It outputs only one ciphertext containing the final resulting equality bit. This circuit was developed because of its low complexity of $n - 1$, whilst the complexity of the ripple comparator is $3n$.

Another very useful sequential circuit is the N:1 multiplexer. As stated in part 4.7, it allows to perform a selection of a binary number over another binary number. This is precisely essential for the Euclidean division implementation, as it will be shown in part 5.8.2. The N:1 multiplexer is called *NMUX* in the source code and simply iterates over all the bits with the same selector bit (ciphertext). The output will hence be the first binary number if the selector bit is 1 and the second otherwise.

The two last sequential circuits implemented are the left and right binary shifter. These are not actually performing any homomorphic operations and only moves ciphertexts around to shit a binary number by a plaintext constant. These operators are thus useful if used by other operators such as the multiplication.

The right shift operation uses the *replace* function across all the bits of a binary number to shift the bits to the right, without knowing their actual content. It appends a new zeros ciphertext as the new MSB to keep the same number of bits for the output.

The left shift does the opposite but will however extend the number of bits of the binary number by the amount of shift. The following shows the code of its implementation, in order to have a clear idea.

```
void HE::SHIFTL(vector<mkt> &k1_bit, const unsigned shift){
    /*  Purpose: Left shift of binary number and sets LSB to 0
        Inputs: Bit k1[0],..., bit k1[nbits - 1] and the shift amount
        Outputs: None, just replaces ciphertexts & extends the size of k1_bit
        Complexity: negligible */
    for (unsigned i = 0; i < shift; i++){
        k1_bit.push_back(copy(k_zeros));
    }
    unsigned nbits = k1_bit.size();
    for (unsigned j = 0; j < shift; j++){
        for (unsigned i = nbits - 1; i > 0; i--){ //MSB to LSB
            replace(k1_bit[i], k1_bit[i-1]);
        }
        k1_bit[0] = copy(k_zeros);
    }
}
```

Figure 5.7.d: *Implementation of the left shift binary operator*

All these sequential circuits allow to add, subtract, compare and shift binary numbers of any size and are a first great step towards usefulness. The next stage is to use all these sequential and combinational circuits to build even more complex circuits such as the multiplication operation and the Euclidean division operation.

5.8 Homomorphic arithmetic circuits

There are two arithmetic circuits implementing the multiplication and the Euclidean division respectively.

5.8.1 The multiplication arithmetic circuit

The multiplication implementation takes two n bit binary numbers (or pad them with zeros so they match the same size) and overwrites/extends the n ciphertexts of the first binary number by the $2n$ ciphertexts of the multiplication result. Indeed, as the cloud computer does not know the content of the numbers, the result has to be $2n$ bit to cover all the possible cases and its size can't be reduced. The following figure sums up the size of the inputs and outputs.

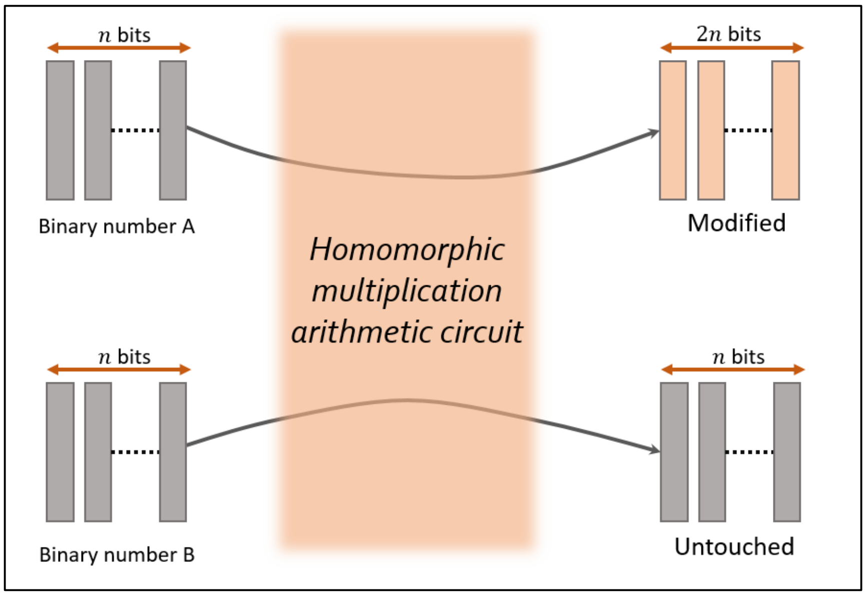


Figure 5.8.1.a: *Multiplication operation – Inputs and Outputs*

As explained in part 4.9, it implements the long multiplication algorithm so it will use the ripple carry adder circuit, some AND logic gates and the left shift.

Let n be the number of bits of the longest input binary number. The program starts by creating n copies of the first binary number A represented by *k1_bit*. This does not require any homomorphic operations and is thus time efficient.

```
//Setting binary rows to k1
vector< vector<mkt> > k(nbits, vector<mkt>(nbits));
for (unsigned row = 0; row < nbits; row++){
    for (unsigned bit = 0; bit < nbits; bit++){
        k[row][bit] = copy(k1_bit[bit]);
    }
}
```

Figure 5.8.1.a: *Multiplication operation – Creating n copies of k1_bit*

It then performs the *AND and left shift* of these copied rows with the second binary number B, and overwrites the rows with their respective result.

```
//ANDing and shifting binary rows
//Requires only one multiplication for ciphertext in term of complexity
//In terms of time, it needs nbits * nbits multiplications
for (unsigned row = 0; row < nbits; row++){
    for (unsigned bit = 0; bit < nbits; bit++){
        AND(k[row][bit], k2_bit[row]); //1 multiplication
    }
    SHIFTL(k[row], row); //0 multiplication
}
```

Figure 5.8.1.b: *Multiplication operation – AND and left shift of copied 'rows'*

Once this is done, the program has to add these rows together with the ripple carry adder circuit. To save an eventual ripple carry addition operation, the accumulator is set to the first row with the following code. Because the output is stored in the first binary number input *k1_bit*, the accumulator is actually *k1_bit* now.

```
//Setting accumulator to the first binary row
for(unsigned bit = 0; bit < nbits; bit++){
    erase(k1_bit[bit]);
    k1_bit[bit] = copy(k[0][bit]);
}
k1_bit.push_back(copy(k_zeros)); //Adds a 0 as the new MSB
```

Figure 5.8.1.c: *Multiplication operation – Setting accumulator*

The ripple carry addition can then proceed to accumulate all the remaining $n - 1$ rows in *k1_bit*. The following code is executed.

```
//Accumulates all the rows in result
//Complexity & Time: 4.5*nbits*nbits - 3.5*nbits - 1
for (unsigned row = 1; row < nbits; row++){
    RCADDER(k1_bit, k[row]); //1 + 3n, and increases n by 1
}
```

Figure 5.8.1.d: *Multiplication operation – Accumulating the computed rows*

This is the most time consuming operation as shown by its homomorphic complexity. The code finally erases all the unnecessary ciphertexts and map keys such as the ones in the vector *k*.

5.8.2 The Euclidean division arithmetic circuit

The Euclidean division implementation takes two n bit binary numbers and overwrites the n ciphertexts of the first and second binary number respectively by the n ciphertexts representing the quotient and by the n ciphertexts representing the remainder of the Euclidean division operation. The following block diagram summarizes this.

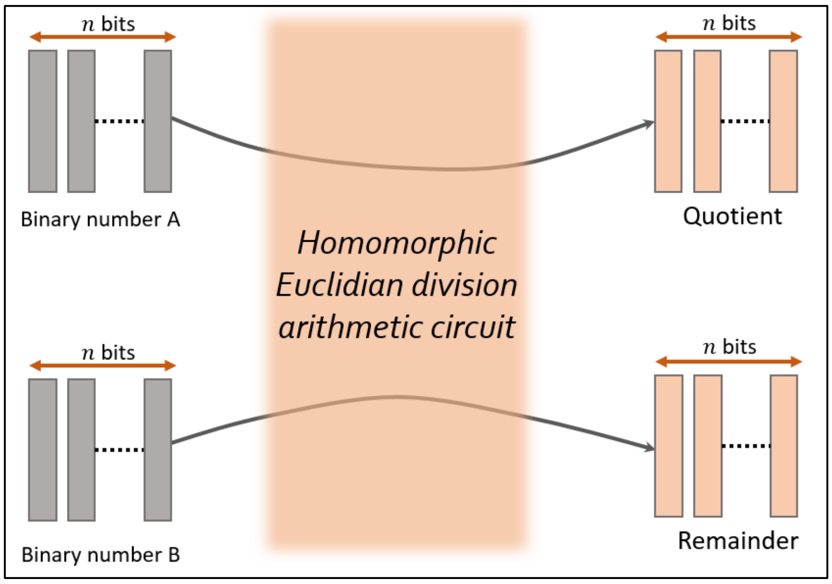


Figure 5.8.2.a: *Euclidean division – Inputs and outputs*

The source code implements the restoring long division algorithm stated in part 4.10. It is very long and contains many memory operations (*copy*, *erase* etc.) so it will not be shown fully here. Detailed explanations can be found in the comments of the code in *he.cpp*.

The if – else condition statement described in the design of the Euclidean division is implemented with the *NMUX* N:1 multiplexer sequential circuit previously implemented. More precisely, the two possible values are pre-computed or copied, and the multiplexer will assign the right one depending on a single bit (sign bit of subtraction).

The rest of the code is memory operations, binary shifts, and restore-test homomorphic subtractions. Overall, this circuit requires the ripple borrow subtractor circuit, the N:1 multiplexer, the right and left shifts, a NOT gate and all the ciphertexts handling functions.

5.9 Homomorphic average operation

For once, the implementation is quite simple. It is only a for loop of ripple carry adders followed by essentially the Euclidean division circuit. The following code implements the average circuit.

```
void HE::AVERAGES(vector< vector<mkt> > &numbers, vector<mkt> &N){
    /*numbers is a vector of numbers, where each number is a vector of
    encrypted bits. N is the divider which has to be provided to perform
    the homomorphic euclidian division. The quotient and remainder of
    the average are returned in numbers[0] and N respectively. */
    for (unsigned i = 1; i < numbers.size(); i++){
        RCADDER(numbers[0],numbers[i]); //accumulates all the numbers
        for(unsigned b = 0; b < numbers[i].size(); b++){
            erase(numbers[i][b]);
        }
    }
    PAD_BITS(numbers[0],N);
    DIVIDE(numbers[0], N);
}
```

Figure 5.9.a: *Source code for the Average circuit*

The ripple carry adder (*RCADDER*) allows to accumulate the numbers provided. The resulting sum is then divided by the divisor with the *DIVIDE* circuit (Euclidean division).

Now, because this circuit is actually very limited because of its high complexity, the fast average circuit described in chapter 4 was also implemented. Its code is longer as it contains checks regarding the number N of binary numbers provided.

```
void HE::FAVERAGES(vector< vector<mkt> > &numbers){
    /*numbers is a vector of numbers, where each number is a vector of
    encrypted bits. This operation will loose the precision as no
    remainder is calculated */
    unsigned N = numbers.size();
    if (N == 0){
        cout << className() <<": "<< __FUNCTION__ <<": N can't be 0"<< endl;
        return;
    }
    float rs = log2(N); //amount of right shift
    bool N_is_pow_of_2 = false;
    for (unsigned i = 0; i < N; i++){
        if(rs == i){
            N_is_pow_of_2 = true; //or N is 1 which is fine too
            break;
        }
    }
    if (!N_is_pow_of_2){
        cout << className() <<": "<< __FUNCTION__ <<": N not power of 2."<<
endl;
        return;
    }

    for (unsigned i = 1; i < N; i++){
        RCADDER(numbers[0],numbers[i]); //accumulates all the numbers
        for(unsigned b = 0; b < numbers[i].size(); b++){
            erase(numbers[i][b]);
        }
    }
    for (unsigned i = 0; i < N - 1; i++){
        numbers.pop_back();
    }
    SHIFTR(numbers[0], (unsigned)rs); //divides by a power of 2
}
```

Figure 5.9.a: *Source code for the Fast Average circuit*

As the average circuit, this fast average accumulates the numbers. It then shifts right the numbers by $rs = log_2(N)$. As it will be shown in the next chapters, this implementation is not accurate but is way faster and lighter than the average circuit.

5.10 Overall chart: complexity and circuit dependencies

The following shows a sum up diagram showing the dependencies and complexities of all the circuits implemented so far.

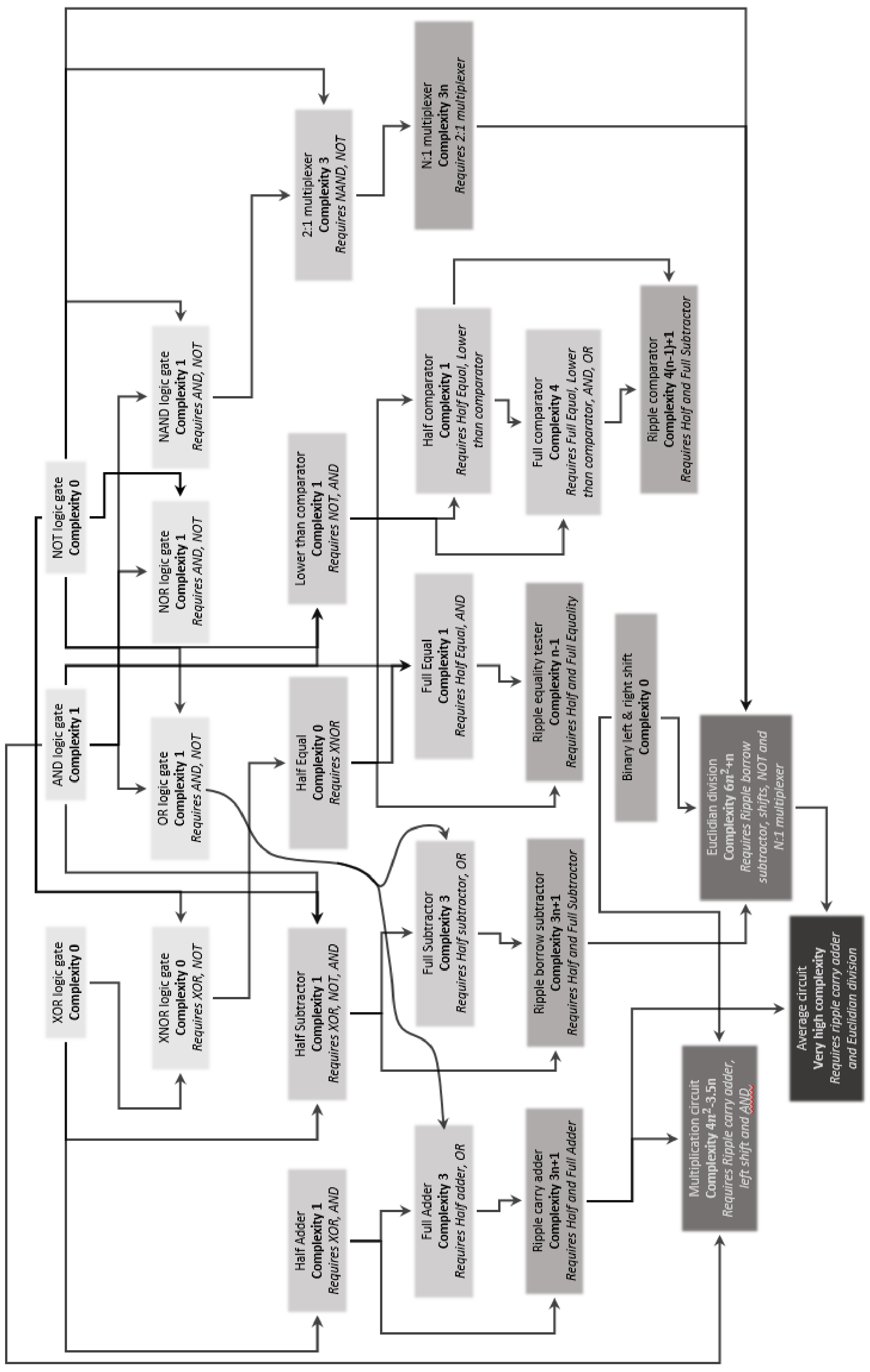


Figure 5.10.a: *Chart of the dependencies and complexities of all the circuits*

Chapter 6

Testing

In order to verify the working and optimise the program developed, several methods were implemented. This chapter is structured as follows. The first part concerns unit tests which has been developed to verify the working of each homomorphic gate and circuit. The second part exposes the time testing methods used on sequential and arithmetic circuits. The last part show how to test the code implemented and push it to its limit.

6.1 Unit testing

In this project, unit testing consists in testing individually and independently each logic gate and circuit implemented, called units in this context, for proper operation. Unit testing has been coded to be automated and is separated in the following files.

- TEST_GATES (.cpp and .h) which tests logic gates
- TEST_CIRC_COMB (.cpp and .h) which tests combinational circuits
- TEST_CIRC_SEQ (.cpp and .h) which tests sequential circuits
- TEST_CIRC_ARITHM (.cpp and .h) which tests arithmetic circuits.

Each of these generate random bits as inputs and verifies that the output of the gate or circuit corresponds to the C++ plaintext operation on the inputs. To do so efficiently, the *Errors* and the *Conversion* objects were designed and implemented in the *helper_functions.cpp* file.

6.1.1 The *Errors* object

The *Errors* object is simple and allow to gather results from several unit tests. It allows for example that one test fails but ten others pass, without stopping the program. It is declared in *helper_functions.h* as:

```
class Errors{
    public:
        Errors(string t);
        void add(string name, bool error);
        void display();
    private:
        string title;
        vector<string> names;
        vector<bool> errors;
};
```

Figure 6.1.1.a: *Declaration of the Errors object*

Its definition is appended below.

```
Errors::Errors(string t){
    title = t;
}
void Errors::add(string name, bool error){
    names.push_back(name);
    errors.push_back(error);
}
void Errors::display(){
    bool no_error = true;
    for (int i = 0; i < errors.size(); i++){
        if (errors[i]){
            no_error = false;
            cout << title << ": Error occured for test " << names[i] << endl;
        }
    }
    if (no_error){
        cout << title << ": ALL TESTS PASSED" << endl;
    }
}
```

Figure 6.1.1.b: *Definition of the Errors object*

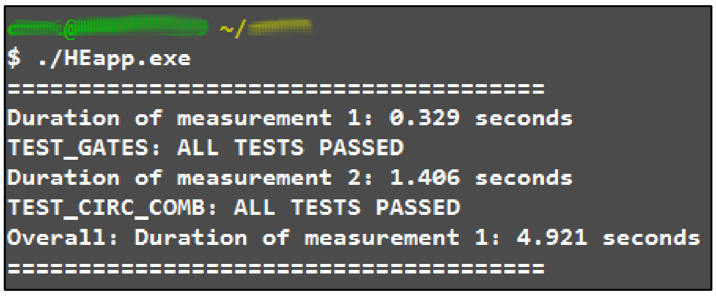
It is used as shown in the code snippet below, coming from the combinational circuit unit test file.

```
Errors TEST_CIRC_COMB::test(){
    Errors e("TEST_CIRC_COMB");
    e.add("HALF ADDER combinational circuit", test_HFADDER());
    e.add("FULL ADDER 1 bit circuit", test_FLADDER());
    e.add("HALF SUBTRACTOR combinational circuit", test_HFSUBER());
    e.add("FULL SUBTRACTOR combinational circuit", test_FLSUBER());
    e.add("HFEQUAL combinational circuit", test_HFEQUAL());
    e.add("FLEQUAL combinational circuit", test_FLEQUAL());
    e.add("SMALLER combinational circuit", test_SMALLER());
    e.add("HFCMP combinational circuit", test_HFCMP());
    e.add("FLCMP combinational circuit", test_FLCMP());
    e.add("MUX combinational circuit", test_MUX());
    return e;
}
```

Figure 6.1.1.b: *Use of the Errors object*

The title of the *Errors* object is set in the first line of this function as *TEST_CIRC_COMB*. Each function starting with *test_* represent one unit test for a specific circuit. Each of these functions return false if there is no error, and true if an error occurred. These are added to the *Errors* object so that it will detect which of the unit tests failed, as shown in its definition. At a higher layer calling the series of unit tests TEST_CIRC_COMB, the *Errors* object is received and its method *display()* has to be called to show if all test passed or to display the eventual errors.

The *Errors* object is particularly very useful for a collection of unit tests relatively quick to run such as the test of all the homomorphic logic gates or combinational circuits. The user can easily see if his changes broke a part of the code in a relatively short time. On the other hand, the object may be less relevant for a set of very long operations such as the Euclidean division circuit test with the multiplication circuit unit test. An example of the end-user unit testing display is shown below for illustration.



```
[redacted] ~/[redacted]
$ ./HEapp.exe
======================================
Duration of measurement 1: 0.329 seconds
TEST_GATES: ALL TESTS PASSED
Duration of measurement 2: 1.406 seconds
TEST_CIRC_COMB: ALL TESTS PASSED
Overall: Duration of measurement 1: 4.921 seconds
======================================
```

Figure 6.1.1.c: *Display produced by the Errors object*

On this screenshot, only the TEST_GATES and TEST_CIRC_COMB lines are displayed by the *display()* method of *Errors*.

6.1.2 The *Conversion* object

The *Conversion* object is also defined in *helper_functions.cpp*. It is a set of methods to convert various types of data between each other. Its methods and their purpose are listed in the following:

- str2Bool: Converts a single character of bit to a Boolean
- bool2Str: Converts a Boolean to its corresponding character '1' or '0'.
- long2Str: Converts a long integer to a string of its value.
- long2bitStr: Converts a positive long integer to a string of bits.
- bitStr2Long: Converts a string of bits to a long integer.
- bitStr2LongStr: Converts a string of bits to a long integer in a string.
- signedBitStr2Long: Converts a string of signed bits to a long integer.
- signedBitStr2LongStr: Converts a string of signed bits to a long integer in a string
- matrix2bitStrVec: Converts a matrix of bits, such as the ones obtained from the decryption of n ciphertexts, to a vector of binary strings.
- matrix2LongVec: Converts a matrix of bits to a vector of integers (long).
- matrix2SignedLongVec: Converts a matrix of bits to a vector of signed integers (long). This is only used for the ripple borrow subtractor unit test actually as its results are signed.
- longVec2Matrix: Converts a vector if integers to a matrix of bits

As it will be shown in the following, all these functions are very useful to easily handle bits, binary numbers and ciphertexts all together. For example, converting a vector of integers to a matrix of bits which can then be directly encrypted is easy with the method *longVec2Matrix*.

6.1.3 Logic gates unit tests

For the collection of logic gates unit tests, the inputs are set as all the 2 bit binary combinations possible, with the following code.

```
    v_in.resize(IO, vector<long> (nslots,0));
    v_in[0][0] = 0; v_in[1][0] = 0;
    v_in[0][1] = 0; v_in[1][1] = 1;
    v_in[0][2] = 1; v_in[1][2] = 0;
    v_in[0][3] = 1; v_in[1][3] = 1;
    for (int i = 0; i < IO; i++){
        k_constant[i] = he.encrypt(v_in[i]);
    }
```

Figure 6.1.3.a: *Logic gates unit tests inputs definition*

The following will not describe the exhaustive list of all the unit tests for the logic gates, although you are welcome to have a look in *TEST_GATES.cpp*. For instance, the code for the test of the homomorphic OR logic gate is shown here.

```
bool TEST_GATES::test_OR(){
    make_copies(); //k2 = k0
    he.OR(k[0],k[1]);
    for (int i = 0; i < IO; i++){
        v_out[i] = he.decrypt(k[i]);
    }
    for (int i = 0; i < nslots; i++){
        if (v_out[0][i] != (v_in[0][i] | v_in[1][i])){
            return true; //error
        }
    }
    return false;
}
```

Figure 6.1.3.b: *Unit test of the OR logic gate*

In this code, *make_copies* simply copies the ciphertexts in order to keep the original ciphertexts intact for the next unit tests. The object *he* represents the API developed in *he.cpp*. k[0] and k[1] are the map keys linking to the two ciphertexts respectively containing the encrypted bits of [0 0 1 1 0 0 … 0] and [0 1 0 1 0 0 … 0]. The resulting ciphertexts are then decrypted, and the resulting plaintext vector of binary values is checked against the C++ plaintext operation of the OR operation. It for any plaintext slot this is not valid, the test has failed and the function returns *true* to the *Errors* object.

All the remaining logic gates unit tests are designed in the exact same way. They also all share a common key pair generated at the beginning of the set of unit tests, with a level parameter of 3, which is the strict minimum for the logic gate requiring one homomorphic multiplication to operate correctly.

6.1.4 Combinational circuits unit tests

The collection of combinational circuits unit tests is similar to the logic gates one. Indeed, the combinational circuits take between 2 to 3 input bits and outputs between 1 and 2 bits. The inputs are set to all the possible binary combination with 3 bits, with the following code.

```
    v_in.resize(bits, vector<long> (nslots,0));
    for(int i = 0; i < nslots; i++){
        bitset<64> bin(i % (unsigned)pow(2,bits)); //max is 2^64
        for(int b = 0; b < bits; b++){
            v_in[b][i] = bin[b]; //fill up matrix of bits
        }
    }
    for (int i = 0; i < bits; i++){
        k_constant[i] = he.encrypt(v_in[i]);
    }
```

Figure 6.1.4.a: *Combinational circuits unit tests inputs definition*

The variable *bits* is set to 3 in this case, so *bin* is the $\frac{nslots}{2^3}$ repetition of the sequence [000 001 010 011 100 101 110 111], hence covering all the combinations of 3 bits.

As for the logic gates, only one of the 10 unit tests will be explained here, the one for the full equality circuit, which is in the figure below.

```
bool TEST_CIRC_COMB::test_FLEQUAL(){
    if(debug){
        cout << className() << ": Running " << __FUNCTION__ << "..." << endl;
    }
    make_copies();
    he.FLEQUAL(k[0],k[1],k[2]);
    vector< vector<long> > v_out = he.decryptNbits(k);
    for (int i = 0; i < nslots; i++){
        if(v_out[0][i] != ((v_in[0][i] == v_in[1][i]) && v_in[2][i])){
            return true; //error
        }
    }
    return false;
}
```

Figure 6.1.4.b: *Unit test of the full equality combinational circuit*

First, *make_copies* and *he* are the same as for the logic gates unit tests. k[0], k[1] and k[2] are the map keys linking to the three ciphertexts generated from the binary matrix *v_in*. The resulting ciphertexts are then decrypted with the shortcut function *decryptNbits*, and the resulting plaintext vectors of binary values are checked against the C++ plaintext operation of the equality block operation. It for any plaintext slot this is not valid, the test has failed and the function returns *true* to the *Errors* object.

Again, the other unit tests are designed in the same way and share a common key pair generated once for whole set of tests. However, the level parameter *L* had to be raised to **5** to accommodate the most complex circuit, the full comparator, which require 4 homomorphic multiplications to operate correctly. Note that the level is set to 5 if the minimum bound of plaintext slots *slb* is not set (or set to 0). Otherwise, this level might have to be raised as the number of slots increases.

6.1.5 Sequential circuits unit tests

The collection of sequential circuits unit tests is more complicated as these circuits take two binary numbers with a flexible number of bits n. To greatly simplify the tests, input binary numbers are now generated randomly. The following code generates the inputs, converts them to matrices of bits and encrypt it into $2n$ ciphertexts (*N_numbers* is set to 2).

```
    inputs.resize(N_numbers, vector < long > (nslots,0));
    v_in.resize(N_numbers,vector<vector<long>>(bits,vector<long>(nslots,0)));
    k_constant.resize(N_numbers, vector < mkt>(bits));

    //inputs to N bit circuits
    for(unsigned i = 0; i < nslots; i++){
        inputs[0][i] = rand() % (unsigned)pow(2,bits);
        inputs[1][i] = rand() % (unsigned)pow(2,bits);
    }

    //Converts inputs to bits into v_in for parallel ciphertexts
    for(unsigned n = 0; n < N_numbers; n++){
        for(unsigned j = 0; j < nslots; j++){
            bitset<64> bin(inputs[n][j]); //max is 2^64 so max nbits = 64
            for(unsigned b = 0; b < bits; b++){
                v_in[n][b][j] = bin[b]; //first ctxt (b = 0) is LSB
            }
        }
    }

    //Encrypts all the vectors into ciphertexts
    for(unsigned n = 0; n < N_numbers; n++){
        for (unsigned b = 0; b < bits; b++){
            k_constant[n][b] = he.encrypt(v_in[n][b]);
        }
    }
```

Figure 6.1.5.a: *Sequential circuits testing – Generation of input ciphertexts*

Once these ciphertexts are created, they can be copied and used by each unit test. As before, only one of the 7 unit tests will be explained, for demonstration purposes: the test of the ripple carry adder circuit. Its code is written below.

```
bool TEST_CIRC_SEQ::test_RCADDER(){
    //NO 2's complement, MSB is used for "overflow" (last carry out)
    make_copies();
    t_start();
    he.RCADDER(k[0],k[1]);
    t_end(__FUNCTION__);
    vector<long> results(nslots);
    results = conv.matrix2LongVec(he.decryptNbits(k[0]));
    for (unsigned i = 0; i < nslots; i++){ //bit level
        if (results[i] != (inputs[0][i] + inputs[1][i])){
            return true;
        }
    }
    return false;
}
```

Figure 6.1.5.b: *Unit test of the Ripple carry adder*

This follows the same procedure as for the previous circuits. *T_start* and *t_end* are timing function which will be explained in the next part 6.2. The method *matrix2LongVec* from the *Conversion* object is used to convert the matrices of bits to a vector of integers. The result is then compared to the plaintext operation of the corresponding two inputs. All the other unit tests work in the same way.

As these circuits are not too slow to complete, a common key pair is still generated at the start of the overall test. However, as there is now the n bit parameter defining the number of bits of the binary numbers generated, the level parameter was designed to adapt to it. Indeed, several tests were conducted and the minimum level parameter L was recorded for the most complex circuits (Ripple comparator). A linear regression algorithm was then applied to the points in the form (n, L) in order to provide a best fit function to the level parameter. This allows the level parameter to be automatically adjusted to the number of bits n, without any user interaction.

6.1.6 Arithmetic circuits unit tests

The testing of the arithmetic circuits is exactly the same as for the sequential circuits except that each of the unit test now generate their own key pair. This was chosen because arithmetic circuits such as the multiplication operation usually take a long time to complete and have a different complexity. Indeed, for $n = 4$ for example, the multiplication needs a level of 13 and the Euclidean division circuit needs a level of 50.

The multiplication unit test works otherwise the same way as for example the ripple carry adder unit test. The unit test for the Euclidean division will however be shown to fully understand its working if it is not the case yet.

```
    vector<long> q(nslots), r(nslots);
    t_start();
    he.DIVIDE(k[0], k[1]);
    t_end(__FUNCTION__);
    r = conv.matrix2LongVec(he.decryptNbits(k[1]));
    q = conv.matrix2LongVec(he.decryptNbits(k[0]));
    ldiv_t expected;
    for (unsigned i = 0; i < nslots; i++){
        expected = div(inputs[0][i], inputs[1][i]);
        if ((expected.rem != r[i]) || (expected.quot != q[i])){
            return true;
        }
    }
    return false;
```

Figure 6.1.6.a: *Unit test of the Euclidean division arithmetic circuit*

The vectors r and q contain respectively the remainders and the quotients resulting from the homomorphic Euclidean division. The structure *expected* is a C++ Euclidean division result structure, and contain the result of the plaintext operation on *inputs[0]* and *inputs[1]* for each slot. It is then compared to r and q for each plaintext slot and will raise an error if there is a mismatch.

6.1.7 Average and Fast average circuit unit test

A unit test for the average circuit was implemented, where $N\_NUMBERS$ binary numbers of size n are generated and converted to ciphertexts similarly as for the other circuits. These are then passed to the homomorphic average circuit, which returns the quotient and remainder of the operation.

This test is actually the only test failing very quickly as the number of bits goes above 3 bits or the number of binary numbers goes above 2. This is not due to an implementation error, but to the complexity being way too high. If one of them is increased, the level parameter has to be greater than 70; the program then needs more than 4 gigabytes of RAM to run for a very long time. This is way too much resources and time lost in this circuit, and is the program is often killed by the operating system. So the test actually works for parameters such as $n = 3$ and $N = 2$ for example.

Regarding the fast average circuit, the unit test is the same as for the average circuit, except that only the quotient is verified (no remainder is calculated). Apart from the generation of the key and inputs, the unit test essentially is based on the following piece of code.

```
    vector<long> q(nslots);
    t_start();
    he.FAVERAGES(k);
    t_end(__FUNCTION__);
    q = conv.matrix2LongVec(he.decryptNbits(k[0]));
    ldiv_t expected;
    unsigned expected_sum;
    for (unsigned i = 0; i < nslots; i++){
        expected_sum = 0;
        for (unsigned n = 0; n < N_NUMBERS; n++){
            expected_sum += inputs[n][i];
        }
        expected = div(expected_sum, N_NUMBERS);
        if (expected.quot != q[i]){
            return true;
        }
    }
    return false;
```

Figure 6.1.7.a: *Unit test for the fast average circuit*

This one was passed successfully for various parameters such as N=4 and n=4.

As a conclusion to this part, every logic gate and every circuit implemented have their corresponding efficient unit tests. All the tests are correctly implemented and are all passed by the circuits. Therefore, all the circuits work fully. The unit tests also serve as example as how to use the circuits developed for a custom application.

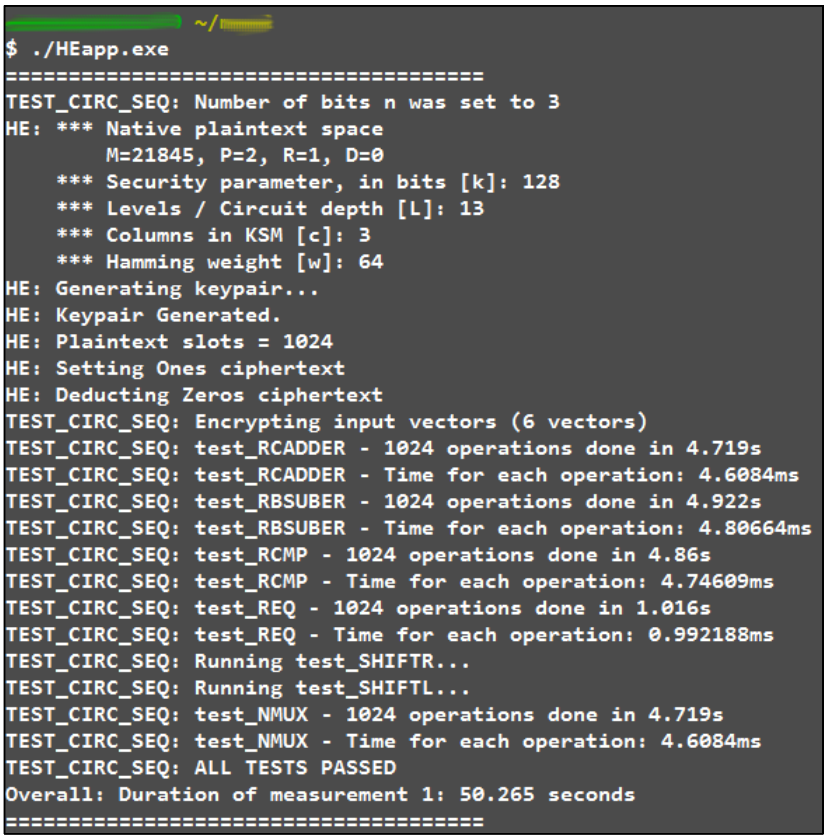
6.2 Time measurements

A relatively simple object class called *Timing* was implemented in *helper_functions.cpp* and serves as an easy, reusable and precise timer. It is not used for the logic gates and combinational circuits tests as these are relatively quick and give stable near-constant timing results. On the other hand, the time of execution of sequential and arithmetic circuits highly depend on the number of bits n, which yield real-world use cases results. This is why these two last kind of circuits are timed for various values of n. Note that this timing function is only enabled if the *debug* flag is set to true in the highest layer of the code.

Both test files *TEST_CIRC_SEQ* and *TEST_CIR_ARITHM* implement the methods *t_start()* which starts the timer, and *t_end(string name)* which displays the overall time taken by the "name" test, as well as its time per operation. Indeed, because the SIMD mode is used, the time per operation is defined by:

$$Time\ per\ operation = \frac{Overall\ Time}{Number\ of\ plaintext\ slots\ (nslots)}$$

As an example display, the following screenshot would be obtained for the set of sequential circuits unit tests.



```
[redacted] ~/[redacted]
$ ./HEapp.exe
=====================================
TEST_CIRC_SEQ: Number of bits n was set to 3
HE: *** Native plaintext space
        M=21845, P=2, R=1, D=0
    *** Security parameter, in bits [k]: 128
    *** Levels / Circuit depth [L]: 13
    *** Columns in KSM [c]: 3
    *** Hamming weight [w]: 64
HE: Generating keypair...
HE: Keypair Generated.
HE: Plaintext slots = 1024
HE: Setting Ones ciphertext
HE: Deducting Zeros ciphertext
TEST_CIRC_SEQ: Encrypting input vectors (6 vectors)
TEST_CIRC_SEQ: test_RCADDER - 1024 operations done in 4.719s
TEST_CIRC_SEQ: test_RCADDER - Time for each operation: 4.6084ms
TEST_CIRC_SEQ: test_RBSUBER - 1024 operations done in 4.922s
TEST_CIRC_SEQ: test_RBSUBER - Time for each operation: 4.80664ms
TEST_CIRC_SEQ: test_RCMP - 1024 operations done in 4.86s
TEST_CIRC_SEQ: test_RCMP - Time for each operation: 4.74609ms
TEST_CIRC_SEQ: test_REQ - 1024 operations done in 1.016s
TEST_CIRC_SEQ: test_REQ - Time for each operation: 0.992188ms
TEST_CIRC_SEQ: Running test_SHIFTR...
TEST_CIRC_SEQ: Running test_SHIFTL...
TEST_CIRC_SEQ: test_NMUX - 1024 operations done in 4.719s
TEST_CIRC_SEQ: test_NMUX - Time for each operation: 4.6084ms
TEST_CIRC_SEQ: ALL TESTS PASSED
Overall: Duration of measurement 1: 50.265 seconds
=====================================
```

Figure 6.2.a: *Example screenshot of the Timing measurements displayed*

Note also that for the timing measurements per operation to be more accurate, the *slb* parameter (lower bound of plaintext slots) is set to 1023 to force all the operations to use 1024 plaintext slots.

A final important element is that all the timing tests are executed on a single core of an i5 6500 CPU processor.

6.3 Pushing the circuits to their limit

All the source code has been built with the aim of being able to easily tweak key generation parameters and algorithms used. Indeed, the level parameter, the minimum number of slots as well as other parameters such as the level of security *k* can be changed quickly. The many sequential and arithmetic circuits implemented can probably be enhanced with better algorithms which can then be easily tested with the unit tests previously described.

But circuit should be pushed to their limit really by testing out how to get the lowest time per operation for various numbers of plaintext slots and level parameter values.

It should also be noted that most if not all the code implemented contain a *debug* switch variable which can be set to the user to display information, warning and more error messages.

Note that the amount of RAM used is generally not an issue but becomes a limit for complex circuit requiring very high level parameter greater than 65, such as the average circuit.

Chapter 7

Results

This chapter concerns the results obtained from tweaking the parameters, timing the circuits and comparing them. It is structured as follows. First, timing and behaviour of homomorphic logic gates is analysed and described. Analysis of the performance behaviour of the combinational and sequential circuits is then covered. Finally, the time analysis of the arithmetic circuits is carried out and highlights the limits of the homomorphic encryption.

7.1 Logic gates

Even if the time a homomorphic logic gate takes to complete is not relevant in itself, it is important to have an idea of how much time a gate takes to complete relatively to other ones. Note that their completion time depend on the level parameter so it should more be taken as a relative information.

To measure this, the level parameter was kept at 3 and the following code was executed for each gate.

```
make_copies();
t.start();
for (unsigned i = 0; i < 1000; i++){
    make_copies();
    he.AND(k[0],k[1]);
}
t.end();
```

Figure 7.1.a: *Logic gate time measurement code*

This runs 1000 times the logic gate on copied ciphertexts - to keep the level parameter at 3 without error. The final time measurement is then divided by 1000 to yield the time a particular logic gate needs to complete.

The following results were obtained, for the levels 3, 10, 15, 18, 20, 30 and 50.

Time in MS								
Level L \ Logic gate	XOR	AND	NOT	NAND	OR	NOR	XNOR	**M/A ratio**
4	0.97	12.06	0.94	12.23	12.81	12.55	0.94	13.06578947
10	4.68	82.35	4.53	83.75	85.31	84.53	4.53	18.33733624
15	10.3	270.2	10.5	272.5	277.2	276.6	10.62	26.17979943
18	15.6	320.3	14.1	323.4	331.3	331.2	15.6	21.62582781
20	19.8	642	19.8	646	643.3	642.6	19.76	32.48144876
25	28.1	892.2	31.2	909.4	917.2	900	31.3	29.95695364
30	37.5	959	37.5	967	998	984	37.5	26.05333333
50	119	3172	118	3185	3180	3165	120	26.68487395

Figure 7.1.b: *Time in milliseconds of homomorphic logic gates for different Level parameters, and M/A ratio for different level parameters*

This table shows the time in milliseconds for each logic to complete on various levels. The rightmost column contain the M/A ratio which actually is the ratio of time needed for a homomorphic multiplication divided by the time needed for a homomorphic addition (XOR, NOT, XNOR gates). These results yield the following two plots.

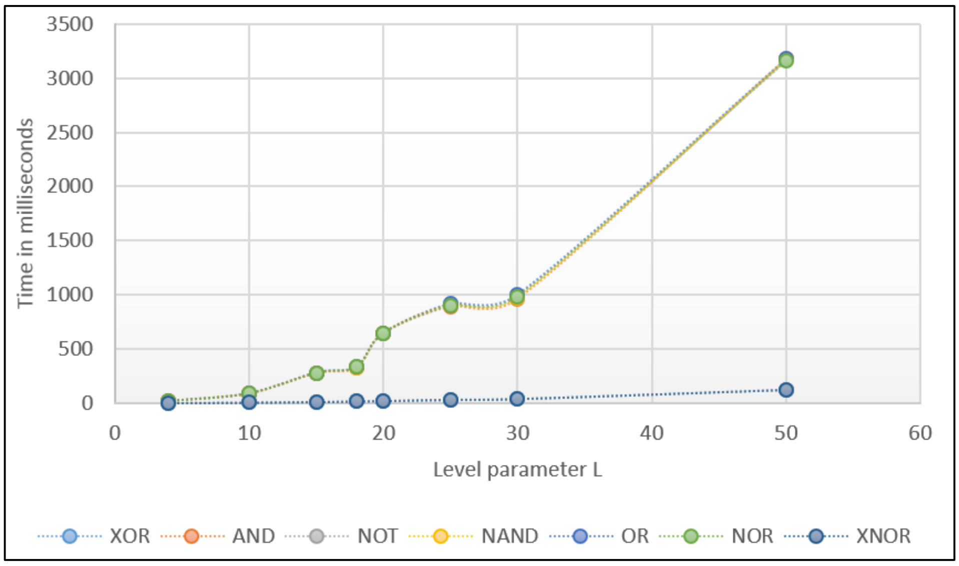


Figure 7.1.c: *Time in milliseconds of homomorphic logic gates vs Level*

The logic gates are grouped into two subsets, one where the homomorphic addition is needed, and the other one where it is the multiplication. The bottom line (blue) represents the time evolution of the gates requiring one or more additions operations only, whilst the green one groups the other gates requiring the multiplication operation. This shows that the homomorphic multiplication not only adds noise to the ciphertext (and thus forces L to be higher), but also consumes more time as the level grows higher. The next figure shows a plot of the M/A ratio.

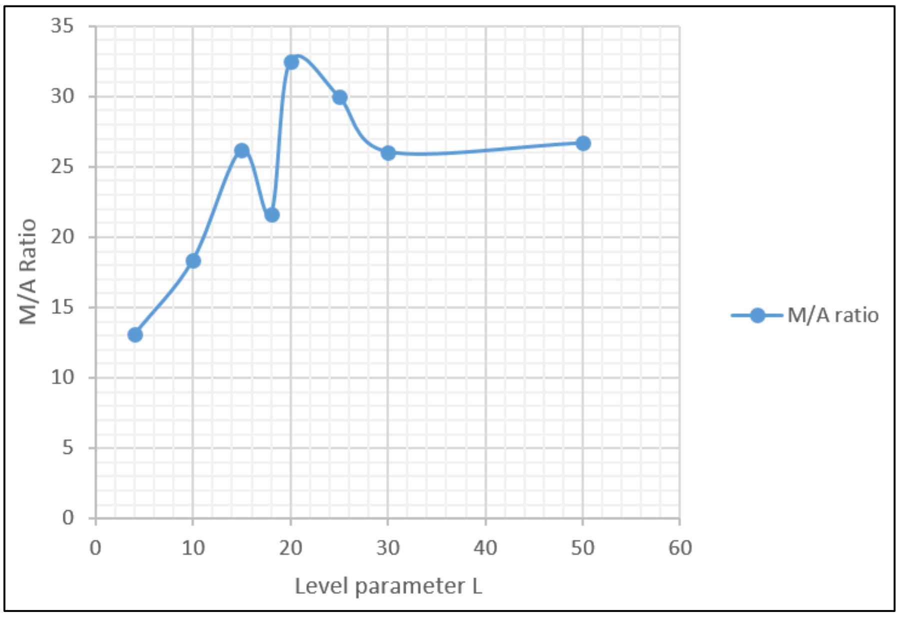


Figure 7.1.d: *M/A ratio vs Level*

This shows that time needed for multiplication is not proportional to the time needed for a homomorphic addition. However, it reaches its maximum at L=20 to stabilises at a ratio of 26. This means that for very complex operations, the multiplication requires 26 times the time of a homomorphic addition.

Another small element noticed was that the security parameter k needed at the key pair generation can influence the speed of execution in some cases. It was seen that the gates were the fastest for $k = 128$, which is actually also a secured enough parameter. Hence all the unit tests have their security parameter set to 128.

The next part will now cover results obtained from the implemented combinational circuits.

7.2 Combinational circuits

The following table shows all the combination circuits with their complexity and their time of execution is milliseconds.

Circuit name	Complexity	Time in ms
Half Adder	1	16
Full Adder	3	78
Half Subtractor	1	31
Full Subtractor	3	78
Half equality comparator	0	0.2
Full equality comparator	1	31
Lower Than comparator	1	16
Half comparator	1	16
Full comparator	4	94
Multiplexer 2:1	3	62

Figure 7.2.a: *Combinational circuits complexity and time, for L=5*

This highlights how the complexity is also related to time needed for a circuit to complete. Note some circuits have the same complexity but different duration. This is due to the fact that L is set to a low value of 5 (not for real use cases) so an a homomorphic multiplication is only 12 times longer than a homomorphic addition, according to figure 7.1.d. To verify this, the timing measurements was performed again at L=15.

Circuit name	Complexity	Time in ms
Half Adder	1	281
Full Adder	3	1016
Half Subtractor	1	281
Full Subtractor	3	991
Half equality comparator	0	0
Full equality comparator	1	297
Lower Than comparator	1	281
Half comparator	1	297
Full comparator	4	1297
Multiplexer 2:1	3	1000

Figure 7.2.a: *Combinational circuits complexity and time, for L=**15***

For L=15, the M/A ratio is higher at 27 instead of 12 and thus the complexity is now fully correlated with the time required for a circuit to complete. This also signifies that for complex circuits requiring a high level parameter L, the homomorphic addition will not influence the time required for this circuit. Instead, only the homomorphic multiplication will impact as it has been shown here.

7.3 Sequential circuits

It is very difficult to fully explore the sequential circuits. In this project, the focus will be on changing the level parameter, the *slb* parameter (lower bound of plaintext slots) and the number of bits *n* for the input binary numbers. The number of bits was limited to 16, as going above will often take very long to test and may as well create some memory errors due to the program trying to use more than 2 Gigabytes of RAM (limit which could be lifted thought).

The ripple carry adder was extensively tested for all values of n ranging from 2 to 16, and with a *slb* parameter of 800. This effectively forced the number of plaintext slots to be 1024 for low levels, and 1800 for higher level parameters. The following table of results was obtained.

Ripple carry adder	*Security 128 bit*				
Number of bits n	Complexity	Minimum level	Time in seconds	Plaintext slots	Timer per operation in milliseconds
2	7	7	1.828	1024	1.78515625
3	10	9	3.235	1024	3.159179688
4	13	12	5.016	1024	4.8984375
5	16	13	6.984	1024	6.8203125
6	19	15	10.453	1024	10.20800781
7	22	17	14.125	1024	13.79394531
8	25	19	17.875	1024	17.45605469
9	28	21	21.36	1024	20.859375
10	31	23	40.491	1800	22.495
11	34	27	40.828	1800	22.68222222
12	37	29	46.657	1800	25.92055556
13	40	31	50.781	1800	28.21166667
14	43	34	59.797	1800	33.22055556
15	46	35	70.906	1800	39.39222222
16	49	37	79.859	1800	44.36611111

Figure 7.3.a: *Time measurements and level for the ripple carry adder*

As it was expected, the overall time and time per operation both increase with the complexity of the circuit. However, it is interesting to compare the evolution of these two times, which is done in the following graphs.

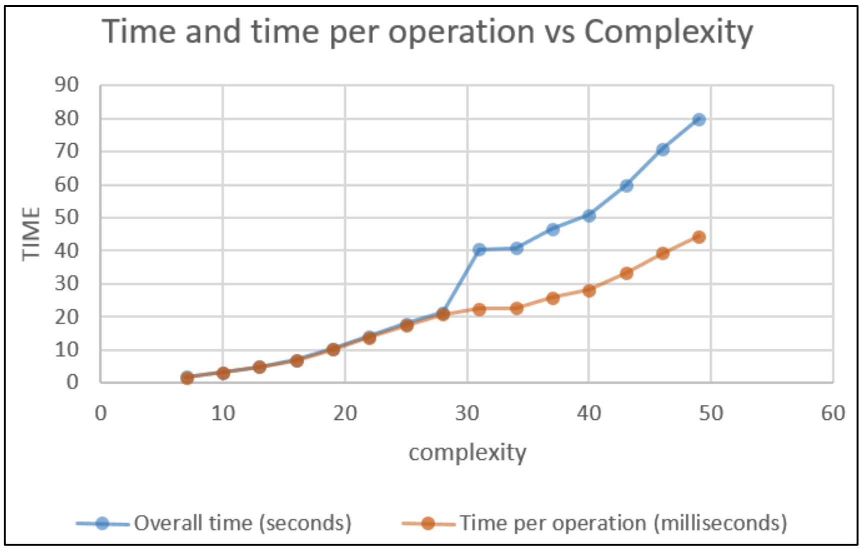


Figure 7.3.b: *Overall time and time per operation vs Complexity, for the ripple carry adder*

This shows that the overall time and timer per operation raise proportionally up to the level where the number of slots switches from 1024 to 1800. The overall time then raises more than the time per operation. This implies that the raising the number of plaintext slots could increase the overall time of execution but would actually lower the time per operation. To verify this, we performed the tests again with the *slb* parameter set to 1700, for a number of bits ranging from 2 to 8. The timing results are then plotted together with the previous obtained where there were only 1024 plaintext slots.

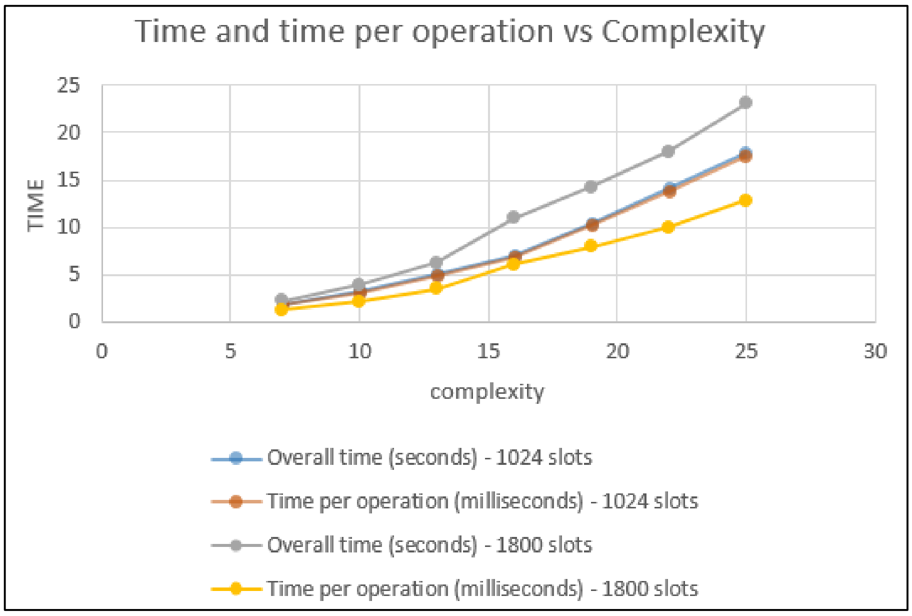


Figure 7.3.c: *Overall time and time per operation vs Complexity, for the ripple carry adder, for 1024 and 1800 plaintext slots*

As it can easily be seen on this graph, the 1800 slots version takes a longer time to complete overall but the SIMD mode pays off here. Indeed, the time per operation is shorter than for the 1024 slots version. On top of this, the gap appears to be even larger as the complexity grows. This implies that for complex homomorphic circuits, if the time per operation is more important that the overall time of execution, the plaintext slots should be expanded to take full profit of the SIMD mode. This should also answer the question raised about SIMD in part 4.5.

In a more practical context, this graph shows that for the addition of two binary numbers of 8 bits (corresponding to the complexity 25), it would take 13 milliseconds per addition for 1800 slots packed in the each ciphertext, and 24 milliseconds for the 1024 slots version. This could be very useful for reducing the time of massively parallel computations.

The exact same results are obtained for the ripple borrow subtractor as it has essentially the same complexity. All the other sequential circuits with a positive homomorphic complexity are governed by the same rules, summarised here:

- Overall time and time per operation increases almost linearly with complexity for the same number of plaintext slots.
- Increasing the number of plaintext slots reduces the time of execution per operation (SIMD)
- Increasing the number of plaintext slots increases the overall time of execution

Note that the set of unit tests for the sequential circuits contain a *switch-case* structure acting as a look-up table for the best level parameter for each number of bits, for the ripple carry adder as well as all the other circuits with the same or a lower complexity. The ripple comparator should not work as its complexity is higher for a number of bits greater than 4, however it actually works for the level parameters of the ripple carry adder in all cases. The following part will explore the performance of the more complex arithmetic circuits implemented.

7.4 Arithmetic circuits

This concerns the results obtained from the binary multiplication and the Euclidean division circuit and the two average circuits. From the trial and error method, the following minimum level parameters were obtained for various values of n (2 to 9 bits) and for 1024 plaintext slots.

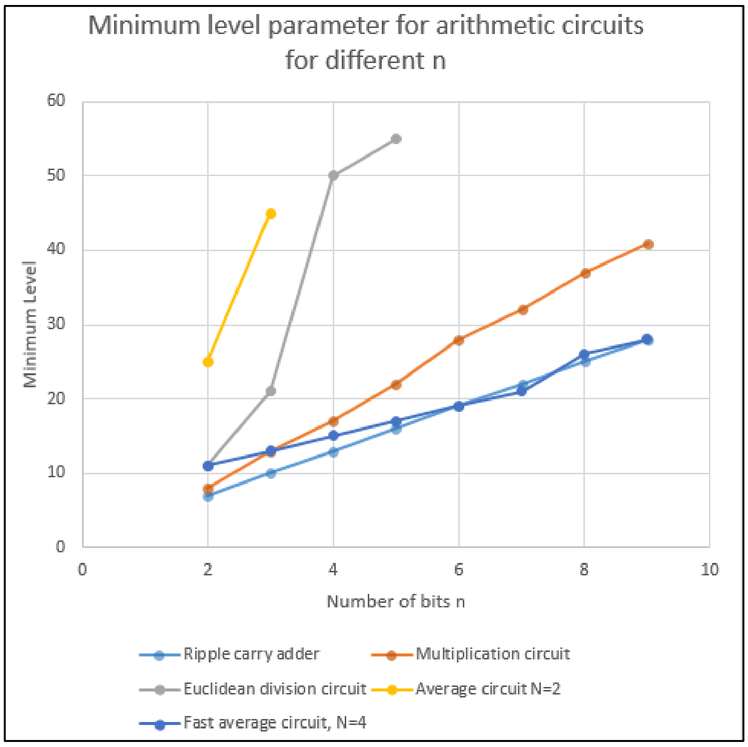


Figure 7.3.c: *Minimum level parameter for arithmetic circuits, for various n.*

The average circuit could only be tested with up to 2 numbers of 3 bits. This already required a very high minimum level of 45. For numbers of 4 bits, the minimum level should be around 85 but this would require too much RAM and time and would often fail because of Memory Errors. On the other hand, the fast average circuit could easily take 4 binary numbers of different size as it doesn't use the Euclidean division circuit.

Regarding the Euclidean division circuit, it could not reach numbers with more than 5 bits as the associated minimum level parameter would then be around 70, producing memory errors and demanding way too much time.

As a reference point, the minimum level parameters for the ripple carry adder were added on this plot. The multiplication circuit requires decent levels L, and does not raise any issue for number of less than 10 bits. Over that limit, it will be very slow and will require very high level parameters.

Regarding the fast average circuit, the result is quite a surprise. It actually performs very closely to the ripple carry adder, even though it actually uses $N - 1$ times the ripple carry adder, with an accumulator growing in size at each iteration. After a lot more testing, this is still a mystery, which is actually great here.

The following shows the time per operation in milliseconds of each arithmetic circuit for values of n ranging from 2 to 8.

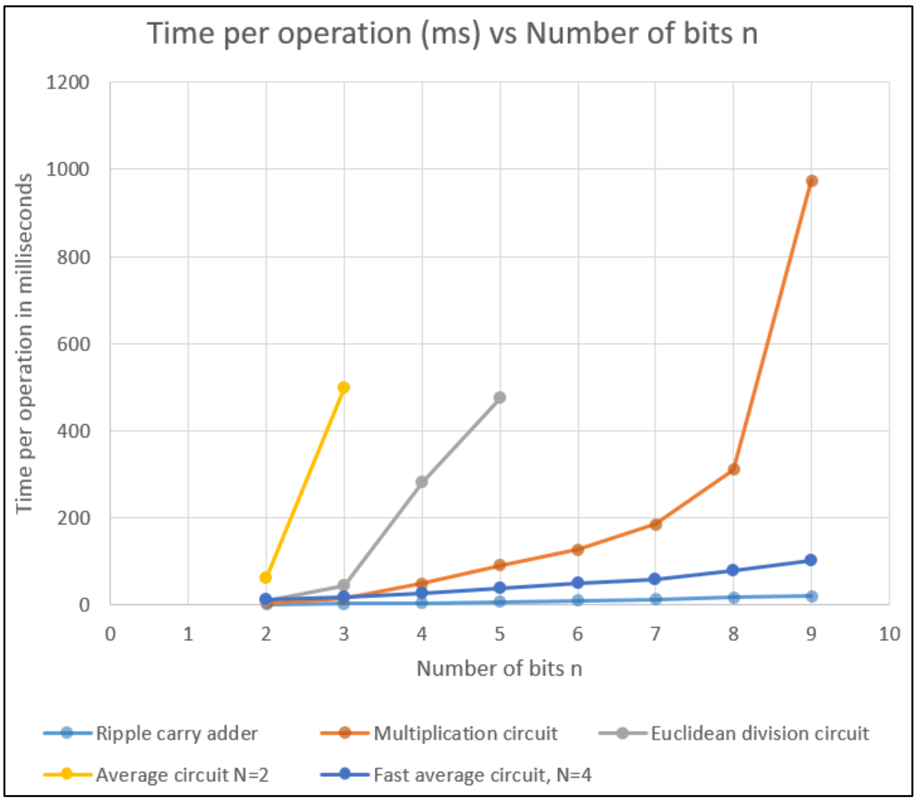


Figure 7.3.c: *Time per operation (milliseconds) for arithmetic circuits, for number of bits n ranging from 2 to 8.*

This figure highlights the very slow operation of the average and Euclidean division circuit, which should actually be avoided for real use. The multiplication circuit is relatively efficient but will tend to be exponentially slower as n grows beyond 7, as shown on the graph. It more precisely reaches its maximum capability at numbers of 9 bits. Now the fast average circuit is relatively fast and efficient. For the cost of losing the “floating point” which would be in the form of a remainder, its time of execution can’t even be compared with the pure average circuit. Even if you lose some precision and can only perform average on N numbers where N is a power of 2, this should actually be a circuit used for real cloud computing purposes.

Chapter 8

Evaluation

From logic gates to complex arithmetic circuits, this project went quite far in terms of pushing existing FHE technologies such as HElib to usability. Even if FHE is still slow, it was tested for concrete applications and some features were found to be possibly useful.

First of all, logic gates implemented are actually quite fast with the SIMD mode. For a level parameter below 20, they each take a maximum of 350 milliseconds to complete which is decent enough. Even if this value is far from the nanoseconds that present silicon logic gates take, it is still a good starting point. Above the 30 level their time of execution however increases substantially and become not very usable anymore.

Binary additions, subtractions and comparisons take between 1 and 50 milliseconds per operation for numbers between 2 bits and 16 bits, as it has been shown in the results section concerning the sequential circuits. This is a decent time which could still be used for cloud computing purposes.

The binary multiplication implemented uses the addition circuit as well as the AND logic gate. It also has a decent time of execution for numbers up to 8 bits. It could therefore also be used for a few operations in a cloud computing context.

The Euclidean division circuit shows however the limit of homomorphic encryption. For only 4 bits numbers, this circuit requires a level of 50, requiring some 2.6 Gigabytes of RAM and taking 280 milliseconds per operation. The time is not too bad, but as soon as the number of bits in increased it becomes exponential. For 5 bits numbers, 475 milliseconds per operation and 3.5 Gigabytes of RAM are required for example. This circuit has very long execution times and requires ridiculous amount of memory, which is not convenient at all.

The most complex circuit, the homomorphic average, uses both the addition and the Euclidean division circuits. Its time of execution and the amount of RAM it needs are therefore enormous again. It is therefore not usable at all for now.

Finally, a trick version of the average circuit, called the *fast average circuit*, uses binary right shifts instead of the Euclidean division to only provide the quotient result of an average of N numbers, where N is a power of 2. It is very light and fast and can therefore be used without trouble for numbers of 16 bit for example.

The project has thus implemented two version of the average operation, and has shown that some of the operations implemented can be used with decent times.
The limitation of the average circuit was slightly expected since the beginning, but there exist many ways to enhance it with further work, as it will be shown in the following chapter.

Chapter 9

Further Work

9.1 Changing the algorithms

The logic gates and combinational circuits are already optimised and there will probably be nothing else to do. However, many of the sequential circuits could be changed and optimised.

The ripple carry adder could be replaced with a ripple-block carry look-ahead adder (RCLA), or a block carry look-ahead adder (BCLA) or with even more performant parallel prefix adders such as the Han-Carlson adder. These designs are very complex to understand but would lower the complexity of the ripple carry adder. The ripple borrow subtractor could also benefit from the symmetric design. As these two circuits are the actual limiting factor of more complex operations such as the Euclidean division circuit, this would certainly propose a greater horizon of development.

In addition, because the wiring complexity and the “power consumption” of current digital circuits designed should not be a limit in this context, more complex algorithms such as the Wallace tree could also be implemented for the addition operation circuit.

The N:1 multiplexer used in the Euclidean division circuit and the ripple borrow subtractor circuit could both be replaced by an efficient conditional subtractor. However, it will be difficult to maintain the flexibility of having a variable number of bits for the inputs of the system.

9.2 Adding homomorphic circuits

Parallel circuits for addition or multiplications could be implemented to increase the throughput of the operations. This would effectively reduce the time per operation without raising the level parameter too much.

Other circuits such as the multiply-accumulator (MAC) could also be added to provide a better performance in some cases.

There are also many circuits such as parity check circuit or the square root circuit which could be implemented to provide even more homomorphic features.

9.3 Using shifts for plaintext constant multiplications

When combined with the addition operation, the binary shifts can multiply a number very quickly. This could be implemented and adapted for multiplying an encrypted number (in the form of multiple ciphertexts) by a constant number known by the cloud computer.

9.4 Bootstrapping

In some cases, such as for the Euclidean division, where the level has to be very high, bootstrapping is faster than the levelled homomorphic encryption. As HElib supports it, it could be implemented in highly complex circuits otherwise requiring a high level.

9.5 Parallel computing and GPUs

The code of this project was always running on a single core of the processor, so it could potentially benefit from using multiple cores. Even better, it could be running on a graphics processor unit (GPU) with the cuHE library. Of course, this would require to rewrite the code in its entirety. But the speed gains should probably be up to 500 times, which is not negligible. This is an ambitious future work to be done.

9.6 Client – Cloud computer interface and networking

Once the performance of the circuits and the features are satisfactory, the whole code could be split into a part for the cloud computer and a cloud for the client. Note that this has been commented in the *he.cpp* file to make the task easier. The client would essentially need the key generation, the encryption and the decryption functions. On the other hand, the server would need all the homomorphic operations and ciphertexts handling. A more user friendly interface could be implemented, starting with the Conversion functions defined in *helper_functions.cpp* precisely.

Overall, there are many possibilities and work that can be carried out from this project. I hope someone will make great use of what has been done. Please do not hesitate to contact me if you have any question regarding these possibilities and enhancements.

Chapter 10

Conclusion

The project has shown several facts. First, it has demonstrated how real cloud computing calculations could be implemented with homomorphic encryption. The limits of current homomorphic encryption schemes were explained and analysed. Several tricks were found to overcome those and to design efficient homomorphic binary circuits. Some complex circuits such as the pure average circuit clearly showed the bounds of homomorphic encryption. But with enough design cleverness and compromises, it was also demonstrated that homomorphic encryption can be used in a few cases. There are also many horizons to continue this project as described in chapter 9. I hope you enjoyed reading this long report and wish you the best if you plan on continuing this project.

Chapter 11

User guide

As it has been explained in the background chapter, several libraries and files have to be downloaded, compiled and installed on the machine to support the project's program developed. A makefile was thus designed to simplify this long installation. You can refer to the webpage http://qdm12.github.io/hbc/ where all the installation procedure is described for all the platforms. You can also access the GitHub repository at https://github.com/qdm12/hbc. The installation will be described here as well for convenience.

By default, the program built runs all the sets of unit tests on the homomorphic circuits, from the *main.cpp* source file actually. You can run the built program *HEapp* with *./HEapp*.

The installation requires an internet connection to download source files and other libraries. The first installation method is by using a makefile I implemented which works for Linux and Cygwin. The second method is to do everything manually. This may be more safe for Mac OSX, OS on which the makefile has not been tested yet.

11.1 Makefile, for Cygwin and Linux

NOTE 1: The makefile has to be copied in an empty directory.
NOTE 2: **FOR CYGWIN ONLY** – Due to permission restrictions on Windows OS, the modules *git* and *gcc-g++* have to be installed manually with the Cygwin installer before launching the makefile.
NOTE 3: The makefile will automatically install the following modules

- apt-cyg (windows only)
- curl
- m4
- perl
- git and g++ (Linux only)

To download, compile and install HElib and other libraries: `make HElib`
To download, compile and run the project: `make project`
To re-compile the project *source* directory & run HEapp: `make HE`
For more information: `make help`

11.2 Manual Installation

The manual installation is explained in details on the Github webpage precisely here.

Bibliography

1. Title: *A fully homomorphic scheme*
   Publishing body: *Stanford*
   Author: *Craig Gentry*
   Date: September 2009
   Link: https://crypto.stanford.edu/craig/craig-thesis.pdf

2. Title: *Fully Homomorphic SIMD Operations*
   Authors: *N.P. Smart and F. Vercauteren*
   Date: *15 March 2011*
   Link: *https://eprint.iacr.org/2011/133.pdf*

3. Title: *Fully homomorphic Encryption with Polylog Overhead*
   Authors: *Craig Gentry, Shai Halevi, Nigel P. Smart*
   Date: *19 October 2011*
   Link: *https://eprint.iacr.org/2011/566.pdf*

4. Title: *Fully homomorphic Encryption over the Integers (BGV scheme)*
   Authors: *Craig Gentry, Shai Halevi, Marten van Dijk and Vinod Vaikuntanathan*
   Date: *11 December 2009*
   Link: *http://eprint.iacr.org/2009/616.pdf*

5. Title: *HElib*
   Authors: *Shai Halevi (IBM) and Victor Shoup (NYU)*
   Date: *25 September 2015*
   Link: *http://shaih.github.io/HElib*
   Github: *https://github.com/shaih/HElib*

6. Title: *Algorithms in HElib*
   Authors: *Shai Halevi (IBM) and Victor Shoup (NYU)*
   Date: *11 February 2014*
   Link: *http://eprint.iacr.org/2014/106.pdf*

7. Title: *Design and Implementation of a Homomorphic-Encryption Library*
   Authors: *Shai Halevi (IBM) and Victor Shoup (NYU)*
   Date: *19 November 2014 (re-compiled in 30 January 2016)*

   Design and Implementation of a Homomorphic-Encryption Library - Shai Haveli (IBM) and Victor Shoup (NYU) - 2014.pdf

8. Title: *NTL library*
   Author: *Victor Shoup (NYU)*
   Link: *http://www.shoup.net/ntl/*

9. Title: *GMP library*
   Link: *https://gmplib.org/*

10. Title: *HEIDE: An IDE for the homomorphic encryption library HElib*
    Authors: *Grant Frame (Faculty of California Polytechnic State University)*
    Date: *June 2015*
    Link:
    http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=2523&context=theses
    GitHub: *https://github.com/heide-support/HEIDE*

11. Title: *An R package for fully homomorphic encryption*
    Authors: *Louis J. M. Aslett*
    Date: *3 August 2015*
    Link: *http://www.louisaslett.com/HomomorphicEncryption/*
    Link: *http://www.louisaslett.com/TechReports/Aslett_Esperanca_Holmes_2015a.pdf*

12. Title: *Somewhat practical fully homomorphic encryption*
    Authors: *Junfeng Fan and Frederik Vercauteren*
    Date: *17 march 2012*
    Link: *https://eprint.iacr.org/2012/144.pdf*

13. Title: *cuHE: A Homomorphic Encryption Accelerator Library*
    Authors: *Wei Dai and Berk Sunar (Worcester Polytechnic Institute, USA)*
    Date: *17 August 2015*
    Link: *https://eprint.iacr.org/2015/818.pdf*
    Github: *https://github.com/vernamlab/cuHE*

14. Title: *Homomorphic AES Evaluation Using NTRU*
    Authors: *Yarkin Doroz, Yin Hu, Berk Sunar (Worcester Polytechnic Institute)*
    Date: *11 January 2014*
    Link: *https://eprint.iacr.org/2014/039.pdf*

15. Title: *On-the-fly multiparty computation on the Cloud via Multikey FHE*
    Authors: *Adriana Lopez-Alt, Eran Tromer, Vinod Vaikuntanathan*
    Date: *22 October 2014*
    Link: *https://eprint.iacr.org/2013/094*

16. Title: *krypto: C++ Implementation of Multivariate Quadratic FHE*
    Authors: *kryptnostic*
    Date: *2015-2016*
    Link: *https://www.kryptnostic.com/*
    Github: *https://github.com/kryptnostic/krypto*

17. Title: *FHEW: Bootstrapping homomorphic encryption in less than a second*
    Authors: *Leo Ducas (Centrum Wiskunde & Informatica, Amsterdam) and Daniele Micciancio (University of California, San Diego)*

Date: *8 October 2014*
Link: *https://eprint.iacr.org/2014/816.pdf*
Github*: https://github.com/lducas/FHEW*
Slides: *http://www.math.uci.edu/~asilverb/CryptoDayFiles/uci15b.pdf*

18. Title: *Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers*
Authors: *Jean-Sebastien Coron, David Naccache and Mehdi Tibouchi*
Date: *January 2012*
Link: *http://eprint.iacr.org/2011/440.pdf*
Github: *https://github.com/coron/fhe*

19. Title: *Homomorphic Encryption and applications (book)*
Authors: *Xun Yi, Russell Paulet and Elisa Bertino*
Date: *September 2014*
Link: *http://www.springer.com/us/book/9783319122281*

20. Title: *Divide algorithm version 3, from ALU design: Division and Floating Point*
Authors: *Ann Gordon-Ross, Electrical and computer engineering, University of Florida*
Link: *http://www.ann.ece.ufl.edu/courses/eel4713_12spr/slides/Lec8-division.pdf*

Appendices

A.1 List of abbreviations

FHE:	Fully homomorphic encryption
SIMD:	Single-Instruction-Multiple-Data
SwHE:	Somewhat homomorphic encryption
NTL:	Number Theory Library
GMP:	GNU Multiple Precision Arithmetic
DHS:	Doroz-Hu-Sunar
LTV:	Lopez-Tromer-Vaikumtanathan
GPU:	Graphics processing unit
API:	Application Programming interface
XOR:	Exclusive OR logic gate
Mkt:	Map key type
PKI:	Public key infrastructure
LSB:	Least significant bit
MSB:	Most significant bit