Valentin Dalibard

Implementing Homomorphic Encryption

Computer Science Tripos, Part II

St John's College

May 18, 2011

Proforma

Name:	**Valentin Dalibard**
College:	**St John's College**
Project Title:	**Implementing Homomorphic Encryption**
Examination:	**Computer Science Tripos, Part II, June 2011**
Word Count:	**11 889**
Project Originator:	**Valentin Dalibard and Malte Schwarzkopf**
Supervisor:	**Malte Schwarzkopf**

Original Aims of the Project

The implementation of a cryptosystem supporting arbitrary computations on encrypted bits. This includes the implementation of a somewhat homomorphic scheme supporting a limited number of operations on encrypted bits, and its extension to a fully homomorphic scheme by providing a homomorphic version of the decryption function.

Work Completed

The project has been a success. I produced a working implementation of the fully homomorphic encryption scheme, and, beyond this, developed a new theoretical approach to understanding how the scheme works. I then used this understanding to extend the cryptosystem from working on bits to working on modular integers, and implemented functions to switch between the two representations. Finally, I parallelized the implementation so it can run on multiple CPUs or machines. Overall, the current implementation is orders of magnitude faster than the current best published implementation.

Special Difficulties

None.

Declaration

I, Valentin Dalibard of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date May 18, 2011

Acknowledgments

A large number of people have contributed to this project with their advice and feedback, but I owe special thanks to:

- **Malte Schwarzkopf**, for encouraging me to do this challenging project, supervising it, and providing advice and help all throughout the year.
- **Dr Richard Gibbens**, for very helpful advice on how to approach some of the theoretical challenges I was facing.
- **Prof. Nigel Smart** at the University of Bristol, for insightful observations on some of the new perspectives taken in this project.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Challenges . . . 2
1.3 Overview of the project . . . 2

2 Preparation **5**
2.1 Notations . . . 5
2.2 Introduction to Lattice-based Cryptosystems . . . 5
2.2.1 Rings, Ideals and Ideal Lattices . . . 5
2.2.1.1 Rings and Ideals . . . 5
2.2.1.2 Lattices . . . 6
2.2.1.3 Ideal Lattices . . . 8
2.2.2 Lattice-based Cryptosystems and Homomorphic Encryption . . . 8
2.2.3 Lattice-based cryptosystem . . . 8
2.2.3.1 Gentry's Somewhat-Homomorphic Cryptosystem . . . 10
2.2.3.2 Gentry's Fully-Homomorphic Encryption scheme . . . 11
2.3 Choice of Tools . . . 12
2.3.1 Programming Language and Libraries . . . 12
2.3.2 Development Environment . . . 13
2.4 Software Engineering Techniques . . . 13

3 Implementation **15**
3.1 Implementing Gentry's scheme . . . 15
3.1.1 Encryption . . . 15
3.1.2 Decryption . . . 17
3.1.3 Key Generation . . . 17
3.1.4 Extension to a Fully Homomorphic scheme . . . 18
3.1.5 An optimised encryption procedure . . . 21
3.2 A new understanding of the somewhat homomorphic scheme . . . 22
3.3 Increasing the message space . . . 24
3.4 Speeding up key generation for large message spaces . . . 25
3.5 Recrypting an integer modulo a power of 2 to bits . . . 26
3.6 Distributing the scheme . . . 28
3.6.1 Asynchronous Operations in the Master Node . . . 29
3.6.2 Distributing the encryption . . . 30
3.6.2.1 Computing the powers of r . . . 30
3.6.2.2 Computing the encryptions of the bits . . . 31

3.6.3 Distributing the recryption . . . 31
3.6.3.1 Distribution of the sumlines operations . . . 31
3.6.3.2 Distributing Grade School Addition . . . 32
3.6.3.3 Distributing the sum of hot bits numbers . . . 33
3.7 Conclusion . . . 33

4 Evaluation **35**
4.1 Test of correctness and basic performance evaluation . . . 35
4.1.1 Unit test of implementation . . . 35
4.1.2 Basic performance of the implementation . . . 37
4.1.2.1 Time and space complexity . . . 37
4.1.2.2 Degree of the recryption . . . 40
4.2 Experimental results on the somewhat homomorphic scheme . . . 42
4.2.1 Distribution of the w_i's . . . 42
4.2.1.1 Determining the distribution . . . 42
4.2.1.2 Size of the standard deviation . . . 44
4.2.2 Impact on the scheme . . . 44
4.3 Performance of my implementation . . . 48
4.3.1 Performance of bit multiplication . . . 48
4.3.2 Evaluation of the large message space against the bit representation . . . 49
4.3.3 Parallelized performance . . . 50
4.4 Conclusion . . . 51

5 Conclusion **53**

A **57**
A.1 Rings and Ideals . . . 57
A.2 Lattice-based cryptosystems . . . 58
A.2.1 Lattice problems . . . 59
A.2.2 GGH-type Cryptosystems . . . 61
A.3 Mathematical demonstrations used in the implementation of Gentry's scheme 61
A.3.1 Decryption . . . 62
A.3.2 Key Generation . . . 63
A.3.2.1 Finding w and d . . . 63
A.3.2.2 Finding r . . . 63
A.4 Summing the large message space z_i's . . . 63
A.5 A single grade-school addition for the recryption to bits . . . 64
A.6 Size of the coefficient g . . . 65
A.7 Specifications of evaluation machines . . . 66

B Proposal **67**

Chapter 1

Introduction

The aim of my project was to implement the most recently published fully homomorphic encryption scheme. In this chapter, I explain what homomorphic encryption is, why implementing it is an interesting and worthwhile effort, and give an overview of the state of the art on this area.

1.1 Motivation

Homomorphic encryption – encryption that supports operations on encrypted data – has a wide range of applications in cryptography. The concept was first introduced in 1978 by Rivest et al. shortly after the discovery of public key cryptography [1], and many popular cryptosystems, such as unpadded RSA or ElGamal, support either addition or multiplication of encrypted data. It was only in 2009 however, that Craig Gentry discovered the first plausible construction of a *fully homomorphic* encryption system supporting both operations [2].

A good metaphor for a homomorphic cryptosystem is the one of a jewelry shop [3]. Alice, the shop owner, has raw precious material – gold, diamonds, silver – that she wants her workers to assemble into intricately designed rings and necklaces. But she distrusts her workers, and is afraid that they will steal the materials if given the opportunity. In other words, she wants her workers to *process* the materials without having *access* to them.

To solve her problem, Alice designs a transparent, impenetrable glovebox for which only she has the key. She puts the raw materials inside the box, locks it, and gives it to a worker. Using the gloves, the worker can assemble the jewels inside the box, but since it is impenetrable, he can not get to the materials inside. Once finished, he hands the box back to Alice, who can now unlock it with her key and extract the ring or necklace.

Here, the materials represent the data that need to be processed, while the box represents the encryption of those data. What makes the box special, aside from being impenetrable like other secure cryptosystems, is the fact that it has gloves, allowing the processing of the data without accessing it.

So, in summary, fully homomorphic encryption allows arbitrary known algorithms to be run

on unknown encrypted data without decrypting it.

Since Gentry's paper in 2009, a lot of work has been done towards a working implementation of the scheme. The principal challenge comes from the fact that the key generation is computationally expensive, making a secure implementation almost impossible to achieve without carefully choosing a number of optimisations. The first attempt was made in 2010 by Smart and Vercauteren [4]. They managed to implement the general procedure, but were unable to execute it for keys of a secure size. This was later followed by another implementation by Gentry and Halevi [5], who built on from [4] and after multiple optimisations generated a nearly secure key.

Homomorphic encryption is currently one of the most active research topics in cryptography. Last February, the Defense Advanced Research Projects Agency (DARPA) awarded $20 million to a project on homomorphic encryption, almost $5 million of which will be spent on speeding up the homomorphic operation algorithms [6].

1.2 Challenges

In attempting this project, I faced a number of challenges. First, I had to study the necessary number theory, which is largely beyond the material taught in the tripos. As I knew I was only going to be able to study a limited range of topics in a one year project, I made the decision to focus on the engineering aspects of implementing the scheme and the understanding of how it works, rather than on the security part. This corresponds well with the aim of optimising previously published work.

Also, it meant familiarising myself with very high performance number theory libraries. The computations were done over integers of millions of bits and therefore had to be as efficient as possible.

A few weeks into the project, I discovered Gentry and Halevi had published the code of their implementation. I had already thoroughly analysed their implementation report, and had noticed various opportunities for optimisations. Hence, in this dissertation, I especially emphasise the improvements to the scheme rather than the re-implementation of existing code. I did, however, implement the entire scheme, and wrote all the code myself, often using approaches dissimilar to those taken by Gentry and Halevi. I only occasionally referred to their code for clarification, with the notable exception of incorporating one very specific function, which I clearly point out in the implementation chapter.

1.3 Overview of the project

The aim of the project was to produce an open implementation[1] of the most recently published scheme and optionally to explore some possible optimisations that could be done. The project was a complete success and a number of extensions were implemented. I built a different and simpler understanding of how homomorphic encryption works. From there, I extended the scheme from working on bits to working on modular integers, providing a

[1] At the time of writing the proposal, I was not aware of any implementation publicly available

tremendous speedup in some cases. I also describe a way to homomorphically convert between these two representations. Finally, I distributed the scheme to run on multiple CPUs or machines.

As far as I know, the set of techniques and algorithms I will describe in this dissertation make my implementation of homomorphic encryption faster than any other previously published.

Chapter 2

Preparation

In this chapter, I summarise the underlying theoretical constructs necessary to understand the fully homomorphic encryption scheme as published by Gentry. I then go over the design considerations I used in my implementation and briefly discuss the tools and software engineering techniques utilised.

2.1 Notations

As this project is heavily based on the work of Gentry and Halevi [5], I decided to use the same notations in order to allow the reader to refer to it if necessary. Given two integers c and d, I denote by $\langle c \rangle_d$ the reduction of c modulo d in the interval $[0, d)$ and by $[c]_d$ the reduction of c modulo d in the interval $[-d/2, d/2)$. For example, we have $\langle 13 \rangle_5 = 3$ and $[13]_5 = -2$.

For a rational number q, I denote by $\lceil q \rfloor$ the rounding of q to the next integer. For example $\lceil \frac{13}{5} \rfloor = 3$. By $[q]$, I denote the distance between q and the nearest integer, that is if $q = \frac{a}{b}$, then $[q] = \frac{[a]_b}{b}$. For example, $[\frac{13}{5}] = \frac{-2}{5}$.

These notations extend to vectors and matrices in the natural way, if $\vec{q} = \langle q_0, q_1, \ldots, q_{n-1} \rangle$ then $[\vec{q}]_d = \langle [q_0]_d, [q_1]_d, \ldots, [q_{n-1}]_d \rangle$.

2.2 Introduction to Lattice-based Cryptosystems

Before I move on to explaining the cryptosystems themselves, I will briefly recall the number theory structures they rely on.

2.2.1 Rings, Ideals and Ideal Lattices

2.2.1.1 Rings and Ideals

Here, I very briefly summarise the number theory structures of rings and ideals. If the reader has no prior background in this area, it is likely that appendix A.1 will provide a better

understanding.

A *ring* R is a set of elements closed under two binary operations $\cdot$ and $+$. An example of a ring is the set of all integers $\mathbb{Z}$ with the usual operations of addition and multiplication. Another example is the one of integer polynomials $\mathbb{Z}[x]$ which can be added to and multiplied with each other. In this project, I will often be working over the ring of polynomials modulo another polynomial $f(x)$, denoted as $\mathbb{Z}[x]/(f(x))$.

An *ideal* I of a ring R, is a subset of R which is itself a ring, such that for all r in R and a in I, $a \cdot r$ is itself in I. An example of an ideal is the set of even integers $(2\mathbb{Z})$, which is an ideal in the ring of integers.

I now move on to describing lattices, from which I will be able to define ideal lattices.

2.2.1.2 Lattices

Let $B = \{\vec{b}_1, \ldots, \vec{b}_n\}$ be a set of n linearly independent vectors in $\mathbb{R}^m$. The *lattice* generated by B is the set of all integer linear combinations of the vectors in B: $\mathcal{L}(B) = \{\sum_i x_i \vec{b}_i \mid x_i \in \mathbb{Z}\}$. The set B is called a *basis* and is usually associated with the matrix $\mathbf{B} \in \mathbb{R}^{n \times m}$, having the vectors $\vec{b}_i$ as rows. This gives a more handy definition:

$$\mathcal{L}(B) = \{\vec{x} \times \mathbf{B} : \vec{x} \in \mathbb{Z}^n\}.$$

A lattice is called *full-rank* if $n = m$. For simplicity, I will only consider full rank lattices. Furthermore, for computational purposes, I will assume the basis vectors $\vec{b}_i$ have integer entries ($\vec{b}_i \in \mathbb{Z}^n$).

An easy way to think of a lattice is a set of points in $\mathbb{Z}^n$, including the origin, that are closed under addition. For example, in $\mathbb{Z}^2$ the set of points shown in figure 2.1 form a lattice, and the set of blue and red arrows form two of the many base of the lattice.

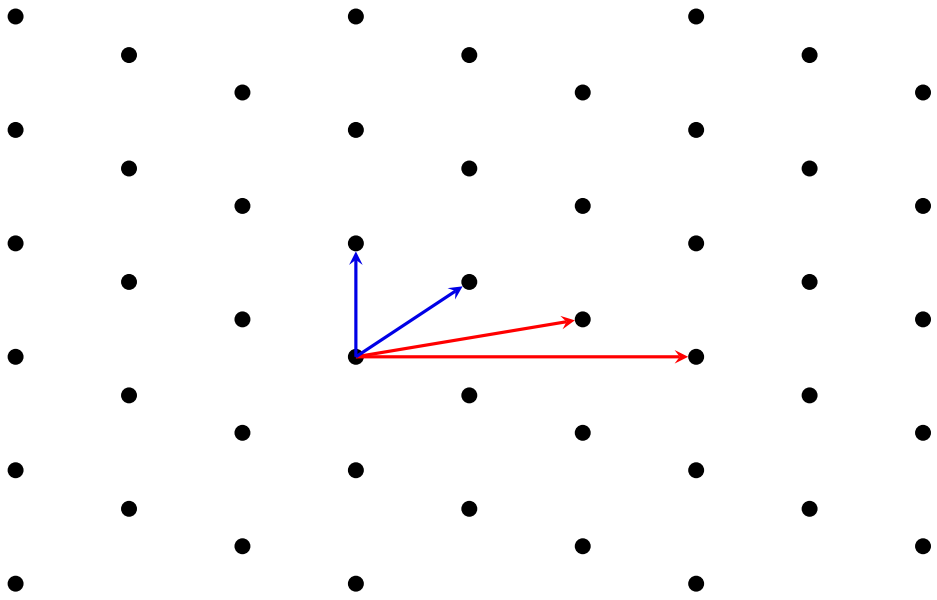

Figure 2.1: A lattice and two of its many bases.

Below, I briefly mention a number of useful facts about lattices.

For every lattice, there is an infinite number of bases. Furthermore, if B_1 and B_2 are two bases of the same lattice L, there is some unimodular matrix U (i.e. U has integer entries and $\det(U) = \pm 1$) such that $B_1 = U \times B_2$. This leads to our second fact.

Since we have $\det(B_1) = \det(U) \times \det(B_2) = \pm \det(B_2)$, $\mid \det(B_i) \mid$ is invariant for all the bases of L. I may therefore refer to it as $\det(L)$.

A particularly convenient basis of a lattice for some applications is the *Hermite Normal Form* (HNF). A basis B in HNF satisfies the following criteria:

- It is lower triangular ($b_{i,j} = 0$ for all $i < j$).
- All elements on the diagonal are strictly positive ($b_{i,i} > 0$ for all i).
- For all $i > j$ $b_{i,j} \in [-b_{j,j}/2, +b_{j,j}/2)$.

Every lattice has a unique basis in HNF; and given any basis B, the HNF basis of $\mathcal{L}(B)$ can be efficiently computed [7]. Given a basis B of a lattice L, the half-open parallelepiped $\mathcal{P}(B)$ is defined as $\mathcal{P}(B) \leftarrow \{\sum_{i=1}^{n} x_i \vec{b}_i \mid x_i \in [-1/2, 1/2)\}$. Note that the volume of $\mathcal{P}(B)$ is precisely $\det(L)$. For example, $\mathcal{P}(B)$ of the blue basis I showed above is the parallelogram drawn in figure 2.2.

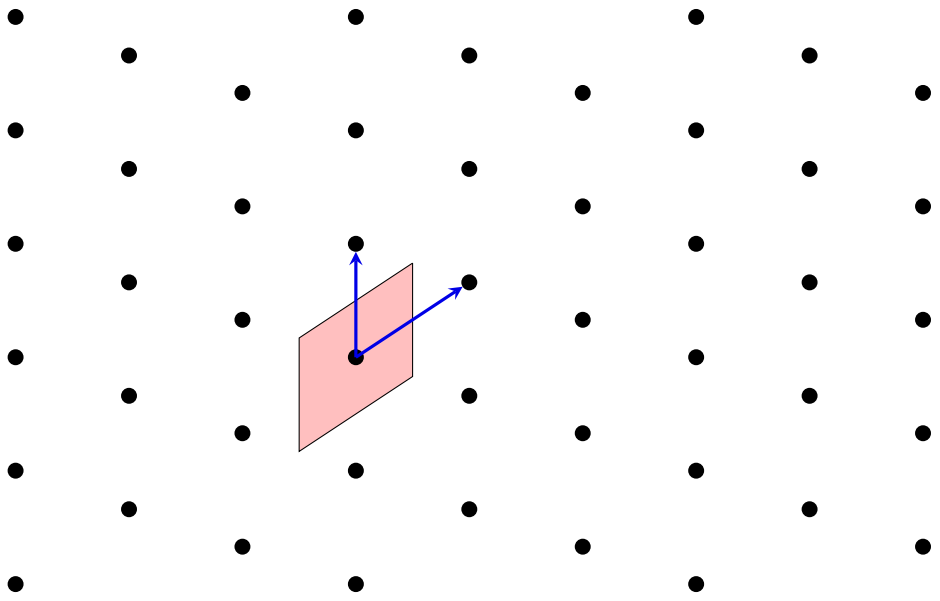

Figure 2.2: Parallelogram $\mathcal{P}(B)$ of the blue basis.

I now define operations modulo a basis; this will be very useful when using lattices for cryptosystems. Given $\vec{c} \in \mathbb{R}^n$ and a basis B of L, $\vec{c} \bmod B$ represents the unique vector $\vec{c'} \in \mathcal{P}(B)$ such that $\vec{c} - \vec{c'} \in L$. Executing a modulo operation can be done efficiently using the inverse B^{-1} of the matrix B.

2.2.1.3 Ideal Lattices

I now use the concepts of ideals and of lattices to define *ideal lattices*. Let $f(x)$ be an integer monic (highest order coefficient is 1), irreducible polynomial of degree n. For reasons that will become clear later, I will only use $f(x) = x^n + 1$, where n is a power of 2. I define R to be the ring of integer polynomials modulo $f(x)$, that is, $R \stackrel{\text{def}}{=} \mathbb{Z}[x]/(f(x))$. Elements of R are integer polynomials of degree $n-1$, it is therefore convenient to associate them with their coefficient vector in $\mathbb{Z}^n$. This way, every element of R can both be viewed as a polynomial and as a vector.

Let I be an ideal of the ring R. Recall that I is closed under addition, it is easy to show that – when considering the elements of I as vectors – the set of points in I form an integer lattice [8]. More formally, every ideal I is isomorphic to a lattice in $\mathbb{Z}^n$.

To emphasize the double nature of I both as an ideal and as a lattice, we call it an *ideal lattice*. To simplify matters, I will only use ideals with a single generator (also called *principal* ideals). Namely, given a single vector $\vec{v} \in R$, the principal ideal I generated by $\vec{v}$ is $I = (\vec{v}) = \{\vec{v} \times \vec{a} \mid \vec{a} \in R\}$ where $\times$ represents multiplication in the ring R (modulo $f(x)$). I then corresponds to the lattice generated by the vectors $\{\vec{v}_i \stackrel{\text{def}}{=} \vec{v} \times x^i \bmod f(x) \mid i \in [0, n-1]\}$; we call this the *rotation basis* of the ideal lattice I. As mentioned before, this is only one of the infinitely many bases of the lattice.

2.2.2 Lattice-based Cryptosystems and Homomorphic Encryption

In this section, I summarise how lattice encryption works, I then go over how it is used in Homomorphic Encryption.

2.2.3 Lattice-based cryptosystem

This section is a very brief summary of how lattice based cryptosystems, and in particular Goldreich–Goldwasser–Halevi (GGH)-type cryptosystems, work. Appendix A.2 includes a somewhat more in-depth explanation.

In a GGH-type cryptosystem, the private key consists of a "good" basis B of a lattice L, while the public key is a "bad" basis B' of the same lattice. The difference between the two bases is that the "good" one has vectors nearly orthogonal to one another while the "bad" ones vector tend to be close to parallel. To encrypt a message, one encodes it in a very small error vector $\vec{e}$ of same dimension as the lattice, and then adds it to a random point within the lattice, setting $\vec{c} = \vec{x}B' + \vec{e}$ where $\vec{x}$ is a random integer vector.

Consider the effect of reducing $\vec{c}$ modulo the "good" basis B. Figure 2.3 is a plot of two ciphertexts $\vec{c_1}$ and $\vec{c_2}$ with error vector being represented by the green and blue arrows respectively. Their reduction modulo B are $\vec{c_1'}$ and $\vec{c_2'}$.

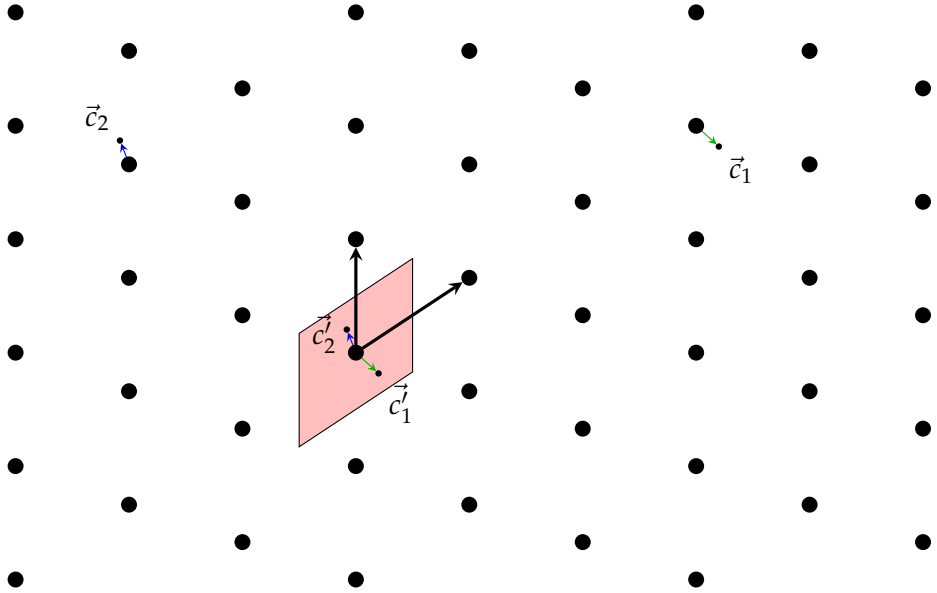

Figure 2.3: Reduction modulo a "good" basis B of $\vec{c}_1$ and $\vec{c}_2$.

Because $\vec{e}$ is short, it should be that the point of coordinates $\vec{e}$ is within $\mathcal{P}(B)$ and therefore $\vec{e}$ can be retrieved as $\vec{e} = \vec{c} \bmod B$. I will often refer to the maximum length $\vec{e}$ can have whilst being sure that $\vec{e} = \vec{c} \bmod B$ as the *decryption radius* of B. Consider what happens when trying to apply the same method with a "bad" basis B' as shown in figure 2.4.

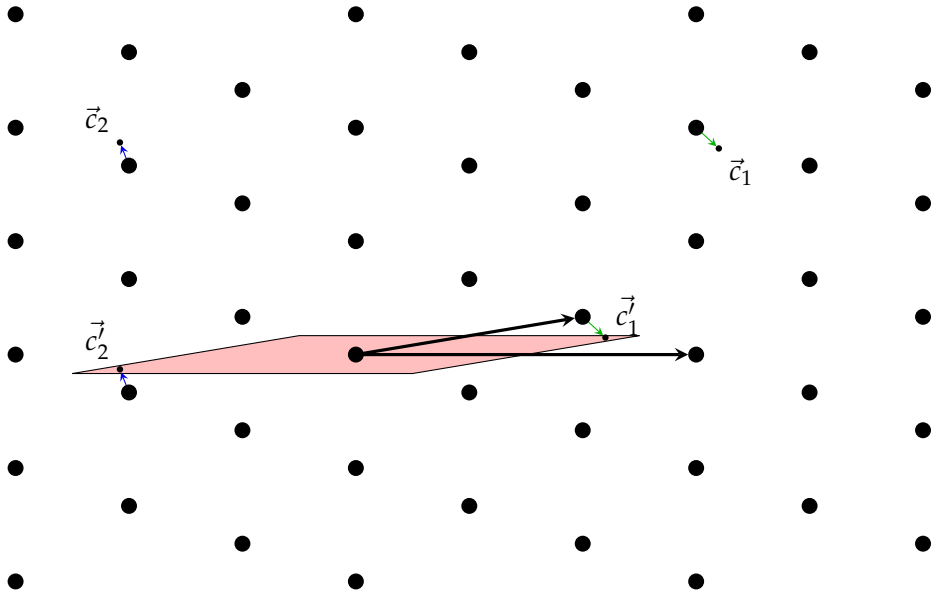

Figure 2.4: Reduction modulo a "bad" basis B′ of $\vec{c}_1$ and $\vec{c}_2$

Because the parallelogram $\mathcal{P}(B')$ is much thinner than $\mathcal{P}(B)$, its decryption radius is much lower than the one of B. It is therefore no longer the case that $\vec{e} = \vec{c} \bmod B$. In fact finding the error vector $\vec{e}$ using a "bad" basis turns out to be a hard problem [9, 10, 11], which is

the reason why lattice encryption is secure. I now move over to Gentry's Homomorphic cryptosystem which is based on lattices.

2.2.3.1 Gentry's Somewhat-Homomorphic Cryptosystem

This section gives an overview of Gentry's implementation of a somewhat-homomorphic cryptosystem [5], further details of which are also discussed in chapter 3. Gentry's cryptosystem is roughly made of two parts: a somewhat homomorphic cryptosystem that allows a limited number of operations to be performed on encrypted data, and a fully homomorphic system built on top which allows "homomorphic decryption" of a ciphertext. This homomorphic decryption is then used to perform arbitrary long computations on ciphertext as shown in [2]. In this section, I describe the somewhat homomorphic scheme.

The somewhat homomorphic scheme can be seen as a GGH-type scheme over ideal lattices. The reason ideal lattices are used instead of normal lattices is that in addition to the additively closed property of lattices, we also want our lattice to be multiplicatively closed so that both addition and multiplication can be done on our ciphertexts. We chose $f_n(x) = x^n + 1$ where n is a power of 2, and work over the ring $R = \mathbb{Z}[x]/(f_n(x))$. Concretely, this means that our ideal lattice, which I will call J, is going to be an ideal of the ring R. To generate J, we first choose a random vector $\vec{v}$ in some n-dimensional cube. As mentioned before, here $\vec{v}$ represents both a vector and the associated polynomial $v(x) = \sum_{i=0}^{n-1} v_i x^i$, I may therefore refer to it as $\vec{v}$ or $v(x)$. We then set $J = (\vec{v})$. Remember, this means $J = \{\vec{v} \times \vec{a} \mid \vec{a} \in R\}$ where $\times$ means multiplication in the ring R. Also, recall that the corresponding lattice has a basis – which I called rotation basis – defined as $\{\vec{v}_i \stackrel{\text{def}}{=} v(x) \times x^i \bmod f_n(x) \mid i \in [0, n-1]\}$.

Take $\vec{v}$ in our n-dimensional cube, consider the polynomial $v(x) \times x \bmod f_n(x)$

$$\begin{aligned} v(x) \times x &= (v_0 + v_1 x + \cdots + v_{n-1} x^{n-1})x \pmod{f_n(x)} \\ &= v_0 x + v_1 x^2 + \cdots + v_{n-1} x^n \pmod{f_n(x)} \\ &= v_0 x + v_1 x^2 + \cdots + v_{n-1} x^n - v_{n-1} f_n(x) \pmod{f_n(x)} \\ &= -v_{n-1} + v_0 x + \cdots + v_{n-2} x^{n-1} \pmod{f_n(x)} \end{aligned}$$

Repeating this for all powers of x, the rotation basis of our ideal lattice is therefore going to be:

$$V = \begin{bmatrix} v_0 & v_1 & v_2 & & v_{n-1} \\ -v_{n-1} & v_0 & v_1 & & v_{n-2} \\ -v_{n-2} & -v_{n-1} & v_0 & & v_{n-3} \\ & & & \ddots & \\ -v_1 & -v_2 & -v_3 & & v_0 \end{bmatrix}$$

This is going to serve as our private key. However, as I will show in the implementation section, there is some more useful representation of V that will be the actual one we use. Also, it can be shown that the Hermite Normal Form of the lattice $L = \mathcal{L}(V)$ is of the following form:

$$HNF = \begin{bmatrix} d & 0 & 0 & 0 & & 0 \\ -r & 1 & 0 & 0 & & 0 \\ -[r^2]_d & 0 & 1 & 0 & & 0 \\ -[r^3]_d & 0 & 0 & 1 & & 0 \\ & & & & \ddots & \\ -[r^{n-1}]_d & 0 & 0 & 0 & & 1 \end{bmatrix}$$

where d is the determinant of the lattice $d = \det(J) = \det(V)$, and r is a root of $f_n(x)$ modulo d: $r^n + 1 = 0 \pmod{d}$. You can see the HNF is as very "bad" basis: all the vectors are very close to being in the same direction. For this reason, it will be used as public key. Also, notice that its form is very handy, it can be implicitly represented using only the two integers d and r.

To encrypt a bit $b \in \{0, 1\}$ in this scheme, we choose a small random vector $\vec{r} \in \mathbb{Z}^n$ and set $\vec{e} = 2\vec{r} + b \cdot \vec{e}_1$. We then set $\vec{c} = \vec{e} \bmod B_{pk}$ [1]. The reason why we only encrypt a single bit will be explained in the implementation chapter.

To decrypt, as for the GGH cryptosystem, we reduce our ciphertext modulo our good basis $\vec{a} = \vec{c} \bmod V$. As previously mentioned, this should set $\vec{a}$ to be the distance away from the closest point in the lattice. An encrypted bit can then be recovered by reducing the coefficient along the first axis of $\vec{a}$ modulo 2. As I said, this procedure will be greatly simplified.

The reason why this scheme is homomorphic is the following: take two ciphertexts $\vec{c}_1 = \vec{j}_1 + \vec{e}_1$ and $\vec{c}_2 = \vec{j}_2 + \vec{e}_2$, where $\vec{j}_1$ and $\vec{j}_2$ are in our lattice J. Their sum is $\vec{c}_3 = \vec{j}_3 + \vec{e}_3$ where $\vec{j}_3 = \vec{j}_1 + \vec{j}_2$ is also in J, and $\vec{e}_3 = \vec{e}_1 + \vec{e}_2$. Since both $\vec{e}_1$ and $\vec{e}_2$ encode a bit along their first coefficient, the bit encoded by $\vec{e}_3$ will become the sum of these two bits modulo 2: the XOR of the two bits. The product of $\vec{c}_1$ and $\vec{c}_2$ is $\vec{c}_4 = \vec{j}_4 + \vec{e}_4$, where $\vec{j}_4 = \vec{j}_1 \times \vec{j}_2 + \vec{j}_1 \times \vec{e}_2 + \vec{j}_2 \times \vec{e}_1$ is also in J (this comes from the ideal properties of the lattice, $\vec{e}_1$ and $\vec{e}_2$ are in the ring R and therefore their product with an element of the ideal J is in J) and $\vec{e}_4 = \vec{e}_1 \times \vec{e}_2$. Again, since both $\vec{e}_1$ and $\vec{e}_2$ encode a bit along their first coefficient, the bit encoded by $\vec{e}_4$ will become the product of these two bits modulo 2: the AND of the two bits.

However, every time we do an operation on our encrypted data, the size of the error vector $\vec{e}$ increases. Therefore, as we do more and more operations, this vector is going to grow and eventually become larger than the decryption radius of our private key. Gentry solved this problem by using bootstrapping, which I will explain next.

2.2.3.2 Gentry's Fully-Homomorphic Encryption scheme

A scheme is called "fully homomorphic" if it can do an arbitrary number of computations. In [2], Gentry observed that a somewhat homomorphic scheme that can homomorphically execute the series of operations necessary to decrypt a ciphertext, plus one extra operation, can be turned into a fully homomorphic scheme.

To get a general idea of why this is, imagine we added an encryption c_{sk} of the secret key

[1] I said in section 2.2.3 that we needed to find a random point in the lattice, however, it was shown in [12] that reducing modulo the Hermite Normal Form has the same effect.

bits in our public key. In order to decrypt a ciphertext, one would usually execute a function on the secret key and the bits of the ciphertext: $\mathsf{Dec}(sk, c)$. Consider the result of applying this same function homomorphically on the bits of the ciphertext and the encrypted bits of the secret key: $\mathsf{Dec}(c_{sk}, c)$. Since we have the bits of c in the clear, we can just treat the 1s as the vector $\vec{e_1}$ and the 0's as $\vec{0}$ to be able to use the homomorphic operations I defined earlier. Call c' the output of $\mathsf{Dec}(c_{sk}, c)$. As long as the function Dec is of a complexity that can be handled by the somewhat homomorphic scheme, the value encrypted by c' will be whatever the output of $\mathsf{Dec}(sk, c)$ would have been: that is, the value encrypted by c. Notice how the size of the error vector in c' (I will often call this *degree* or *noise* of c') is independent of the degree of c. Only a fixed number of homomorphic operations were used in order to do this homomorphic decryption of c, and only this determines the degree of c'. It is clear that this can be used to reduce the error in a ciphertext. This leads to Gentry's definition of bootstrappability: consider the two functions

$$\mathsf{DAdd}(c_1, c_2, c_{sk}) \stackrel{\text{def}}{=} \mathsf{Dec}(c_{sk}, c_1) + \mathsf{Dec}(c_{sk}, c_2) \text{ and}$$
$$\mathsf{DMul}(c_1, c_2, c_{sk}) \stackrel{\text{def}}{=} \mathsf{Dec}(c_{sk}, c_1) \times \mathsf{Dec}(c_{sk}, c_2)$$

A somewhat homomorphic scheme is called bootstrappable if it is capable of homomorphically evaluating the functions DAdd and DMul for any two ciphertexts c_1 and c_2. It is easy to see that a boostrapable scheme can be fully homomorphic since by having c_{sk} in our public key, we can use the functions DAdd and DMul an arbitrary number of times on our encrypted data.

Unfortunately, the somewhat homomorphic scheme I described above is not bootstrappable. The degree of the decryption function is too high to be computed homomorphically. To solve this problem, Gentry showed how to "squash" the decryption procedure to make its complexity acceptable. In the somewhat homomorphic scheme, the secret key is (as we will see in the implementation chapter) an integer w. Gentry added to the public key a big set of integers $\mathcal{S} = \{x_i \mid i \in \{1, \ldots, S\}\}$ such that a hidden sparse subset of $\mathcal{S}$ adds up to w. He also added to the public key the new equivalent of the encryption of the secret key: a vector of encrypted bits $\vec{\sigma} = \langle \sigma_1, \sigma_2, \ldots, \sigma_S \rangle$ corresponding to the sparse subset of $\mathcal{S}$. This means that we have $w = \sum_{i=1}^{S} \sigma_i \times \vec{x_i}$. This is extremely useful as it means that most of the decryption procedure can be computed in the clear using the integers x_i, and once this is done we can use $\vec{\sigma}$ to finish the procedure with only a low homomorphic computation complexity.

2.3 Choice of Tools

2.3.1 Programming Language and Libraries

The computational complexity of the project implied from the very beginning that it would have to rely on fast libraries. With this is mind, I decided to let library availability determine the implementation language. Two libraries were needed:

1. A number theory library: to represent large integers and to achieve good performance on them.

2. An I/O library: one of my proposed extensions was to distribute the scheme over several machines to improve performance. It was therefore necessary to have a platform to enable communications between parallel execution contexts.

For the first choice, I decided to opt for Victor Shoup's NTL library [13]. It is regarded as one of the fastest number theory libraries available and has been used to successfully break some cryptosystems. This meant having to use C++ as the implementation language. I did not have much experience programming in C++, but it seemed like a necessary choice to produce an efficient implementation.

There are a surprisingly low number of portable I/O libraries available for C++. After some research, I decided to use Boost Asio [14]. The Boost collection of libraries is widely renowned for its efficiency and portability, and the Asio library enabled asynchronous I/O operations which were needed for the parallelization. The poor quality of the documentation however later turned out to make it more difficult to use than anticipated.

2.3.2 Development Environment

The majority of the development was carried on my personal machine running Windows 7, although additional resources provided on the PWF and by the SRCF were used for backups.

All of the code was developed on my personal machine using Microsoft Visual Studio 2008 Professional Edition. Visual Studio is a very powerful IDE that combines easy development with fast compiled code. It also provides a very useful debugger, allowing single-step debugging and even compiling code at run time.

To back up the code of my project, I decided to use the Subversion 1.6.9 source control management system. This allowed me to go back to a previous revision of a file whenever necessary. Subversion was also used to hold the LaTeX source of this dissertation and the proposal as they were being written.

The project was backed up in a way that did not rely on any single component. The Subversion repository was held on the SRCF server and automatically copied to the PWF file space once a day using a shell script.

2.4 Software Engineering Techniques

The development methodology was chosen to match the nature of the project. Most of the procedures were hard to implement, but usually easy to test for correctness. I therefore decided to adopt a *Waterfall Development Model* along with formalised unit-testing methodologies. This way, every function was first implemented and then tested for correctness individually (see section 4.1.1). Some regression testing was also used as I was adding more and more functionalities to the implementation.

Chapter 3

Implementation

This chapter describes the implementation of homomorphic encryption and of the extensions mentioned in the introduction. I first describe my implementation of the fully homomorphic system as described in [5]. From this, and using some results from the evaluation, I outline a different and hopefully easier to understand explanation of why homomorphic encryption works. I then use this new understanding to increase the message space of the scheme, allowing me to encrypt modular integers instead of individual bits. Finally, I show how I have distributed the scheme, allowing it to take advantage of parallel processing on multiple CPUs or machines.

Section 3.1 represents prior work, although some differences with my implementation are noted. Sections 3.2 - 3.6 are new contributions beyond the state of art in homomorphic encryption.

3.1 Implementing Gentry's scheme

In this section, I describe the implementation of the scheme as it was done by Gentry and Halevi in [5]. The success criterion for the project was to reimplement this scheme successfully. Most of the mathematical arguments made here are prior work and from this perspective could have equally been in the preparation chapter. I start this chapter with the implementation of the encryption and the decryption. I mentioned in the preparation that there would be some optimisations leading to a slightly different representation of the secret key than expected; these come directly from the observations of how the encryption and description work.

3.1.1 Encryption

In this subsection, I describe how to encrypt a bit $b \in \{0, 1\}$, with a public key B, where B is in HNF and therefore can be implicitly represented by the two integers d and r. First, I choose a random noise vector $\vec{u} = \langle u_0, u_1, \ldots, u_{n-1} \rangle$. Here, my procedure differs somewhat from the one used in [5]: I randomly pick a small fixed number ε of entries in $\vec{u}$ which I randomly set to 1 or -1 and set all other entries to 0. In [5], each entry had a probability ε/n to be picked

to be ± 1. The reason for this change is that, even though we loose some entropy, meaning we will need a slightly higher ε, this lets us keep better track of the Euclidean length of our encrypted data, especially after a large number of multiplications. In my implementation, I systematically use $\varepsilon = 15$, which is secure if n is large enough. Once this is done, set $\vec{a} = 2\vec{u} + b\vec{e_1} = \langle 2u_0 + b, 2u_1, \ldots, 2u_{n-1} \rangle$, and the ciphertext is the vector:

$$\vec{c} = \vec{a} \bmod B = [\vec{a} \times B^{-1}] \times B$$

Recall from section 2.1, that the notation $[q]$ corresponds to the fractional part of q. As I said before, B is of the form:

$$B = \begin{bmatrix} d & 0 & 0 & 0 & & 0 \\ -r & 1 & 0 & 0 & & 0 \\ -[r^2]_d & 0 & 1 & 0 & & 0 \\ -[r^3]_d & 0 & 0 & 1 & & 0 \\ & & & & \ddots & \\ -[r^{n-1}]_d & 0 & 0 & 0 & & 1 \end{bmatrix}.$$

Note that this also means that B^{-1} is of the following form:

$$B^{-1} = \frac{1}{d} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & & 0 \\ r & d & 0 & 0 & & 0 \\ [r^2]_d & 0 & d & 0 & & 0 \\ [r^3]_d & 0 & 0 & d & & 0 \\ & & & & \ddots & \\ [r^{n-1}]_d & 0 & 0 & 0 & & d \end{bmatrix}.$$

I now show that the vector $\vec{c}$ can be represented by a single integer. Denote $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, and the corresponding integer polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$. We have

$$\begin{aligned} \vec{c} &= [\vec{a} \times B^{-1}] \times B \\ &= [\langle \frac{s}{d}, a_1, \ldots, a_{n-1} \rangle] \times B \end{aligned}$$

with $s = \sum_{i=0}^{n-1} a_i [r^i]_d$. Notice that we have $s = a(r) \pmod{d}$. It follows that:

$$\begin{aligned} \vec{c} &= \langle [\frac{s}{d}], [a_1], \ldots, [a_{n-1}] \rangle \times B \\ &= \langle \frac{[a(r)]_d}{d}, 0, \ldots, 0 \rangle \times B \\ &= \langle [a(r)]_d, 0, \ldots, 0 \rangle. \end{aligned}$$

It is clear that this vector can be represented by the integer $c = [a(r)]_d = [b + 2\sum_{i=0}^{n-1} u_i r^i]_d$. Note this means we must use a d that is odd otherwise $b = c \pmod 2$ In the following subsection, I show how this integer can be decrypted.

3.1.2 Decryption

In this subsection, I describe how to decrypt a vector $\vec{c} = \langle c, 0, \ldots, 0 \rangle$ that has been encrypted as described in the previous subsection. Recall that the decryption procedure consists of doing:

$$\vec{a} = \vec{c} \bmod V = [\vec{c} \times V^{-1}] \times V.$$

We can then recover the bit encrypted by $\vec{c}$ by using the fact that $\vec{a} = 2\vec{u} + b \cdot \vec{e_1}$. As I mentioned before, V is of the form:

$$V = \begin{bmatrix} v_0 & v_1 & v_2 & & v_{n-1} \\ -v_{n-1} & v_0 & v_1 & & v_{n-2} \\ -v_{n-2} & -v_{n-1} & v_0 & & v_{n-3} \\ & & & \ddots & \\ -v_1 & -v_2 & -v_3 & & v_0 \end{bmatrix}.$$

As it turns out, the inverse of V is a matrix $V^{-1} = \frac{1}{d} \cdot W$ such that

$$W = \begin{bmatrix} w_0 & w_1 & w_2 & & w_{n-1} \\ -w_{n-1} & w_0 & w_1 & & w_{n-2} \\ -w_{n-2} & -w_{n-1} & w_0 & & w_{n-3} \\ & & & \ddots & \\ -w_1 & -w_2 & -w_3 & & w_0 \end{bmatrix} \tag{3.1}$$

which I demonstrate in appendix A.3.2.1. This means we have:

$$\begin{aligned} \vec{a} &= \vec{c} \bmod V \\ &= [\vec{c} \times W/d] \times V. \end{aligned}$$

I demonstrate in appendix A.3.1 that using the fact $\vec{a} = 2\vec{u} + b \cdot \vec{e_1}$, this implies

$$[cw_i]_d = b \cdot w_i \pmod 2.$$

In order to decrypt c, it is therefore sufficient to keep only one odd entry w_i of W, and recover the bit by setting $b = [cw_i]_d \pmod 2$. As one can see, this decryption is much more efficient than the original one, which is the reason why I encrypt a single bit in a ciphertext rather than a message $m \in \{0,1\}^n$.

I will now describe how to generate this entry w_i along with the public key.

3.1.3 Key Generation

In this section, I describe how to generate a public key implicitly represented by the integers d and r, along with a private key implicitly represented by one odd entry w_i of the matrix

W defined above. To do this, we first need to randomly generate an ideal lattice in the ring $R = \mathbb{Z}[x]/(f_n(x))$ where $f_n(x) = x_n + 1$. As mentioned in section 2.2.3.1, this is done by selecting a random vector $\vec{v}$ in an n-dimensional cube. The lattice is then the one generated by the rotation basis V of $\vec{v}$: $\mathcal{L}(V)$. Therefore, there are two parameters for the key generation:

1. The dimension n of our ideal lattice.
2. The bits size t of the coefficients of $\vec{v}$, where each coefficient v_i is taken uniform randomly from the range $-2^{t-1} \leq v_i < 2^{t-1}$.

As usual, I may refer to $\vec{v}$ as the polynomial $v(x)$. Consider the scaled inverse of $v(x)$, that is, an integer polynomial $w(x)$ of degree at most $n-1$ such that

$$w(x) \times v(x) = d' \pmod{f_n(x)} \tag{3.2}$$

where d' is a constant. In appendix A.3.2.1, I show not only that $d' = d$, the determinant of V, but also the rotation basis of $w(\cdot)$ is the matrix W I defined in equation 3.1. Applying the extended Euclidean algorithm (XGCD) on the polynomials $v(x)$ and $f_n(x)$ will give us both $w(x)$ and d. This is because

$$w(x) \times v(x) + z(x) \times f_n(x) = d$$

for some integer polynomial $z(x)$. Once we have $w(x)$ and d, we can check that $v(x)$ is a good generating polynomial. First of all, $w(x)$ must have an odd coefficient to use as the secret key. This extremely likely to be the case for large dimensions. Another requirement I mentioned above is that d must be odd. Finally, we need to generate r, the second integer in the public key. I show in appendix A.3.2.2 that it must be the case that

$$\langle -w_{n-1}, w_0, \ldots, w_{n-2} \rangle = r \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle \pmod{d}.$$

This leads to the important property that $w_i = r \cdot w_{i+1} \pmod{d}$ and $w_{n-1} = -r \cdot w_0 \pmod{d}$. Therefore, given two consecutive coefficients of $w(x)$(which we have), r can be calculated as $r = w_{i+1}/w_i \pmod{d}$ which is just an instance of the GCD.

I said above, I use the extended Euclidean algorithm to compute $w(x)$ from $v(x)$. However, in [5], an extremely efficient procedure to do this computation is described, taking advantage of the fact that $f_n(x) = x^n + 1$ where n is a power of two. As this function is already very optimised, I did not feel that I would be able to improve it and decided to use the XGCD algorithm. For the evaluation however, and once the success criteria had been met, I adapted Gentry's and Halevi's code for this particular function to work with my implementation in order to give an accurate idea of the complexity of homomorphic encryption.

3.1.4 Extension to a Fully Homomorphic scheme

In the first three sections of this chapter, I covered how to implement the encryption, decryption and key generation of Gentry's somewhat homomorphic scheme. This already supports a limited number of homomorphic computations. As previously mentioned, this is simply

done by adding or multiplying the vectors corresponding to the ciphertexts. Now that these vectors are represented by a single integer, our homomorphic operations simply are:

$$c_1 \,\mathsf{XOR}\, c_2 = [c_1 + c_2]_d$$
$$c_1 \,\mathsf{AND}\, c_2 = [c_1 \times c_2]_d.$$

The reduction of the results modulo d is only for convenience. Notice that this concretely means we are keeping our lattice points in the parallelepiped of the HNF.

I now explain how we can turn this scheme into a fully homomorphic one. As I mentioned in the preparation, this is done by homomorphically decrypting our ciphertext using an encryption of the secret key that we have put in the public key. I will refer to this operation as a *recryption* since it lets us recrypt a ciphertext with a possibly large error vector into a fresh ciphertext. The optimised decryption procedure is very useful for this; it means that the recryption is going to be optimised too and therefore the error vector as we finish recrypting will not be too large.

So far, decrypting a ciphertext c using the secret key w is done by setting $b = [wc]_d \bmod 2$. As I mentioned before, this operation is too complex to be done as such using the somewhat homomorphic scheme. To solve this problem, we add in the public key a "hint" about w. In section 2.2.3.2, I explained this in terms of adding a single big set of elements, which was a slight simplification.

Given the two parameters s and S (in the implementation, I systematically use $s = 15$, S is a big number), we add to the public key s integers $x_1, \ldots, x_s$, all in $\mathbb{Z}_d$. Each of these integers x_k will represent a big set $\mathcal{B}_k$ constituting of the S integers $\{\langle x_k \cdot R^i \rangle_d \mid i \in \{0, \ldots, S-1\}\}$ where R is a parameter of the scheme. I will refer to the elements of $\mathcal{B}_k$ as $\{x_k(i) \mid i \in \{0, \ldots, S-1\}\}$. As I said, we want these big sets to be a hint about w. To do this, we choose our integers $x_1, \ldots, x_s$ so that there is a single index i_k for every big set such that $\sum_{k=1}^{s} x_k(i_k) = w \pmod d$. This means in particular $[cw]_d = [\sum c x_k(i_k)]_d$

We now want a new secret key associated with this hint that we will be able to encrypt and use to do the recryption. We define the following new bit vectors $\vec{\sigma}_1, \ldots, \vec{\sigma}_s$, each of size S such that

$$\sigma_k(i) = \begin{cases} 1 & \text{if } i = i_k \\ 0 & \text{otherwise} \end{cases}$$

Each of these bits can then be individually encrypted and included in the public key. As it turns out, I do not encrypt exactly these bits, but a compression of these bits. I will come back to this later. For simplicity, I will refer to $\sigma_k(i)$ as $\sigma_{k,i}$. Next, I consider how we are going to implement the recryption.

We can use the hint to precompute most of the decryption in the clear. Given a ciphertext c, we calculate all the integers $y_{i,j} = \langle c x_i(j) \rangle_d$ (recall $\langle \rangle_d$ means modulo d in the interval $[0, d)$).

The decryption function can then be written as

$$\begin{aligned}\mathsf{Dec}(c) &= \left[\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}y_{i,j}\right]_d \bmod 2 \\ &= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}y_{i,j}\right) - d\cdot\left\lceil\frac{\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}y_{i,j}}{d}\right\rceil \bmod 2 \\ &= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}y_{i,j}\right) - d\cdot\left\lceil\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\frac{y_{i,j}}{d}\right\rceil \bmod 2.\end{aligned}$$

Note that the quotients $y_{i,j}/d$ can also be calculated in the clear and that their value is a rational number in $[0,1)$. Given a precision parameter ρ, consider the approximation $z_{i,j}$ of each quotient $y_{i,j}/d$ within ρ bits after the binary point. It is shown in [5] that if we keep the distance from c to the lattice below $1/(s+1)$, then using $\rho = \lceil\log_2(s+1)\rceil$ will make $\lceil\sum_{i=1}^{S}\sigma_{i,j}z_{i,j}\rfloor$ yield the same result as $\lceil\sum_{i=1}^{S}\sigma_i\frac{y_{i,j}}{d}\rfloor$.

The low bit size of the $z_{i,j}$'s is going to be very useful. Multiplying an encrypted bit $\sigma_{i,j}$ with $z_{i,j}$ is going to give a vector of encrypted bits of size $\rho+1$ (recall $z_{i,j}$ is made of ρ bits after the binary point and one before). The value represented by any element of this vector is just going to be 0 when the corresponding bit of $z_{i,j}$ is 0, and $\sigma_{i,j}$ when it is 1. We are now very close to having a working recryption procedure. Using the modulo 2 in the formula above, we can turn the decryption into the following bit computation

$$\mathsf{Dec}(c) = \left(\bigoplus_{i=1}^{s}\bigoplus_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_2\right) \oplus \langle d\rangle_2 \cdot \left\langle\left\lceil\sum_{i=1}^{s}\bigoplus_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rfloor\right\rangle_2$$

where the XORing of the $\sigma_{i,j}z_{i,j}$ just means the bitwise XOR of the encrypted bit vector they generate. Note that this works because only one $\sigma_{i,j}$ per big set vector $\vec{\sigma}_i$ is non-zero. The last step we need to take to make this function work is to add the s bit vectors generated by $\oplus_{j=0}^{S-1}\sigma_{i,j}z_{i,j}$ and round the result to the nearest integer. To sum the bit vectors, we are going to have to implement a homomorphic version of Grade-School Addition(GSA). Note that since we will round the result and reduce it modulo 2, all we need to do is find the bit before and the bit after the decimal point and XOR them together. Below I describe how to perform GSA.

First, let me define what an elementary symmetric polynomial is. Given n variables $X_1,\ldots,X_n$, the elementary symmetric polynomial of degree k denoted $e_k(X_1,\ldots,X_n)$ is the sum of all the products of k-tuples of the n variables. For example,

$$\begin{aligned}e_2(X_1,\ldots,X_n) &= \sum_{1\le i<j\le n} X_iX_j \\ e_3(X_1,\ldots,X_n) &= \sum_{1\le i<j<k\le n} X_iX_jX_k.\end{aligned}$$

When using this for bits, an addition will be represented by a XOR and a multiplication by an AND, as usual. Recall that we need to do the addition of s numbers each with $\rho+1$

bits of precision. We arrange the bits in s rows and $\rho + 1$ columns and index the columns $0, -1, \ldots, -\rho$; column 0 contains the bits to the left of the binary point, column -1 contains the first bit to the right of the binary point and so on. For each column, we keep a stack containing the bits of this column. We then proceed from column $-\rho$ up to column 0, each time homomorphically computing the carry bits that this column sends to the columns on the left and pushing these carry bits on the corresponding stack. The computation of these carry bits is simple: the carry bit that a column $-j$ sends to a column $-j + \Delta$ is the elementary symmetric polynomial of degree 2^Δ in the bits of column $-j$. Note that this computation of the elementary symmetric polynomials can be batched, if a column has m bits and needs to evaluate the elementary symmetric polynomials of degree up to 2^Δ, this can be done in $m2^\Delta$ multiplications. Of course, the result bit of a column is just the XORing of the bits in that column.

Finally, I am briefly going to discuss what I referred to earlier as the compression of the secret key bits. There is a clear tradeoff between space and computation when encrypting the secret key. All that a vector $\vec{\sigma}_j$ encodes is one index $i_j \in \{0, \ldots, S-1\}$. This could be represented in $\log_2 S$ bits, but this would require much more computation when doing a recryption. The tradeoff I decided to take – which is the same one that was done in [5] – is to have a depth of one in the computation of the $\sigma_{i,j}$s. Namely, for every big set $\mathcal{B}_k$, we encrypt a bit vector $\vec{\eta}_k = \{\eta_{k,i} \mid i \in \{1, \ldots, q\}\}$, where q is an integer greater than $\lceil \sqrt{2S} \rceil$ and all but two of the bits in $\vec{\eta}_k$ are 0. A secret key bit $\sigma_{k,i}$ is then retrieved by multiplying two of these ciphertexts. To match a pair of ciphers to an index, we consider pairs in the lexicographical order over pairs of distinct numbers in $\{1, \ldots, q\}$:

$$ind(a,b) = (a-1) \cdot q - \binom{a}{2} + (b-a). \tag{3.3}$$

In particular, if $\eta_{k,a}$ and $\eta_{k,b}$ are the two encryptions of 1 in $\vec{\eta}_k$, then $i(a,b) = i_k$.

I can now summarise the final function used for the recryption. Note that I removed $\langle d \rangle_2$ since d is odd.

$$\begin{aligned}
\mathsf{Dec}(c) &= \left(\bigoplus_{i=1}^{s} \bigoplus_{a,b} \eta_{i,a}\, \eta_{i,b}\, \langle y_{i,ind(a,b)} \rangle_2 \right) \oplus \left\langle \left\lfloor \sum_{i=1}^{s} \bigoplus_{a,b} \eta_{i,a}\, \eta_{i,b}\, z_{i,ind(a,b)} \right\rfloor \right\rangle_2 \\
&= \left(\bigoplus_{i=1}^{s} \bigoplus_{a} \eta_{i,a} \bigoplus_{b} \eta_{i,b}\, \langle y_{i,ind(a,b)} \rangle_2 \right) \oplus \left\langle \left\lfloor \sum_{i=1}^{s} \bigoplus_{a} \eta_{i,a} \bigoplus_{b} \eta_{i,b}\, z_{i,ind(a,b)} \right\rfloor \right\rangle_2 .
\end{aligned}$$

This method of having a depth of one computation for the $\sigma_{i,j}$s is systematically used in my implementation. However, I will keep on referring to them as $\sigma_{i,j}$ for simplicity.

3.1.5 An optimised encryption procedure

I finish this section about the implementation in [5] by noting the only real difference to my implementation thus far. Recall that the encryption of a bit b is $c = [b + 2\sum_{i=0}^{n-1} u_i r^i]_d$. Therefore we only need to evaluate the 0-1 polynomial $u(\cdot)$ at r, multiply the result by two and add b. It is clear that the most computationally expensive operation is the evaluation of $[u(r)]_d$, and more specifically, the multiplications done when evaluating $[u(r)]_d$.

In [5], the authors use a technique described in [15] to batch the computation of k polynomials and compute it in $O(\sqrt{k})$ time. The technique basically consists of evaluating k polynomials of degree $2n-1$ (recall our dimension is a power of 2) by splitting each polynomial into two polynomials of degree n; the one on the left representing the powers from 0 to $n-1$ and the one on the right representing the powers from n to $2n-1$. Once both sides of each polynomials are evaluated, the resulting polynomials can be evaluated in k multiplications by multiplying each of the right polynomials by $[r^n]_d$ and adding them to their corresponding left polynomial. The function then recurses until the number of multiplications necessary to recurse one more time is greater than the number of multiplications to compute every power of r between 0 and (the current) $n-1$. Then, it makes a temporary variable r_{tmp}, and iterates over every power of r from 0 to $n-1$, each time adding or subtracting r_{tmp} from the polynomials that have the corresponding coefficient as 1 or -1.

The method I use slightly differs from this approach: it calculates each individual polynomial the same way, but without computing all of them at the same time. First, I calculate the number of times the method above recurses. It is the smallest integer s such that $n/2^{s+1} < 2^s k + 1$, which is $s = \left\lceil \frac{1}{2} \log_2(\frac{n}{2k}) \right\rceil$. Then, instead of evaluating all the polynomials simultaneously, I calculate all the powers of r between 0 and $n/2^s - 1$ (which would have been $n-1$ on the bottom level) and keep the results in an array. I also calculate every r^{2^i} up to $2^i = n/2$ and also store them in an array. This way, I can calculate each polynomial individually the same way it would have been calculated as part of a batch. This modification has the following advantages:

- Since the bits do not need to be evaluated at the same time, I can take advantage of the $O(\sqrt{k})$ for bits encrypted later on. For example, the key generation of the fully homomorphic scheme requires the encryption of a large number of bits, I can store the powers of r to speed up the encryption of bits in the future.
- This method is much more parallelizable. I will study parallelization of the scheme in more details in section 3.6.3.
- This method can compute every even power of r using modulo square rather than modular multiplication which is usually about 20% faster.

This approach can have a larger or smaller memory footprint depending on n and k. On average, its memory footprint is slightly worse than with the original methods.

3.2 A new understanding of the somewhat homomorphic scheme

In section 3.1, I covered my implementation of the existing approaches. The rest of this dissertation consists of new contribution. I will now discuss a different perspective on how the somewhat homomorphic scheme works, which I will substantiate in the evaluation chapter (see section 4.2.1).

Recall that the public key is made of the two integers d and r, and the private key of the integer w, which is an odd coefficient of the polynomial $w(\cdot)$. I have shown earlier that the coefficients w_i of $w(\cdot)$ satisfy $[r \cdot w_{i+1}]_d = w_i$ and $w_{n-1} = -[r \cdot w_0]_d$. Note that this also means

that $[r^n]_d = -1$. For simplicity below I will assume the odd coefficient we chose for w was w_0, but this should not have any impact. In the evaluation (see section 4.2.1), I will show that the size of the coefficients w_i is about 2^t times smaller than d. Recall that t is the bit size that we used for our generating polynomial $v(\cdot)$. This result gives us a great insight to how the scheme works.

When I encrypt a bit using $c = [b + 2\sum_{i=0}^{n-1} u_i r^i]_d$, the result I get is an integer c evenly distributed in the range $[-d/2, d/2)$. The reason for this is that the powers of r modulo d are themselves in the range $[-d/2, d/2)$. However, once I multiply my ciphertext by w and reduce it modulo d, I get

$$\begin{aligned}[cw]_d &= [bw + 2\sum_{i=0}^{n-1} u_i r^i w]_d \\ &= [bw + 2\sum_{i=1}^{n} u_{i-1} w_{n-i}]_d.\end{aligned}$$

But since the values of the w_i's are much smaller than d, it should be the case that $[bw + 2\sum_{i=0}^{n-1} u_i w_i]_d = bw + 2\sum_{i=0}^{n-1} u_i w_i$. Reducing this modulo 2 should therefore give:

$$\begin{aligned}[cw]_d \bmod 2 &= bw + 2\sum_{i=0}^{n-1} u_{i-1} w_{n-i} \bmod 2 \\ &= bw \bmod 2 \\ &= b\end{aligned}$$

using the fact that I picked w to be odd. It can also be seen that adding two ciphertexts, once decrypted will give the following

$$\begin{aligned}[(c_1 + c_2)w]_d \bmod 2 &= (b_1 + b_2)w + 2\sum_{i=0}^{n-1} u'_{i-1} w_{n-i} \bmod 2 \\ &= b_1 + b_2 \bmod 2\end{aligned}$$

and multiplying two ciphertexts will give

$$\begin{aligned}[(c_1 \cdot c_2)w]_d \bmod 2 &= (b_1 \cdot b_2)w + 2\sum_{i=0}^{n-1} u''_{i-1} w_{n-i} \bmod 2 \\ &= b_1 \cdot b_2 \bmod 2\end{aligned}$$

where $u'(\cdot)$ and $u''(\cdot)$ are some small integer polynomials. This explains the homomorphism of the scheme. However, as we keep on adding and multiplying the ciphertexts, the coefficients of the integer polynomials $u'(\cdot)$ and $u''(\cdot)$ grow in size. After a number of operations, it will become the case that $bw + 2\sum_{i=0}^{n-1} u_i w_i > d/2$. At that point, $[bw + 2\sum_{i=0}^{n-1} u_i w_i]_d = bw + 2\sum_{i=0}^{n-1} u_i w_i$ will no longer be true and therefore the decryption cease to work. This is why only a limited number of homomorphic operations can be done on a ciphertext.

This new approach makes the scheme more easily understandable. It also yields the question why we limit our operations to modulo 2. In the next section, I adapt the scheme to work modulo any integer p and describe the necessary changes to the recryption function to make this a fully homomorphic scheme.

3.3 Increasing the message space

In this section, I describe how to encrypt fully homomorphic integers in the message space $\{0, \ldots, p-1\}$. From the results of the previous section, it is clear we can adapt the encryption and decryption of the somewhat homomorphic scheme. Namely, using a w such that $w = 1 \pmod{p}$, they can be turned into the following:

$$\begin{aligned} \mathsf{Enc}(m, p) &= [m + p \sum_{i=0}^{n-1} u_i r^i]_d \\ \mathsf{Dec}(c, p) &= [cw]_d \bmod p. \end{aligned}$$

Further, we can still use the homomorphism of the scheme. If c_1 and c_2 are the encryptions of the messages m_1 and m_2, then $c_3 = [c_1 + c_2]_d$ should be the encryption of the message $m_3 = \langle m_1 + m_2 \rangle_p$ and $c_4 = [c_1 \cdot c_2]_d$ the encryption of $m_4 = \langle m_1 \cdot m_2 \rangle_p$. One important fact to notice is that as we increase p, we also increase the size of the coefficients in the polynomial I referred to as $u(\cdot)$ before. The consequence is that for a fixed bit size of the generating polynomial t, the maximum number of operations that can be done on a given ciphertext is going to decrease as we increase p.

I now turn to implementing the recryption function in such a way that it can be used for integers in this larger message space. First, we need to decide how to encrypt the secret key. It seems tempting to use the new larger message space to condense the encryption of w into fewer ciphertexts. However, as far as I can see, there is no good way of achieving this. The problem is that with the current recryption procedure, the equations contain computations of the form $\sigma \langle xc \rangle_d$. Ideally, we would encode a part of the x into the σ, taking more out of the "hint" and into the secret key. But since $\sigma \langle xc \rangle_d \neq \langle \sigma xc \rangle_d$, I could not find a good way of achieving this. Therefore, we keep the exact same representation of the secret key, with the only difference that instead of encrypting the 0's and 1's modulo 2, we encrypt them modulo p.

Now consider the actual decryption function. Recall that before we simplified the case to bits in section 3.1.4, I had derived the formula for the decryption as (with 2 replaced by p):

$$\mathsf{Dec}(c) = \left(\left(\sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} y_{i,j} \right) - d \cdot \left\lceil \sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} \frac{y_{i,j}}{d} \right\rfloor \right) \bmod p.$$

Again, we can apply the mod p on both sides:

$$\mathsf{Dec}(c) = \left(\sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} \langle y_{i,j} \rangle_p \right) - \langle d \rangle_p \cdot \left\langle \left\lceil \sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} \frac{y_{i,j}}{d} \right\rfloor \right\rangle_p .$$

Calculating the left hand side is fairly straightforward: all we need to do is reduce the $y_{i,j}$'s modulo p and sum them as we did before. The right hand side is more tricky. Before, the operation of rounding was made very easy by the fact that we could access each individual bit of our $\rho + 1$ bit number. But when working modulo p, rounding a number using only addition and multiplication is much harder. If p is not prime, it is actually impossible to do the rounding operation; and if p is prime, then the complexity of rounding is $O(p^2)$. This is

obviously not good enough. To get around this, I decided to use a different representation for my fractional numbers (which I will, as before, call $z_{i,j}$). I represent the integer part (to the left of the binary point) of the $z_{i,j}$s as a single integer modulo p. The fractional part is represented using a "hot bit". Recall we had set $\rho = \lceil \log_2(s+1) \rceil$, here I use $2^\rho = s+1$ integers to represent this fractional part, s of which are 0 and one of which is 1.

Now I need to show how to implement the sum of these $z_{i,j}$'s. First note that just like for bits, the first sum $\sum_{j=0}^{S-1} \sigma_{i,j} z_{i,j}$ is very easy to do. Since only one $\sigma_{i,j}$ per big set is one, all we need to do is the following: calculate $z_{i,j}$ for every j, replace all the ones in the representation of the $z_{i,j}$'s by their corresponding $\sigma_{i,j}$, sum for all j's each individual parts of the $z_{i,j}$'s together. Denote this result by z_i. Now I need to take care of the more complex sum $\sum_{i=1}^{s} z_i$. The process of computing this sum is slightly technical, I therefore explain it in appendix A.4.

Once the sum is computed, all that is left to do is to round the sum of these integers. This is made very easy by the representation I use. Call z the sum, then the value of $\lceil z \rfloor$ is the integer part of z, plus the sum of the hot bits of z representing fractions greater or equal to $1/2$.

To make things simpler, I make a requirement for the key generation that $d = 1 \pmod{p}$. I describe in section 3.4 a procedure to avoid having to keep generating keys until this is the case. Therefore, the decryption procedure is:

$$\mathsf{Dec}(c) = \left(\sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} \langle y_{i,j} \rangle_p \right) - \left\langle \left\lceil \sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} z_{i,j} \right\rfloor \right\rangle_p .$$

On the left side, we just need to reduce the $y_{i,j}$'s modulo p and multiply them by $\sigma_{i,j}$. In the implementation, I do this on the side while computing the $z_{i,j}$'s. Since all the additions are implicitly done modulo p, we get an integer modulo p on the left side and an integer modulo p on the right side as we just saw. Subtracting these two should produce an integer representing the same value as c, but with a noise vector independent of the noise vector in c.

3.4 Speeding up key generation for large message spaces

As I previously mentioned, a requirement for both w and d is that their value modulo p is 1. For w this is not a very strong requirement. Since we have the vector $\vec{w}$ we just need to scan through its elements to find one that satisfies the condition. For d this is tougher: in [5] the approach taken was to generate keys until d was odd (everything was done modulo 2). In my implementation, I speed things up by noticing that d is the determinant of the matrix V and therefore is just a polynomial in the coefficients of v: $d = f(v_0, v_1, \ldots, v_{n-1})$. Therefore, $d = f(\langle v_0 \rangle_p, \langle v_1 \rangle_p, \ldots, \langle v_{n-1} \rangle_p) \pmod{p}$. Since the bit size of p is much smaller than t the bit size of the coefficients of $\vec{v}$, this operation is going to be much faster. Therefore, before generating a key, we check that $\langle f(\langle v_0 \rangle_p, \langle v_1 \rangle_p, \ldots, \langle v_n - 1 \rangle_p) \rangle_p = 1$, which should happen with probability $1/p$.

One last point about the key generation: apart from the two requirements I mentioned above, there is no link between the somewhat homomorphic key and the value of p. Therefore it is

possible to use a single somewhat homomorphic key for different values of p. If we have $p_1, p_2, \ldots, p_k$, what we need to do regarding the choice of w and d is to make sure that their value modulo the lowest common denominator of the p_i's is 1.

3.5 Recrypting an integer modulo a power of 2 to bits

In this section, I will explain how to tune the recryption method to recrypt a ciphertext c encrypted modulo $p = 2^k$ into k bits. But first I consider the pros and cons of using a large message space rather than bits.

A couple important facts to realise:

- The increase in the error vector when doing an addition is close to negligible compared to the increase when doing a multiplication. This is why I will refer to the degree of a ciphertext as it quantifies its error vector.
- The larger the p we use, the faster the error vector increases. In fact, for two ciphertexts of the same degree deg, but of different moduli: p_1 and p_2, the size of the error vector in the second ciphertext should be about $(\frac{p_2}{p_1})^{deg}$ times bigger than in the first one.
- The maximum error vector in a ciphertext (and therefore the maximum number of multiplications) is mainly dependent on the bit size t of the coefficients of the generating polynomial.
- When deciding on a value for t, the important factor to consider is the number of multiplications that can be done on ciphertexts that have already been recrypted before having to do another recryption. In [5] for example, the chosen bit size was $t = 380$ which allowed the authors to do a single multiplication on two recrypted bits.
- The value of t has a strong impact on the complexity of the computations. Recall that all operations are done modulo d. From experiments, the bit size of d is about nt.
- When considering the efficiency of a scheme, the two main parameters are going to be the number of multiplications that can be done between two recryptions and the complexity of doing this recryption. This is because the time to do one multiplication is usually negligible compared to the time to do one recryption.

I am now going to compare the representation of a number encrypted as a bit vector of length k against the encryption of the same number modulo 2^k. There are three types of operation I consider:

- *Additions and multiplications*. In the bit representation, addition is done using grade-school addition, which requires about $2k$ multiplications (elementary symmetric polynomial for every column with two bits and one carry bit). This means $O(k)$ recryptions are necessary, with the exact value depending on how many multiplications per recryption the scheme allows us to do. Multiplications in the bit representation are going to have an even worse complexity. First, the product of half of the possible pairings

of bits has to be computed. This already means $k^2/2$ multiplications. Then we have to perform the addition of k numbers, with bit sizes going from 1 to k which is another $k^2/2$ multiplications. This means $O(k^2)$ recryptions, making this scheme alone is not viable for large integers. Using a large message space however, one can do a very large number of additions or (at least) one large multiplication before having to recrypt.

- *Right shift*. In the bit representation, doing a right shift is trivial, all one needs to do is shift the encrypted bits in the bit vector. However, in the large message space modulo a power of 2, it is impossible to perform a right shift using only additions and multiplications. The bit representation therefore needs to be used here.

- *Comparison*. I only consider a test for equality between two ciphertexts, but other comparisons – such as greater than or smaller than – show the same properties. The aim of a comparison is to output (in a ciphertext) 1 if the comparison is true and 0 if the comparison is false. In the bit representation, testing for equality can be done easily. First we do a pairwise XOR on the bits, if two bits are the same then their XOR is 0, therefore if the two ciphertexts are the same, their XOR should be a bit vector where every bit is 0. Then we add 1 to every bit, to get a vector of 1s, and finally AND every bit together. Since the XORs are additions and the ANDs multiplication, this procedure corresponds to $k-1$ multiplications. In the large message space, if p is not prime it is impossible to do a comparison, and if it is prime then it takes p – and therefore about 2^k to be in the same range – multiplications. For these operations again, the bit representation is more efficient.

It should be clear now that depending on the operation, one representation is more efficient than the other. Therefore, what would be useful to have is a function that converts from one representation to the other. I now describe an efficient way to achieve this.

As I mentioned before, it is possible to use the same somewhat homomorphic key to do operations for different moduli: $p_1, p_2 \dots p_k$. Note that this means that we include in the public key encryptions of the secret key using each p when extending the scheme to be fully homomorphic. If it is the case that for some i and j, $p_i = \kappa \times p_j$ where κ is an integer, then we could re-use the encryption of the secret key we used for p_j for p_i. However, since $p_j > p_i$, the error vectors in the ciphertexts of the encrypted secret key are higher if we use p_j than if we use p_i. This means that we possibly would not be able to do as many multiplications between recrypted ciphertexts if we used the key of p_j instead of p_i. For the case of using $p = 2^m$ and $p_1 = 2$, I will therefore have an encrypted secret key for both p and p_1. Next, I am going go over how to use the encryption of the secret key modulo 2 to recrypt a ciphertext modulo p into a series of bits. Recall that what I have called "recryption" is just the operation of homomorphically decrypting a ciphertext. Here, I am going to use the encryption of the secret key modulo 2 to homomorphically decrypt a ciphertext modulo p.

Recall one more time the decryption function

$$\mathsf{Dec}(c) = \left(\sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} \langle y_{i,j} \rangle_p \right) - \left\lceil \sum_{i=1}^{s} \sum_{j=0}^{S-1} \sigma_{i,j} z_{i,j} \right\rfloor .$$

The left sum that was straightforward to calculate before is now slightly harder. For each

$y_{i,j}$, we are going to reduce $y_{i,j}$ modulo p and then turn it into a bit vector of size m (recall $p = 2^m$). This bit vector of 0's and 1's is then multiplied by $\sigma_{i,j}$, which is done by replacing the 1's by $\sigma_{i,j}$. Because there is only one $\sigma_{i,j}$ that is non-zero per block, the sum $\sum_{j=0}^{S-1} \sigma_{i,j}\langle y_{i,j}\rangle$ can be done as before by just pairwise XORing every vector. The outer sum can then be done using the Grade-School addition (GSA) that I described for bits earlier. I will come back to this later.

The right sum is of the exact same type as the one we had for bits. However, when we were recrypting bits, we were only interested in the bit to the left of the binary point. This is because we were considering the value of the decryption modulo 2. Now that p is larger, we use the GSA algorithm to compute the m bits to the left of the binary point. There is one slight complication to this however: because we are rounding to the nearest unit, the bit to the right of the binary point should be added to the integer part of the sum. Instead of using GSA to calculate the sum and then using GSA again to add the single bit, I modified the GSA algorithm to carry both the carry bit and the result bit from the column to the right of the binary point to the column to the left of the binary point. Note that using this method greatly decreases the degree of the recryption.

We are not quite finished, at the moment we are doing a GSA on the left side and a GSA on the right side. As we have seen, it is better to do GSA's together to minimise the degree of computations. In my implementation, the two sums are done in a single GSA; the details of how to do this in this case are slightly technical, I therefore include them in appendix A.5.

In my implementation, the maximum degree for the individual bits set by the somewhat homomorphic scheme is higher than the degree required for this procedure. However, if the complexity of the procedure was to be too high (if the top bit had a noise too large to be decrypted), it would be possible to use the recryption of individual bits during the procedure. For example, recrypting every carry bit before it is carried would decrease significantly the maximum degree needed. It is likely, however, that this would make it a much heavier computation.

I am going to finish this section with a note on how to go back from a bit representation to a large message space representation. Recall that a bit is encrypted as $c = [b + 2R]_d$ for some value R. Hence $c^2 = [b^2 + 4R + 4R^2]_d$ which is the value of b encrypted in the message space modulo 4. Therefore if we want to go back from m bits to a representation in the message space modulo m, all we need to do is square each bit m times, possibly recrypting on the way. Then for $k \in \{1 \dots m\}$, multiply the k^{th} bit by 2^k and finally add all the bits together.

3.6 Distributing the scheme

In this section I describe how I distributed the encryption and recryption functions over multiple parallel computation units. I chose to distribute these two functions for the following reasons:

- They are both highly parallelizable.
- They are the two most computationally expensive functions. The key generation also is expensive, but only because it involves encrypting the secret key.

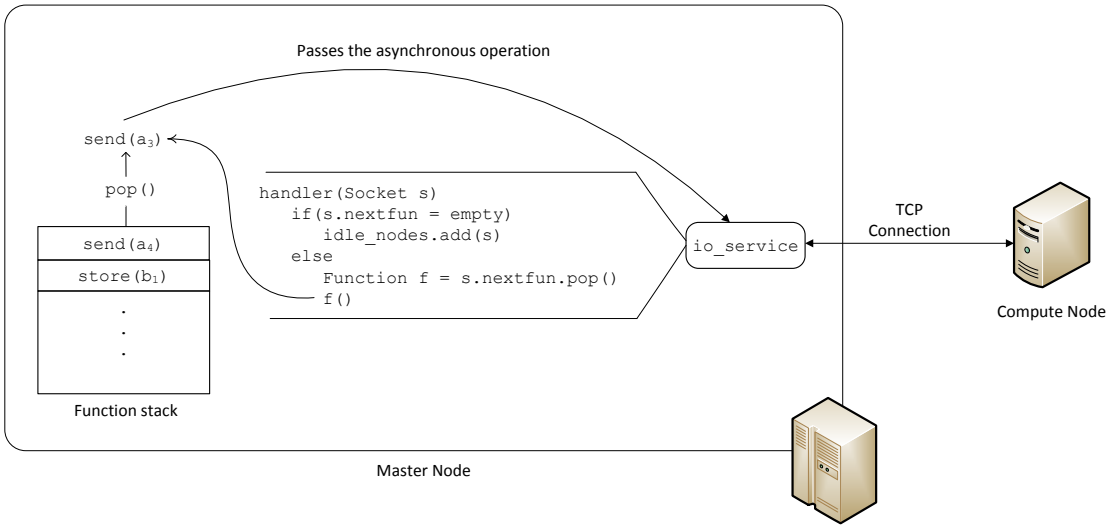

Figure 3.1: Asynchronous operation in the Master Node. When an asynchronous operation terminates, the `io_service` calls the handler function which itself calls the top function in the stack. If the function is another asynchronous operation, it gets passed to the `io_service` to be executed next.

The scheme was distributed in a "task farm" model: a *Master Node* (MN) uses a number of *Compute Nodes* (CNs) as big calculators. The underlying connections were implemented using TCP. The message passing was implemented using synchronous I/O on the CNs, and asynchronous I/O on the MN. The reason for using asynchronous I/O on the MN is to avoid waiting on a result from a CN while some other CNs are finished computing their result. However, this also means making sure that there are never two asynchronous operations running on the same socket, as otherwise the messages passed will get corrupted. In a first subsection I am going to go over how I implemented the asynchronous I/O in the MN, then I will describe the distribution of the encryption, and finally that of the recryption.

3.6.1 Asynchronous Operations in the Master Node

In this section I describe the asynchronous operation in the MN. First I am going to detail how asynchronous operations work in the library that I am using (`boost.asio`).

When working with `boost.asio`, the programmer first creates an `io_service` which is going to handle all the I/O operations. After that, a number of asynchronous operations can be passed to the `io_service`, but they are not executed directly. The programmer associates a handler function with each operation, which will be called whenever the operation is finished. Note that it is possible to ask the `io_service` to perform some more asynchronous operations within a handler. Once some asynchronous functions have been passed to the `io_service`, it is possible to call `io_service.run()` which is a blocking function running all asynchronous operations until there are none left. To do a series of operations in order, the programmer will therefore have to ask the handler of the first operation to pass the second operation to the `io_service` and so on.

In my implementation, there are only very few atomic operations such as sending or receiv-

ing an integer. To each socket, I associate a stack of functions that need to be executed. I attach the arguments of the functions to the functions themselves using `boost::bind`; this way, none of the functions in the stack needs arguments. To every operation done asynchronously, I attach the same handler function. The handler when called pops the first function out of the stack associated with the socket of the operation that just finished and calls it. Most of the time, this function is another asynchronous operation which will be then passed to the `io_service` and executed whenever possible as shown in Figure 3.1.

Consider the example of sending a number of integers $a_1, \ldots, a_k$ to the CN so that the values of the elementary symmetric polynomials $b_1, \ldots, b_l$ of these integers can be computed. First, I put $send(a_1), \ldots, send(a_n)$ on the queue associated with this CN, then I add to the queue the functions $store(b_1), \ldots, store(b_l)$. Finally I pass to the `io_service` an asynchronous operation that tells the CN what to expect (k integers and the elementary symmetric polynomials in return). When this message is sent, the handler is called, which will itself pass $send(a_1)$ to the `io_service` and so on. Once the queue of functions is empty, the handle function notifies the MN which puts the socket into a list of idle nodes. It may then get used again to compute more operations.

3.6.2 Distributing the encryption

In this section, I explain how I implemented the distribution of the encryption.

There are two parts to the encryption function: first the computation of the different powers of r, then the encryption of the bits using these powers. Notice that if the powers have already been computed (which is the case after the secret key is encrypted), then there is no need to compute them again and we can therefore skip to encrypting the bits. This is thanks to my change in the encryption procedure from section 3.1.5. Another interesting point to note is the fact that, while the computation of the encryption of bits has to be done on secure nodes, the computation of the powers of r can be done on any node as r is in the public key.

3.6.2.1 Computing the powers of r

In this section I explain how to distribute the computation of the powers of r. I attempt to minimise the amount of I/O between the MN and the CNs by only sending to the CNs the powers of r that they need for their computations. This might seem useless since any CN that is going to encrypt bits later is going to need every power of r, but I do it because some CNs might not be secure and therefore it is not necessary that they hold every power of r. Note, however, that whenever an CN computes a power of r, it returns the result to the MN. The MN therefore holds the value of every power of r that has already been computed.

The computations of the powers is scheduled in the following way: I have a queue of idle nodes, and whenever a node finishes a computation, it is added at the end of the queue. Whenever any computations are possible – i.e. r^a and r^b have finished computing, but r^{a+b} has not yet been computed – and the queue of idle nodes is not empty, the MN pops the first node n_i out of the queue to assign a task to it. The MN prefers to schedule the computation that minimises the number of powers of r that need to be sent to nodes. To do this, the MN first considers the set of powers that have not been computed yet, and can be computed

straight away from the powers n_i has in memory. If this set is empty, it considers the set of powers that can be computed by sending one integer to n_i. If this set is empty again, it considers the set of powers that can be computed by sending two integers to n_i (this is always non-empty since some computations are possible). When considering a set of different powers node n_i could compute, the MN picks the power that the least number of nodes can compute straight away, and if there is a tie, it picks the one that the least number of nodes can compute by being sent one integer. This heuristic way to solve the problem of sending the minimum integers works very well, with only a very small number of powers being sent from the MN to the CNs.

3.6.2.2 Computing the encryptions of the bits

Once all the powers are computed, we can start encrypting the messages. Since it is likely that we have many more messages than we have nodes, there is no need to parallelize on a sub-message granularity. Therefore, all I do is to send the missing powers of r to all secure CNs, and then send them messages which they encrypt using the method I described in section 3.1.5 and return the result.

3.6.3 Distributing the recryption

In this section, I describe my implementation of the different types of recryption, that is, recryption of a bit, recryption of an integer, and recryption of an integer to bits. First recall from section 3.1.4 that when encrypting the secret key, I do not encrypt the $\sigma_{i,j}$'s directly. Instead, I encrypt a smaller number of integers which I call $\eta_{i,j}$. Using the function $ind(a,b)$ from equation 3.3 I then have $\sigma_{i,ind(a,b)} = \eta_{i,a} \cdot \eta_{i,b}$. Remember this means that the two sums computed in the recryption, $\sum_{j=0}^{S-1} \sigma_{i,j} \langle y_{i,j} \rangle_p$ and $\sum_{j=0}^{S-1} \sigma_{i,j} z_{i,j}$, are instead computed as $\sum_a \eta_{i,a} \sum_b \eta_{i,b} \langle y_{i,ind(a,b)} \rangle_p$ and $\sum_a \eta_{i,a} \sum_b \eta_{i,b} z_{i,ind(a,b)}$ respectively. The maximum values of a and b are chosen so that $ind(a,b)$ spans the range $\{0, \ldots, S-1\}$.

An operation that is common to every recryption is the one of computing, for a given i and a, the two summations $\eta_{i,a} \sum_b \eta_{i,b} \langle y_{i,ind(a,b)} \rangle_p$ and $\eta_{i,a} \sum_b \eta_{i,b} z_{i,ind(a,b)}$. I will often refer to these sums as a *sumline* as we are summing one line of the two dimensional array of the $\sigma_{i,j}$ formed by the products of the η's. Due to the different representations of the three recryption approaches, there are a few differences in the way these sumlines are computed. However, the way they are distributed is the exact same, as I am going to demonstrate now.

3.6.3.1 Distribution of the sumlines operations

As can be seen, for each block there will be a fixed number of lines that need to be computed (the a's in the function above). To compute a line in block i, it is necessary to have all (or very close to all) the $\eta_{i,j}$'s in the block i. In this implementation, we will send blocks ($\vec{\eta}_i$) individually to the CNs as they need them. To optimise the running time of the function, I will attempt to minimise the number of blocks sent to the CNs. As for the encryption, we have a list of idle nodes that we can use. While this list is not empty and there are some lines in some blocks that still need to be computed, I do the following operations:

1. Pop the first node of the idle nodes list.
2. Consider the set of lines that have not been computed yet and can be computed by the CN without sending a new block.
3. If this set is empty, consider instead the set of lines that have not been computed in general.
4. Pick a line in the set being considered, so that the block containing the line is the one that has been sent to the least number of nodes.

This way, I optimise node utilisation as I preferentially compute results that cannot be computed elsewhere.

Once the sumlines operations are all done, the MN adds up the sumlines of the same block to get the final values of $\sum_a \eta_{i,a} \sum_b \eta_{i,b} \langle y_{i,ind(a,b)} \rangle_p$ and $\sum_a \eta_{i,a} \sum_b \eta_{i,b} z_{i,ind(a,b)}$. Because there is only one non-zero pair $\eta_{i,a} \cdot \eta_{i,b}$ per i, this operations only requires additions and no multiplications and therefore should be very fast in comparison to the rest of the computation.

There are now only two operations left which can usefully be distributed, grade school addition, which will let us do both recryption of bits and recryption to bits, and the addition of numbers in the special representation I described in section 3.3.

3.6.3.2 Distributing Grade School Addition

Recall that grade school addition is performed by putting one number per line so that bits of the same order are on the same column. Then, starting from the rightmost column, we compute the carry bits sent to the other columns by evaluating the elementary symmetric polynomials of the bits in that column. Recall that the elementary symmetric polynomial of degree k of the variable $X_1, \ldots, X_n$ is calculated by

$$e_k(X_1, \ldots, X_n) = \sum_{1 \leq i_1 < i_2 < \cdots < i_k \leq n} X_{i_1} X_{i_2} \ldots X_{i_k}.$$

However, by factorising the terms that are a factor of X_n, we can modify this equation to give:

$$\begin{aligned} e_k(X_1, \ldots, X_n) &= X_n \sum_{1 \leq i_1 < i_2 < \cdots < i_{k-1} \leq n-1} X_{i_1} X_{i_2} \ldots X_{i_{k-1}} + \sum_{1 \leq i_1 < i_2 < \cdots < i_k \leq n-1} X_{i_1} X_{i_2} \ldots X_{i_k} \\ &= X_n \times e_{k-1}(X_1, \ldots, X_{n-1}) + e_k(X_1, \ldots, X_{n-1}). \end{aligned}$$

This is very useful. It means that we can do almost all calculations to compute the elementary symmetric polynomials of a column before we know what the carry bits of the column are, enabling parallelism. I therefore implemented the evaluation of the elementary symmetric polynomials in the CNs so that every time a bit is sent, the CN does all the computations it can before doing a blocking wait on the next bit.

In the MN, grade school addition is implemented in the following way: while there are some idle nodes and some columns that have not started to be computed, I pop the top node out

of the idle nodes list and tells it to evaluate the rightmost column not yet computed. I also include in the instructions the number of bits the node should expect before returning the result. However, I only send it the bits that I already have. Then, whenever a node returns a result, I transfer the carry bits to the nodes computing the relevant column.

3.6.3.3 Distributing the sum of hot bits numbers

In appendix A.4, I describe the addition of two numbers using the "hot bit" representation. I also explain that the sum of s of these numbers is done in a "knock out" manner in order to minimise the final noise. This concretely means that I only add two numbers if they are of the same degree. Here, I will describe the scheduling of doing a sum of s numbers, where a CN can return the addition of two numbers.

To distribute this, I use an array of stacks of length $\lceil \log_2(s) \rceil + 1$. I then put all s numbers in the first stack of this array. Then, whenever there is an idle node and one of the stacks has two or more elements, I pop the first node out of the idle nodes list. I then pop two numbers out of the stack that has two or more elements and has the lowest index i in the array. I send these two elements to the node and tell it to return the result in the stack with index $i + 1$. In the special case where there is a single element by itself in a stack and there are no other numbers expected to come, I promote the number from this stack to the next. Using this method, I expect to get the sum of my s numbers in the last stack of the array after $s - 1$ additions.

3.7 Conclusion

In this chapter, I have detailed the implementation of my project. I started in section 3.1 by describing my implementation of the previous scheme as described by Gentry and Halevi [5]. I then followed with my contributions, starting with a different and simpler understanding of the somewhat homomorphic scheme (3.2) built from some of the evaluation results. In sections 3.3 - 3.5, I used this understanding to modify the scheme from encrypting bits to encrypting modular integers, and described functions to switch between the two representations. An optimisation to avoid generating bad keys was also included. Finally, in section 3.6, I described the distribution of the computationally expensive functions of the scheme

Chapter 4

Evaluation

The objective of this chapter is to evaluate the parallel fully-homomorphic encryption system described in chapter 3. There are three main parts to it: first, I will show that homomorphic computations using my implementation work and present the general performance of the scheme. Then I will describe some experimental results on the properties of the somewhat homomorphic scheme; I will study the distribution of the coefficients of the vector $\vec{w}$ and its impact on the scheme. Finally, I will review the gain in performance induced by my optimisations. I will use benchmarks to review quantitatively the efficiency of the different schemes and compare the standard and parallel implementations.

Throughout the evaluation, I used two different machines; one to evaluate the local implementation and the other one to evaluate the distributed implementation. Their specifications are detailed in appendix A.7.

4.1 Test of correctness and basic performance evaluation

In this section, I test the different components of the scheme and outline the basic complexity of the different functions.

4.1.1 Unit test of implementation

To test the correctness of the implementation, I had to verify that I was able to:

- Generate a somewhat homomorphic key.
- Encrypt and decrypt messages using this key.
- Perform homomorphic operations on these messages.
- Generate a fully homomorphic key.
- Recrypt messages using the fully homomorphic key.

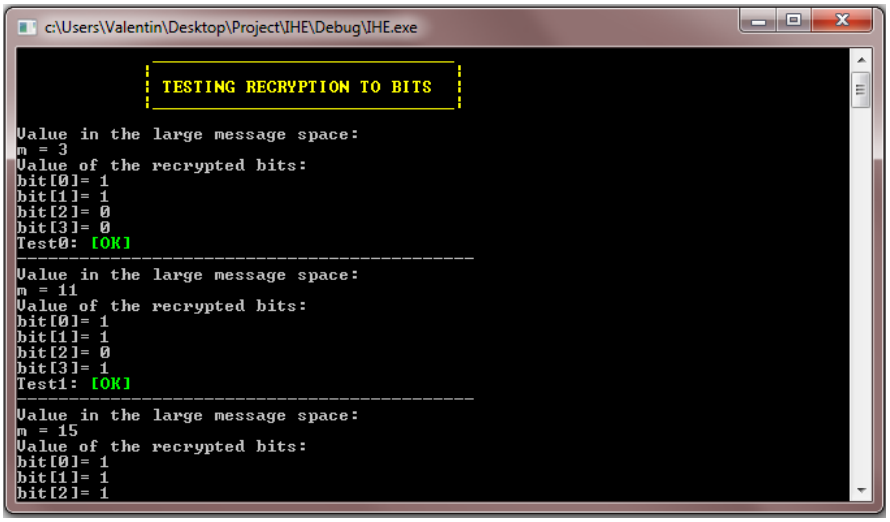

Figure 4.1: Output from the unit test of recryption to bits.

Both the encryption and recryption had to be verified both for the simple implementation and the distributed implementation. To verify that the somewhat homomorphic functions are working, I generated ten keys for parameters $n = 2^8, 2^9, \ldots, 2^{12}$ and $t = 200, 400$. For each key, I encrypted twenty random numbers in the message space $0 \ldots 255$. I then compared the decryption of every pairwise multiplication and every pairwise addition and compared it to the expected result. In each case, the decryption gave the expected output. I then repeated this procedure using the distributed version of the encryption, which again yielded the expected result.

I then proceeded to verify the correctness of the recryption functions. I had to verify that the three functions of recrypting a single bit, recrypting a large message and recrypting a large message to bits all generate the correct outputs. For both the standard and distributed implementations, I generated a somewhat homomorphic key with parameters $n = 2^{10}$ and $t = 1000$. I then extended this key by adding an encryption of the secret key – recall that we need a separate encryption for each message space – using one encryption in the bit message space and one encryption in the $0 \ldots 255$ message space. For both representations, I then encrypted twenty uniformly distributed random numbers and recrypted them straight away. After this, I computed every pairwise multiplication and recrypted the results.

There are two important characteristics of the recryption: the values represented by the ciphertext should be the same before and after recryption, and the noise in the recrypted ciphertext should be independent of the noise of the ciphertext before recryption. I therefore checked two properties on the recrypted products: that they were holding the expected result and that the noise – estimated by $[cw]_d$ – was about the same as the one of the ciphertexts freshly encrypted and then recrypted. For each of the forty variables I had encrypted, the results were the one expected.

Finally, I tested the recryption to bits. I encrypted twenty recrypted ciphertexts in the message space $0 \ldots 15$ and applied it to them. I then compared the decrypted result with the bit representation of the messages in the clear. Again, the result was the correct one both in the standard and distributed implementation. The output of the standard implementation is

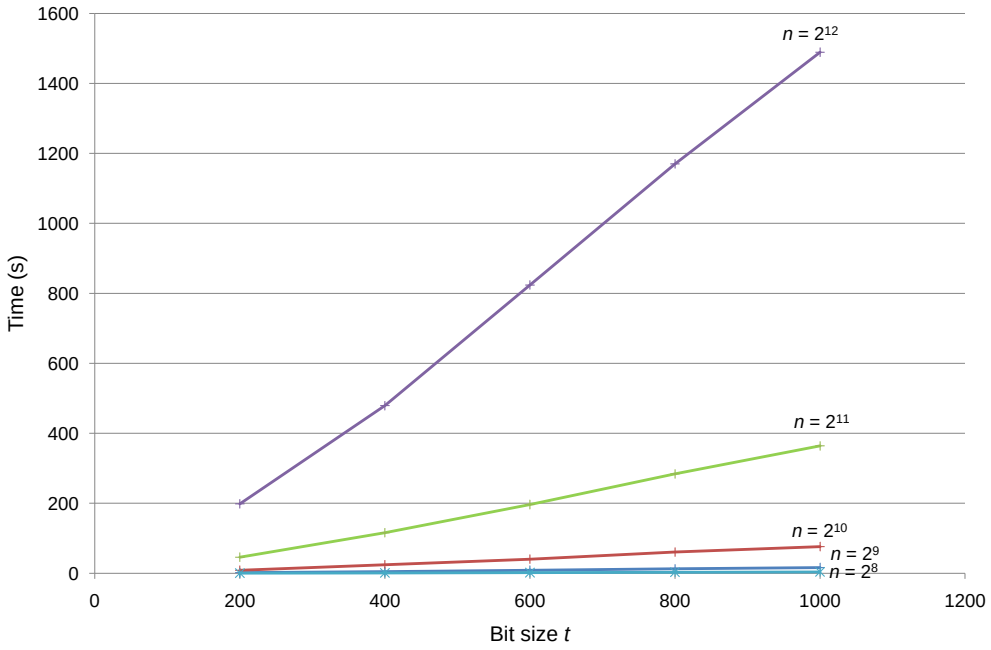

Figure 4.2: Time taken to generate a fully homomorphic key.

shown in figure 4.1.

4.1.2 Basic performance of the implementation

I now review the general performance of the homomorphic encryption scheme to give an idea of the complexity of the different operations. To do this, I show how time and space requirements of the different functions are affected by both parameters n and t. The values of the other parameters used in the fully homomorphic scheme can be deduced from n and t using some equations from [5] which I do not detail here. After this, I will have a brief look at the noise of a ciphertext that has just been recrypted.

4.1.2.1 Time and space complexity

In this section, I analyse the time and space complexity of the key generation and the time complexity of recryption using empirical results. Since most of the computations of the key generation are spent encrypting the bits of the secret key, I do not consider the complexity of the encryption as the results would be similar. Because the decryption is basically only one multiplication, I do not consider its complexity either, as it is negligible in comparison to the other functions. As the space complexity of the recryption is very small compared to the space already taken up by the public key, I do not consider it either.

I start with the complexity of the key generation. I generate twenty-five (fully homomorphic) keys for the five dimensions $n = 2^8, \ldots, 2^{12}$ and the five bit sizes $= 200, 400, \ldots, 1000$. Each key includes two encryptions of the secret key, one modulo 2, and another modulo 16. Figures 4.2 and 4.3 show the time and space used to compute each key.

I now move on to looking at the time taken to do one recryption. Recall that there are three types of recryptions; for each of the keys above I perform one bit recryption, one recryption in the message space modulo 16, and one recryption to bits from the large message space. The figures 4.4 - 4.6 show the time taken to compute each of them.

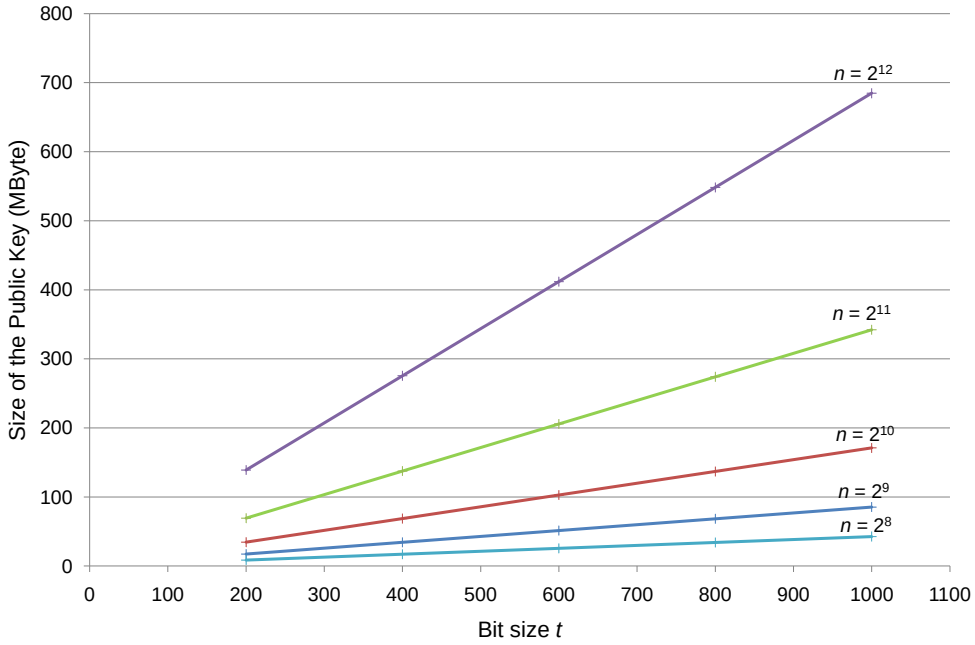

Figure 4.3: Size of the public key.

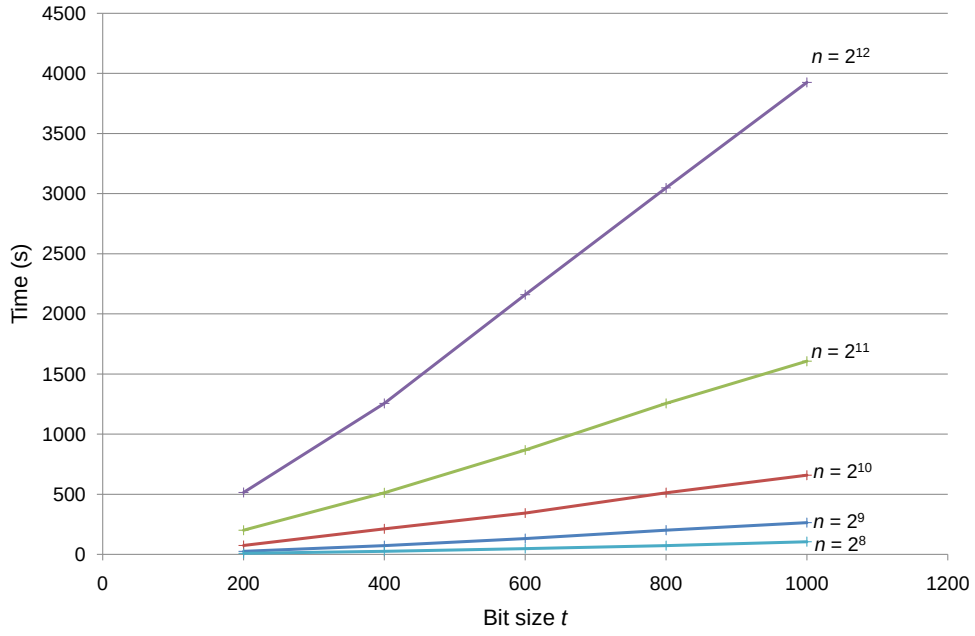

Figure 4.4: Time to recrypt a single bit.

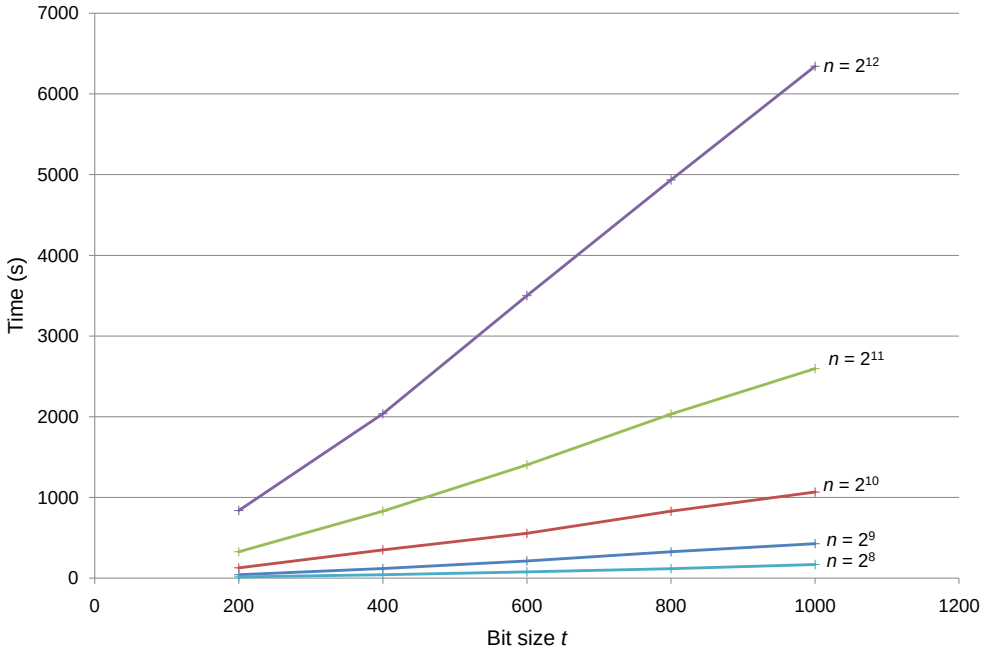

Figure 4.5: Time to recrypt an integer in the large message space.

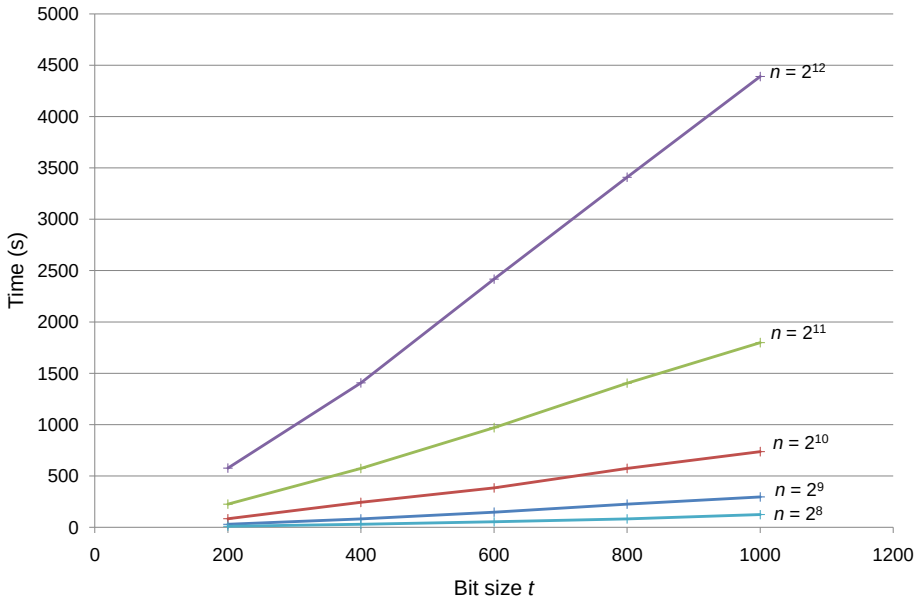

Figure 4.6: Time to recrypt an integer to bits.

Note however that some of these recryptions were ineffective. For the low values of t, the degree of the decryption is too high for the somewhat homomorphic scheme to support it and therefore the ciphertexts returned by the recryption are corrupted.

As one can see from these graphs, both time and space complexity tend to vary linearly with n and t, except for key generation time, which has a slightly higher complexity in n. To make the rest of the evaluation easier, I will mostly use $n = 2^8$; this is far from secure, but results in higher dimensions can be deduced by using the graphs above. I will often use $t = 1000$, as this is the minimum bit size that can handle the complexity of recrypting to bits an integer in the message space $0, \ldots, 15$.

4.1.2.2 Degree of the recryption

Because the recryption has a large number of additions associated with it, it is not easy to estimate in advance the noise of a ciphertext that has just been recrypted. However, it is useful to associate a degree deg_{dec} with the procedure, meaning the resulting ciphertext has approximately the same noise as the one of the product of deg_{dec} ciphertexts. In this section, I experimentally evaluate the degree of the different recryptions.

The parameters that have an impact on deg_{dec} and should be varied are:

- The number of big sets in the public key s. However, in [5] it is shown that using $s = 15$ should always be secure. Therefore I always use $s = 15$ and there is no need to consider its effects on deg_{dec}.
- The number of elements in a big set S. The conditions on this parameter are $S \geq 512$ and $S \geq n/s$. Therefore, in high dimensions ($n > 2^{12}$), we expect n to have an impact on deg_{dec}. For this reason, I show how deg_{dec} varies with S so that the result presented here can be used to estimate deg_{dec} in any dimension.

Using $n = 2^8$ and $t = 1000$, I encrypted twenty bits and twenty integers in the message space $0, \ldots, 255$. In order to know what the noise in a ciphertext of degree deg should be, for each ciphertext, I computed each power c^{deg} for $deg = 1, \ldots, 300$. I then evaluated the noise in each of these powers as $[cw]_d$. Using these results, for each degree deg, I stored the average noise[1] measured in a ciphertext of degree deg for both encrypted bits and encrypted integers.

Using this, I can evaluate the degree of the recryptions. For $S = 100, 200, \ldots, 1000$, I computed each type of recryption (recryption of a bit, recryption of an integer and recryption of an integer to bits) on each relevant ciphertext. I then evaluated the noise of each recrypted ciphertext. Because I wanted a pessimal noise, I only stored the highest noise and compared it to the ones computed for each degree. The results are shown in figures 4.7 - 4.9.

As one can see, the degree of recrypting to bits is much higher than the one of the other recryptions. This is an indication that it may be useful to use recryption of the individual bits during the computation, as I suggested in section 3.5. This will most likely be necessary if one wanted to encode decent size integers such as 32 bits.

[1] The noise grows exponentially and therefore I used the average in the log scale

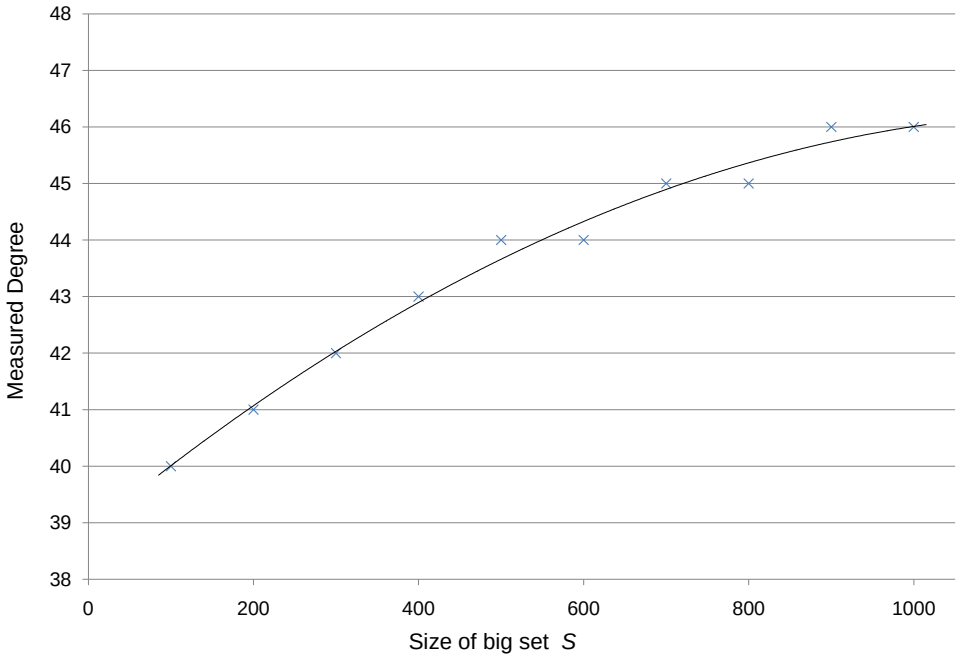

Figure 4.7: Degree of the recryption of a bit.

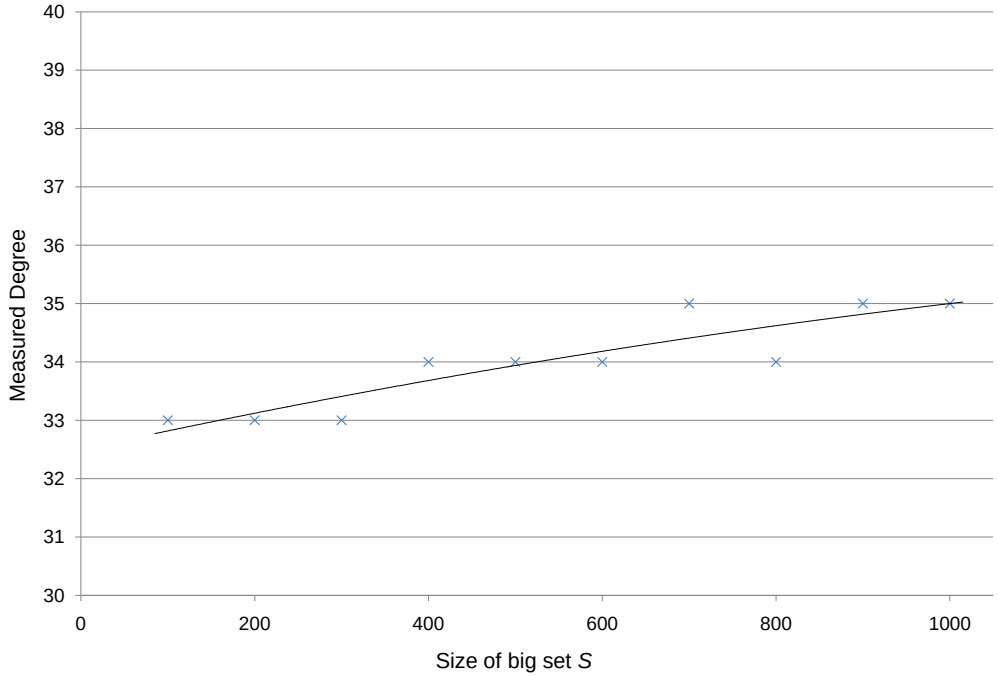

Figure 4.8: Degree of the recryption of an integer.

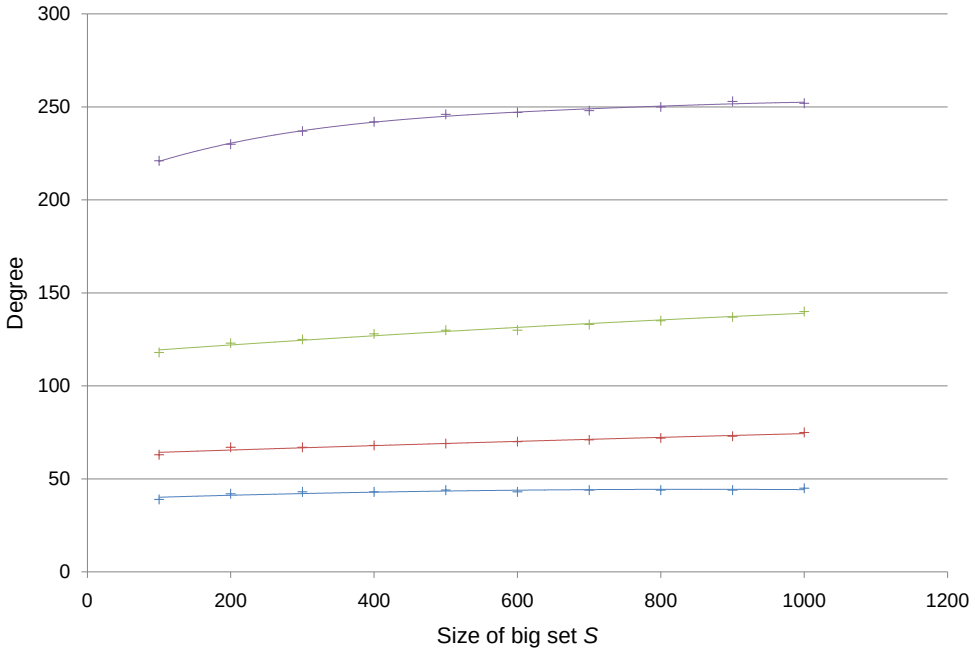

Figure 4.9: Degree of the recryption of the four bits of lower order when recrypting to bits. The curve with lowest degree represents the degree of the bit of order 0 and so on.

4.2 Experimental results on the somewhat homomorphic scheme

The aim of the following experiments is to get a better understanding of the somewhat homomorphic scheme. This section represents new contribution to the field of homomorphic encryption, and some of its results were used in section 3.2 to build a new understanding of how the somewhat homomorphic scheme works. I am going to discuss two main topics, first the distribution of the coefficients of the vector $\vec{w}$, then the impact of this distribution on the way the scheme works.

4.2.1 Distribution of the w_i's

As I mentioned in the implementation chapter, I show in this section that the size of the w_i's is much smaller than d. In fact, I show it is a normal distribution centered at 0 and with standard deviation approximately $\sigma = K\frac{d}{2^t\sqrt{n}}$, where K is a constant. I first go over the results indicating that it is a normal distribution, then I discuss the approximation of the standard deviation.

4.2.1.1 Determining the distribution

In this section, I show that the coefficients of the polynomial $\vec{w}$ satisfy a normal distribution.

One important property to note is that the w_i's must be identically distributed and that their distribution is centered around 0. Consider the generating polynomial $\vec{v} = \langle v_0, v_1, \ldots, v_{n-1} \rangle$. If $\vec{w} = \langle w_0, w_1, \ldots, w_{n-1} \rangle$ is the polynomial associated with $\vec{v}$, then the polynomial associated with $\vec{v}' = \langle v_1, v_2, \ldots, v_{n-1}, -v_1 \rangle$ must be $\vec{w}' = \langle -w_{n-1}, w_0, w_1, \ldots, w_{n-2} \rangle$. This is basically

multiplying and dividing by x the left side of equation 3.2. Since the coefficients of $\vec{v}$ are uniformly distributed around 0, it is clear that the distribution of the w_i's is independent of i. Also, given a probability for a value of w_0, one can see that there must be the exact same probability to get the same set of coefficients, but with the value of $-w_0$ instead. Since the coefficients are identically distributed, I conclude that their distribution must be centered around 0.

I generated five random key using the key generation algorithm, one for each dimension n in the range $2^8, 2^9, \ldots, 2^{12}$, and using random values for t in the range 100-500. For each key, I computed the standard deviation σ of the distribution of the w_i's. Since I know the w_i's are distributed around 0, I used 0 as the mean value in my calculation. Then, I compare the distribution of the w_i's (including $-w_i$'s to double the number of data points) with a Gaussian $\mathcal{N}(0, \sigma^2)$ using a Kolmogorov-Smirnov test. The table below shows the value of the Kolmogorov-Smirnov test statistic D for each test. That is, the maximum difference between the expected and the measured value of the cumulative distribution.

n	t	D
2^8	454	0.0301
2^9	166	0.0120
2^{10}	200	0.0087
2^{11}	485	0.0071
2^{12}	258	0.0087

As one can see, D is rather small, meaning the measured distribution is very close to a normal distribution. For example, figure 4.10 shows the measured distribution of the w_i's for $n = 2^{12}$ along with the corresponding normal distribution.

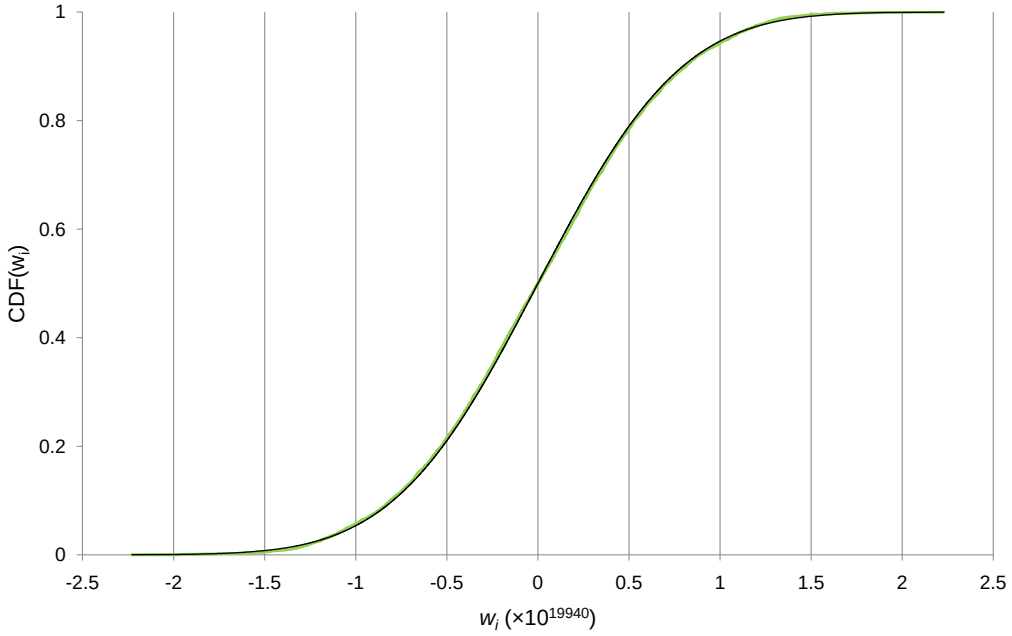

Figure 4.10: Cumulative probability distribution of the w_i's with parameters $n = 2^{12}$ and $t = 258$ is plotted in green and its corresponding normal distribution with the same standard deviation in black.

4.2.1.2 Size of the standard deviation

In this section, I show a way to approximate the standard deviation of the w_i's. I begin with a guess of what I think the standard deviation should be, then empirically verify this guess. Note that the estimation is just an approximation: it is not very mathematically rigorous, and is definitely not a proof.

From equation 3.2 one can see that $d = \sum_{i=0}^{n-1} v_i w_{n-i}$, where for simplicity I call $w_n = w_0$. To do my approximation, I make the (false) assumption that the v_i's are independent of the w_i's. To simplify matters, I use the notation $\mathcal{N}(\mu, \sigma^2)$ within equations to represent a random variable $X \sim \mathcal{N}(\mu, \sigma^2)$. I call D the random distribution of d:

$$D = \sum_{i=0}^{n-1} v_i \mathcal{N}(0, \sigma^2) = \sum_{i=0}^{n-1} \mathcal{N}(0, v_i^2 \sigma^2).$$

An important property of normal distributions is that adding two normal distributions gives another normal distribution with its mean as the sum of the two means and variance the sum of the two variances. Therefore,

$$D = \mathcal{N}(0, \sum_{i=0}^{n-1} v_i^2 \sigma^2)$$

and hence, since we always pick d to be positive (if it is not we negate both d and w),

$$d \approx \sqrt{\sum_{i=0}^{n-1} v_i^2 \sigma^2} \approx \sigma \sqrt{\sum_{i=0}^{n-1} v_i^2}.$$

Recall the v_i have distribution $V \sim \mathcal{U}(-2^{t-1}, 2^{t-1}) = 2^{t-1} \cdot \mathcal{U}(-1, 1)$. Therefore, $\sqrt{\sum_{i=0}^{n-1} v_i^2}$ is $\sqrt{n}$ times the standard variation of V. This itself is $2^{t-1}/\sqrt{3}$. Grouping the constant terms into $K = 1/\sqrt{6}$ this gives

$$\sigma \approx \frac{Kd}{2^t \sqrt{n}}. \tag{4.1}$$

In order to verify this, I computed 100 random keys with $n \in \{2^8, 2^9, \ldots, 2^{12}\}$ and $t \in \{100, 101, \ldots, 500\}$ and measured σ/d. Figure 4.11 shows a plot of $\log_2$ of the expected value against $\log_2$ of this measured value along with a line of best fit.

As one can see, the approximation is quite good. However, the value of K seems to be slightly smaller than expected, which is probably due to the different approximations used. The line appears to intercept the y-axis at -3.03, and I will therefore use the corrected value $K = \sqrt{6}/2^{3.03} = 0.300$ from now on.

4.2.2 Impact on the scheme

In this section, I describe the impact of the w_i's distribution on the somewhat homomorphic scheme. I already discussed in section 3.2 the different understanding that could be taken

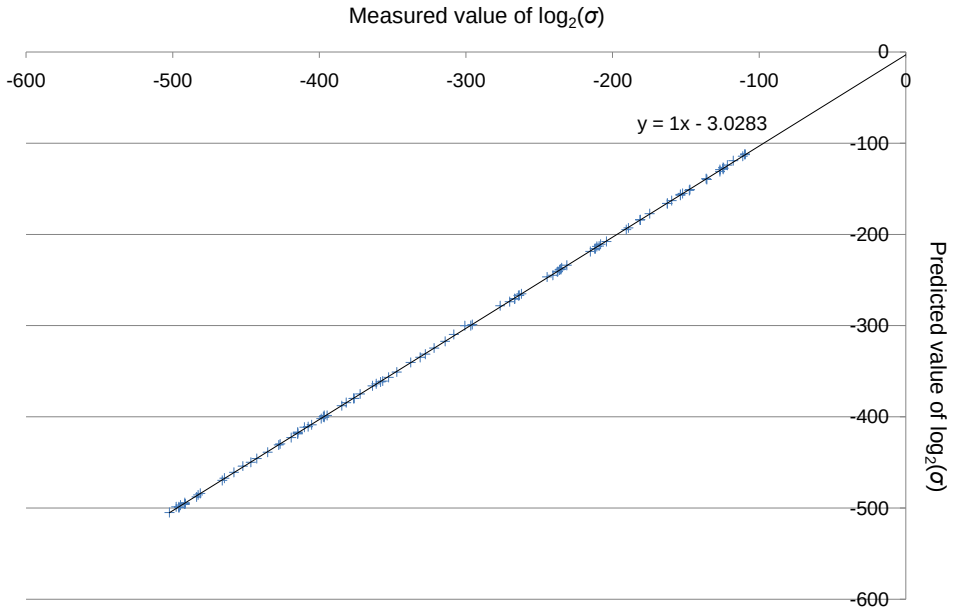

Figure 4.11: Measured value of $\log_2(\sigma)$ *against its predicted value using equation 4.1.*

from this scheme, here I will discuss the maximum degree a ciphertext can have while still being decryptable.

Recall that the ciphertext is still decryptable while $[cw]_d = [m + \sum_{i=0}^{n-1} u_i w_i]_d = m + p\sum_{i=0}^{n-1} u_i w_i$ meaning $-d/2 \leq m + \sum_{i=0}^{n-1} u_i w_i < d/2$ (I inserted the p inside the coefficients u_i for simplification). Now consider the size of the sum $\sum_{i=0}^{n-1} u_i w_i$. Let S be the random variable representing this sum. We know the w_i's are random variables with distribution $W \sim \mathcal{N}(0, \sigma^2)$. This means

$$\begin{aligned} S &= \sum_{i=0}^{n-1} u_i \mathcal{N}(0, \sigma^2) \\ &= \sum_{i=0}^{n-1} \mathcal{N}(0, u_i^2 \sigma^2) \\ &= \mathcal{N}(0, \sum_{i=0}^{n-1} u_i^2 \sigma^2). \end{aligned}$$

Therefore, the value of cw should be about

$$\begin{aligned} cw &\approx \pm\sqrt{\sum_{i=0}^{n-1} u_i^2 \sigma^2} \\ &\approx \pm\sigma\sqrt{\sum_{i=0}^{n-1} u_i^2}. \end{aligned}$$

But since we know $\sigma \approx \frac{Kd}{2^t\sqrt{n}}$,

$$cw \approx \pm\frac{Kd}{2^t\sqrt{n}}\sqrt{\sum_{i=0}^{n-1} u_i^2}.$$

And therefore, $\mid cw \mid < d/2$ whenever $\sqrt{\sum_{i=0}^{n-1} u_i^2} < 2^t\sqrt{n}/2K$. Note that $\sqrt{\sum_{i=0}^{n-1} u_i^2}$ is the Euclidean length of the vector $\vec{u}$. So in order to see what the maximum degree for a set of parameters is, we must see how the Euclidean length $\parallel \vec{u} \parallel$ evolves with the degree.

In [5], the authors used 2^t as an approximation to the decryption radius (which my results suggest to be $2^t\sqrt{n}/2K$). They also assumed that multiplying two polynomials $\vec{c}_1$ and $\vec{c}_2$ in the ring $R = \mathbb{Z}[x]/(f(x))$ gave a polynomial $\vec{c}_3$ with Euclidean length $\parallel \vec{c}_3 \parallel \approx \parallel \vec{c}_1 \parallel \cdot \parallel \vec{c}_2 \parallel$. This means that the product of k polynomials each with Euclidean length c should have Euclidean length of about c^k. In order to test the number of homomorphic operations that could be done on a ciphertext, they computed the elementary symmetric polynomials of a number of encrypted bits and looked at the maximum degree that could still be decrypted. When computing the elementary symmetric polynomial of degree deg on m variables, there are $\binom{m}{deg}$ elements of degree deg being added together. For encrypted bits of Euclidean length c, the expectation of the maximum degree therefore was the largest deg such that $c^{deg} \times \sqrt{\binom{m}{deg}} \leq 2^t$. However, for large values of t – and therefore large values of deg – they noticed that the real maximum degree was slightly larger than the expected one, meaning the Euclidean length was not as large as expected. They attributed this to the fact that the Euclidean length of bits of degree deg was a distribution around c^{deg} and therefore the Euclidean length of their sum differs somewhat from $c^{deg} \times \sqrt{\binom{m}{deg}}$.

However, I tested the maximum degree of a single product and it also appeared to be larger than the expected $c^{deg_{max}} < 2^t\sqrt{n}/2K$. I therefore decided to test the fact that the Euclidean length of a bit is c^{deg}.

In order to test this, I made a small implementation of the multiplications of two vectors $\vec{u}_1$ and $\vec{u}_2$ in the ring of polynomials modulo $x^n + 1$. Recall that this just means the resulting coefficients that would usually be of an order $k > n$ are subtracted from the coefficients $k - n$. As I did in the encryption, I set ε of these coefficients to be ± 1. Here however, I did not multiply the coefficients by p. The reason for this is that the effect on the Euclidean length of multiplying by p is obvious: the coefficients of a vector of degree deg would just all be multiplied by p^{deg}. The initial Euclidean length is therefore $\sqrt{\varepsilon}$. Then for dimensions $n = 2^6, 2^7, \ldots, 2^{10}$ and for each degree $deg = 1, \ldots, 180$, I computed the average Euclidean length of 200 vectors of degree deg. The dimension turned out to only have a very small impact, therefore I only present here the result for $n = 2^{10}$. Figure 4.12 shows the graph of the logarithm of the Euclidean as a function of the degree.

As one can see, the curve is very close to be linear and therefore I will approximate it as being so. Also notice how the gradient $g = 0.908$ of the linear approximation is not quite 1, meaning it is not quite true that $\parallel \vec{c}_3 \parallel \approx \parallel \vec{c}_1 \parallel \cdot \parallel \vec{c}_2 \parallel$ as was suggested in [5]. This means that the expected length of a product of k polynomials each of Euclidean length c is c^{gk}. In appendix A.6, I give an argument to why it is the case that $g < 1$.

Using this result, we can deduce the Euclidean length $\parallel c \parallel$ of a ciphertext after deg multi-

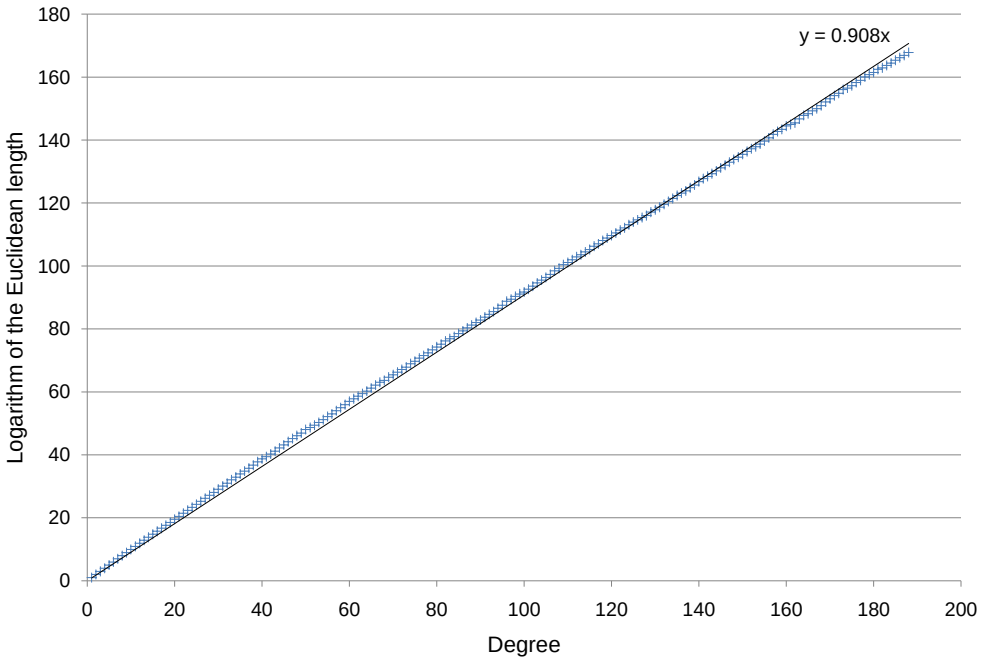

Figure 4.12: Logarithm of the Euclidean length of a polynomial in R against its degree.

plications, it should be $\| c \| = p^{deg} \varepsilon^{g \times deg/2}$. Further, we can use this to find the maximum degree a ciphertext can have before going out of the decryption radius. This is the largest integer deg_{max} such that

$$\begin{aligned} \sigma \| c \| < \frac{d}{2} &\iff \| c \| < 2^t \sqrt{n}/2K \\ &\iff deg_{max}(\log_2(p) + \frac{g}{2}\log_2(\varepsilon)) < t + 1/2\log_2(n) - \log_2(2K). \end{aligned}$$

This last formula is extremely useful. It shows exactly how the different parameters of the somewhat homomorphic scheme interact with each other. Renaming $K' = -\log_2(2K)$ and $g' = g/2$ gives the equation

$$(\log_2(p) + g'\log_2(\varepsilon))deg_{max} \approx t + 1/2\log_2(n) + K' \tag{4.2}$$

where $g' = 0.454$ and $K' = 0.737$ are two experimentally determined constants. This leads to two observations:

- Since n and ε are set from the security of the scheme (which I do not discuss here), p and t are the only parameters which we have complete choice over. p is most likely going to be set depending on what is required from the scheme, for example using $p = 2^{32}$ would be a homomorphic implementation of standard 32-bit int's. This leaves t to be set in function of the deg_{max} we need. Usually, we will often want deg_{max} to be a multiple of the decryption degree deg_{dec}, this way setting $deg_{max} = \kappa\, deg_{dec}$ means one can do $\kappa - 1$ multiplications between two recryptions. t can then be set accordingly.

- The formula shows that the necessary t for a particular deg_{max} varies linearly with p. This shows well how much of an improvement is brought by the large message space I introduced in this project. Using the bit representation, we need $O((\log_2(p))^2)$ multiplications to do one multiplication between two messages modulo p. In the large message space representation, this is done in only one multiplication. There is a cost to increase t in order to keep deg_{max} constant, however as I just said, this only requires a linear increase in t which itself only is a quasi-linear increase in complexity. I therefore expect the large message space to be more and more efficient compared to the bit representation as p increases.

4.3 Performance of my implementation

In this section, I evaluate the performance of the scheme comparing the use of different message spaces, and evaluate the performance of the parallelized operations.

Recall that, depending on the bit size t, we might be able to do more than one multiplication between two recryptions. First, in order to be able to compare the bit representation with the larger message space accurately, I evaluate the average time to do one bit multiplication depending on which t we use. This way, I can use the bit size t that only allows a single multiplication – which is easier to program with – and rescale the results to know how well this would have performed using a larger t. Then, I compare the bit representation with the large message space representation over a number of selected benchmarks. Finally, I evaluate the performance of the parallel, distributed implementation against the non-distributed one.

As mentioned in section 4.1.2.1 I run tests in dimension $n = 2^8$; it is however easy to deduce what the results in higher dimensions would be. The rest of the parameters are deduced from n and t using formulas from [5].

4.3.1 Performance of bit multiplication

Here I evaluate the performance of multiplying two bits using different values for the bit size of the generating polynomials t. The value of t has two impacts:

1. The greater the value of t, the more homomorphic multiplications we can do. As we saw earlier, the maximum degree increases linearly with t.

2. The complexity of the scheme increases with t

In figure 4.13 I plot the amortised time to compute the multiplications of two bits – including the time to recrypt – as a function of t. The reason why the time initially decreases as t increases is because, as t gets larger, we get to do more multiplications between two recryptions. Then, the fact that the complexity of the recryption is slightly higher than linear with respect to t starts having an impact and the multiplication time goes back up. The values of t were chosen such that the k^{th} value is the minimum one using which we can do k multiplications between two recryptions.

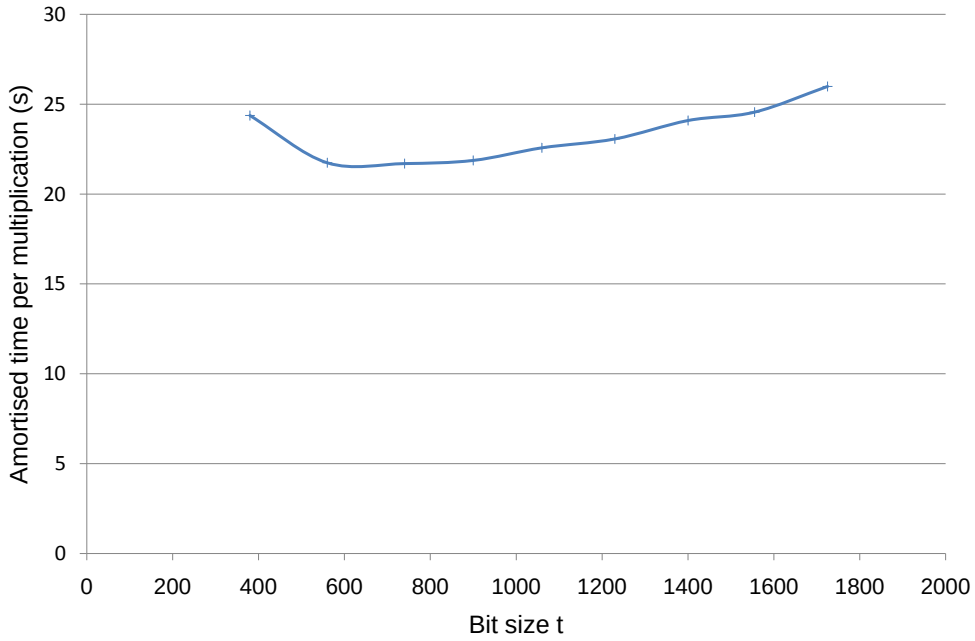

Figure 4.13: Amortised time of bit multiplication.

From the graph, one can see that the optimal value for t in the bit representation is $t = 740$ (time 21.5 sec) allowing three multiplications between every recryption. Because I use $t = 380$ (time 24 sec) allowing one multiplication in my benchmarks, the results will have to be scaled by a factor of about 0.9 in order to be accurate.

4.3.2 Evaluation of the large message space against the bit representation

As I said in section 3.5, there is a tradeoff between using the bit and the large message space representation. In this section, I evaluate the gain in performance brought by the large message space. The large message is useful for two functions: additions and multiplications. I therefore implemented the two following benchmarks:

1. Homomorphic computation of the first 12 Fibonacci numbers, given an encryption of 0 and of 1.
2. Homomorphic computation of $\prod_{i=k}^{k+4} i$, given an encrypted number k.

Note that all operations are done modulo 16. This means I use four bits in the bit representation and large message space in the range $0, \ldots, 15$. Table 4.1 shows the results of the first benchmark.

When using this benchmark on the large message space, the procedure was too fast for the time to be registered. I therefore tried to compute Fibonacci with higher parameters (recall operations are done modulo 16 and therefore the complexity does not scale as it usually

Performance of the bit representation	Scaled performance of the bit representation	Performance of the large message space
12 min 11 sec	10 min 58 sec	< 0.01 sec

Table 4.1: First benchmark: Computation of the 12 first Fibonacci numbers.

would), and found computing time went over the threshold of 0.01 sec around `Fib`(500). This means my implementation of the large message space is about 3×10^6 times faster than the bit representation for this benchmark. The reason for such a drastic improvement is that, because we are not doing any multiplications in the large message space, there is no need to do a recryption. In fact, with the current parameters, and using equation 4.2, one can see that we should be able to do as many as 10^{118} additions before having the same effect on the noise as a single multiplication.

Performance of the bit representation	Scaled performance of the bit representation	Performance of the large message space
18 min 16 sec	16 min 27 sec	2 min 44 sec

Table 4.2: Second benchmark: Computation of $\prod_{i=k}^{k+4} i$.

The results of the second benchmark are in table 4.2. Again, the performance of the large message space was higher than the one of the bit representation. Of course, this is not completely representative. As I mentioned in section 3.5, the bit representation is necessary for certain operations such as doing a right shift or a comparison. But thanks to the recryption to bits, switching to a bit representation can be done with a complexity even lower than the one of recrypting a single integer.

4.3.3 Parallelized performance

In this section, I review the gain in performance of the distributed scheme for the encryption and the recryption. To do this, I ran some Compute Nodes and a Master Node on a single machine, taking advantage of its multiple cores. The details of the machine I was using are included in appendix A.7.

To evaluate the improvement of the encryption, I compare the time to encrypt the secret key for $n = 2^{12}$ and $t = 1000$ as a function of the number of nodes used. I also compute each of the three types of recryptions, again varying the number of nodes. The results are all summarised in Figure 4.14.

As can be seen, parallel computation achieves a significant speedup. For the encryption of the secret key, the running time with 10 nodes was within one percent of the running time of a single node divided by ten. For the recryption procedures, the scaling is not quite as good, but is still very significant.

Finally, to compare the performance of my implementation to the one of [5], I generated a single fully homomorphic key of dimension $n = 2^{15}$ and bit size $t = 380$ which are the largest parameters used to evaluate Gentry's and Halevis's implementation. In [5], this was

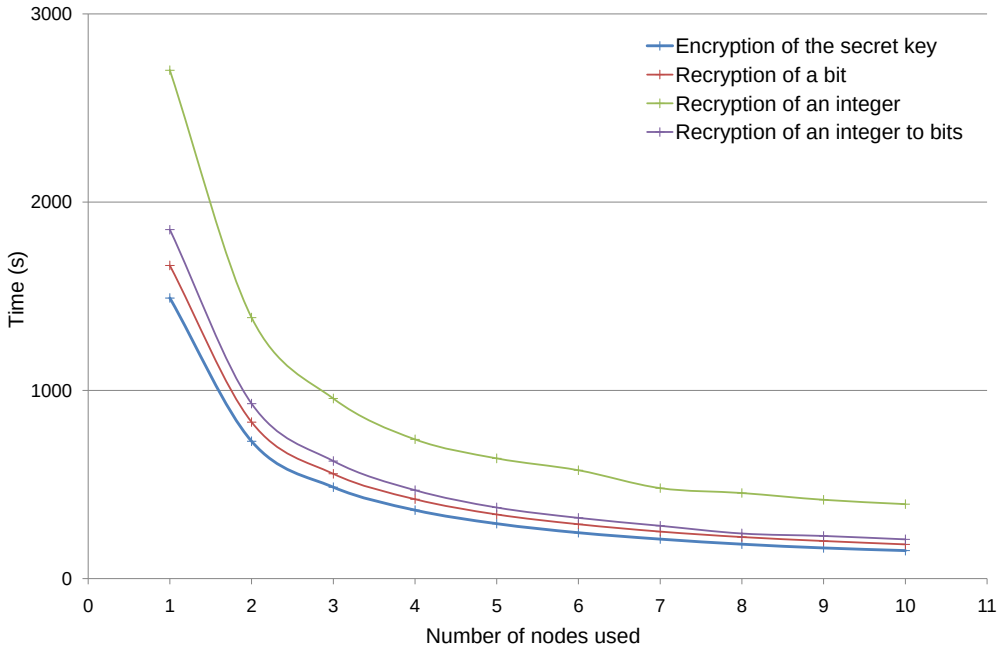

Figure 4.14: Time to compute each of the distributed function against number of nodes used

done in 2.2 hours. I ran this on a machine of approximately the same speed – a decryption, which is a single multiplication, is done in 0.66s on their machine and 0.72s on the one I was using – but with 48 cores, and generated a key of the same size in 11 minutes and 29 seconds using 40 parallel processes. While this is a great start, I expect using the same parallelization mechanisms along with a shared memory approach would have gained another order of magnitude. In this experiment, I was limited by the fact that each process has an individual copy of each power of r that was computed. Therefore, in order to fit within the 64GB of RAM of the machine, I had to reduce the number of powers of r I was computing from the optimal value of 2^{13} down to 2^{10}. This means I computed about 3.6 times more multiplications than were necessary. Of course, my process-based implementation does have the advantage of being able to run on different machines.

I also was able to get an improvement on the recryption time shown in [5]. Using 46 concurrent Compute Nodes, I recrypted a single bit, with the same parameters as before, in 11 minutes, as opposed to 31 minutes in [5].

4.4 Conclusion

In this chapter, I reviewed the evaluation of the implementation described in chapter 3. I started in section 4.1 by unit testing the different functions of the scheme and evaluating their basic performance. I then studied how the somewhat-homomorphic scheme concretely worked in section 4.2. To do this, I first analysed the distribution of the coefficients of the

vector $\vec{w}$. This led to the different understanding of the somewhat homomorphic scheme I had described in section 3.2. Then, I was able to use this result, and after conducting further experiments, demonstrate a general formula for the maximum degree a ciphertext can have, illustrated in equation 4.2. Finally, in section 4.3, I used benchmarks to compare the efficiency of the large message space I introduced in this project, against the "traditional" bit representation. The results showed improvements ranging from significant to tremendous, with a very notable 3×10^6 speedup for the computation of the Fibonacci sequence. I also evaluated the performance of my parallelized implementation, and most notably used it to generate a fully-homomorphic key in under 12 minutes, when the same key was produced in 2.2 hours using Gentry's and Halevi's implementation [5].

Chapter 5

Conclusion

This project concerned the implementation of the first and very recently found homomorphic encryption scheme. The aim was to make the scheme as efficient as possible.

The project was a complete success: not only did I add some improvements to the existing scheme, but I also built a different and simpler understanding of the scheme and how it works. On top of this, I modified the scheme to handle integers instead of bits, with a gain in complexity going from significant to tremendous depending on the applications. Finally, I parallelized the scheme, bringing an even greater performance improvement.

A practical use of the work presented in this dissertation would be in the construction of a platform that can be used to execute homomorphic computations. The platform would have to hide the underlying representation of the variables to the programmer, and manage the recryption operations. Another, more theoretical, future work, would be to study how a different representation of the encryption of the secret key may take advantage of the large message space.

As far as I know, this project breaks new ground in the field of homomorphic encryption, bringing it one step closer to being a viable tool for practical computation.

Bibliography

[1] R.L. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, pages 169–178, 1978.

[2] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178. ACM, 2009.

[3] C. Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.

[4] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography–PKC 2010*, pages 420–443, 2010.

[5] C. Gentry and S. Halevi. Implementing Gentrys fully-homomorphic encryption scheme. *Manuscript*, 2010.

[6] `http://www.i-programmer.info/news/112-theory/2330-darpa-spends-20-million-on-homomorphic-encryption.html`.

[7] D. Micciancio and B. Warinschi. A linear space algorithm for computing the Hermite normal form. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 231–236. ACM, 2001.

[8] V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant. *Automata, Languages and Programming*, pages 144–155, 2006.

[9] N. Gama and P.Q. Nguyen. Predicting lattice reduction. In *Proceedings of 27th annual international conference on Advances in cryptology*, pages 31–51. Springer-Verlag, 2008.

[10] Y.K. Liu, V. Lyubashevsky, and D. Micciancio. On bounded distance decoding for general lattices. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 450–461, 2006.

[11] P. van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Mathematics Department, University of Amsterdam. Technical report, TR 81-04, 1981.

[12] D. Micciancio. Improving lattice based cryptosystems using the Hermite normal form. *Cryptography and Lattices*, pages 126–145, 2001.

[13] V. Shoup. NTL: A library for doing Number Theory. `http://www.shoup.net/ntl/`, 2011. Version 5.5.2.

[14] Boost C++ libraries. `www.boost.org`, 2011. Version 1.46.1.

[15] M.S. Paterson and L.J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2:60, 1973.

[16] H. Daudé and B. Vallée. An upper bound on the average number of iterations of the LLL algorithm. *Theoretical Computer Science*, 123(1):95–115, 1994.

Appendix A

A.1 Rings and Ideals

A *ring* is an ordered triple $(R, +, \cdot)$ consisting of a nonempty set R and two binary operations $+$ and $\cdot$ defined on R such that

- $(R, +)$ is a commutative group and as such:
  1. It is closed under addition:
     For all a, b in R, the result of the operation $a + b$ is also in R.
  2. Addition is associative:
     For all a, b, c in R, the equation $(a + b) + c = a + (b + c)$ holds.
  3. There exists an additive identity:
     There exists an element 0 in R, such that for all elements a in R, the equation $0 + a = a + 0 = a$ holds.
  4. There exists an additive inverse:
     For each a in R, there exists an element $-a$ in R such that $a + -a = -a + a = 0$.
  5. Addition is commutative:
     For all a, b in R, the equation $a + b = b + a$ holds.
- $(R, \cdot)$ is a monoid and as such:
  1. It is closed under multiplication:
     For all a, b in R, the result of the operation $a \cdot b$ is also in R.
  2. Multiplication is associative:
     For all a, b, c in R, the equation $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ holds.
  3. There exists an multiplicative identity:
     There exists an element 1 in R, such that for all elements a in R, the equation $1 \cdot a = a \cdot 1 = a$ holds.
- The operation $\cdot$ is distributive over $+$ meaning:
  1. For all a, b and c in R, the equation $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ holds.
  2. For all a, b and c in R, the equation $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ holds.

A typical example of a ring is the set of all integers $\mathbb{Z}$ with the usual operations of addition and multiplication. Another, less trivial example is the set of integer polynomials with a finite number of coefficients $\mathbb{Z}[x]$, again under the usual operations of addition and multiplication.

Let $(R, +, \cdot)$ be a ring and $S \supseteq R$ a nonempty subset of R. If the system $(S, +, \cdot)$ is itself a ring, then $(S, +, \cdot)$ is said to be a *subring* of $(R, +, \cdot)$.

Using our previous examples again, we can see the ring of integers is a subring of the ring of integer polynomials.

A subring I of the ring R is said to be an *ideal* of R if and only if for all r in R and a in I, both $r \cdot a$ and $a \cdot r$ are also in I.

A good example of an ideal is the set of even integers ($2\mathbb{Z}$) which is an ideal in the ring of integers. This is because the sum of two even integers is even, and the product of any integer with an even integer is also even.

Finally, given a ring R and an ideal I, we define the equivalence relation:

$$a \equiv b \text{ if and only if } a - b \in I$$

where $-$ has the intuitive definition of the addition of the additive inverse. This leads to the definition of an equivalence class:

$$[a] = a + I = \{a + i \mid i \in I\}.$$

The set of all such equivalence classes is denoted by R/I. It is itself a ring called the *quotient ring* of R modulo I.

Modular arithmetic is a typical example of a quotient ring. Take $\mathbb{Z}$, the ring of integers, and $n\mathbb{Z}$, the ring of integers multiples of n: $n\mathbb{Z}$ is an ideal of $\mathbb{Z}$, and the quotient ring $\mathbb{Z}/n\mathbb{Z}$ is a ring with n elements. Modular arithmetic then is arithmetic in the ring $\mathbb{Z}/n\mathbb{Z}$.

A.2 Lattice-based cryptosystems

Here, I describe the lattice problems that are considered hard. I then move on to explaining how these are used within Goldreich-Goldwasser–Halevi (GGH)-type cryptosystems.

Before I actually get into the details, I very briefly summarise how lattices are used in cryptosystems, to give the reader has an idea of what I am aiming for in this section. Given a lattice that was randomly generated, we are going to encrypt data by taking a random point within the lattice and adding to it a small random error vector within which we embed the message. The generated point that is therefore close to, but not in the lattice. The secret and public key are made of "good" and "bad" bases of the lattice. While it is easy to find which point of the lattice is closest to our encrypted point – and therefore the error vector along with the embedded message – using the "good" basis, it is computationally hard to do this using the "bad" basis.

A.2.1 Lattice problems

Here, I am going to show how lattices can be used as tools for our cryptosystems. As I said above, an important property of a "good" basis is that given a vector $\vec{v}$ not in the lattice, we can find the closest vector to it in the lattice. I am first going to go over how this is done.

In order for a basis B to be "good", its vectors $\vec{b}_i$ must be nearly orthogonal to one another. For example, consider the two dimensional lattice and its basis in figure 2.1.

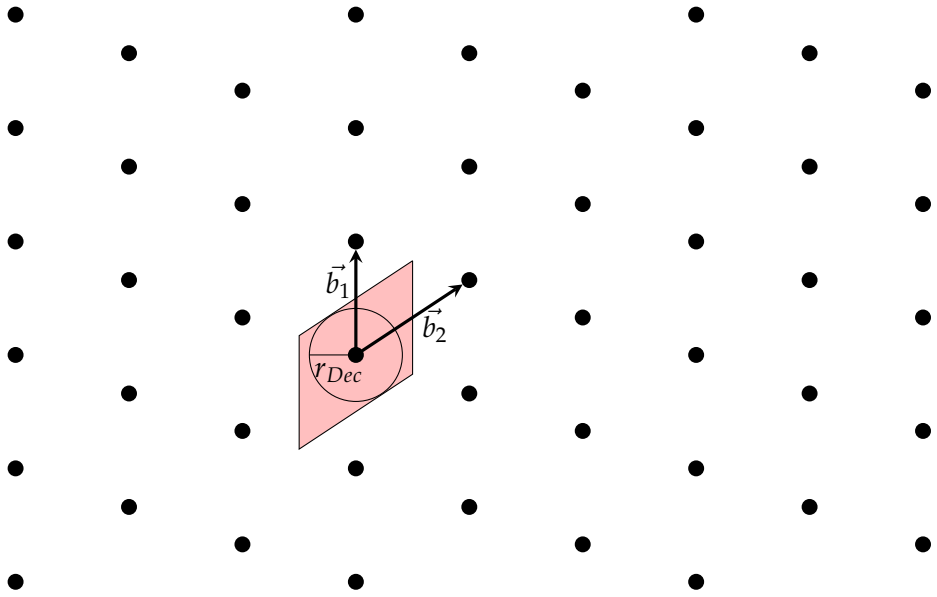

Figure A.1: A "good" basis B and its decryption radius.

As one can see, the two arrows representing the two vectors $\vec{b}_1$ and $\vec{b}_2$ of the "good" basis are close to be orthogonal. Consider the largest circle (n-dimensional "ball") centered at the origin, I call its radius r_{Dec}. Recall that when reducing any vector $\vec{c}$ mod B, we get a vector $\vec{c'}$ in $\mathcal{P}(B)$ (the shaded parallelogram in the figure A.1). In figure A.2, I set the two points $\vec{c_1} = 3\vec{b_2} + \vec{e_1}$ and $\vec{c_2} = 2\vec{b_1} - 2\vec{b_2} + \vec{e_2}$ where the vectors $\vec{e_1}$ and $\vec{e_2}$ are represented by the green and blue arrow respectively. I also set $\vec{c_1'} = \vec{c_1}$ mod B and $\vec{c_2'} = \vec{c_2}$ mod B. Now, for any point $\vec{c} = \vec{x}B + \vec{e}$ if the length (Euclidean norm) of $\vec{e}$ is below r_{Dec}, then $\vec{c}$ mod $B = \vec{e}$. This means that as long as $\vec{c}$ is not too far from the lattice, we can find the error vector using our "good" basis B. Retrieving $\vec{x}$ can then easily be done: $\vec{x} = (\vec{c} - \vec{e})B^{-1}$.

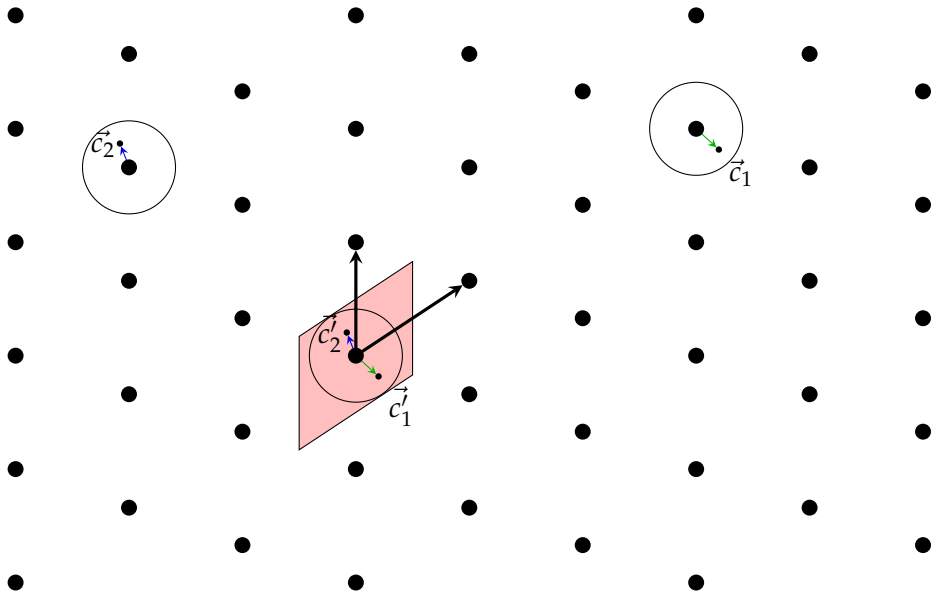

Figure A.2: Reduction of $\vec{c}_1$ and $\vec{c}_2$ modulo B.

Now consider what happens if instead we are using a "bad" basis B', whose vectors are not orthogonal at all, as shown in figure A.3.

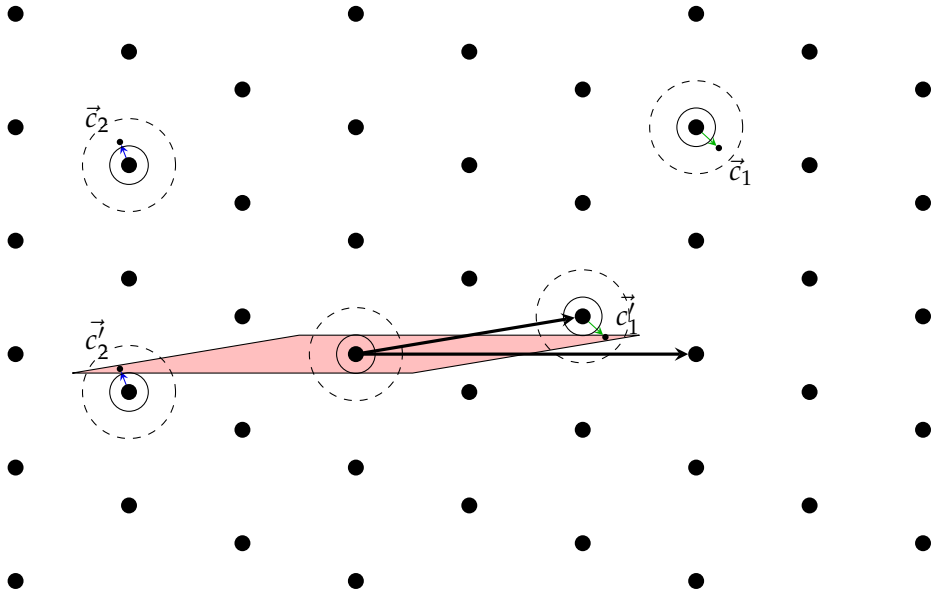

Figure A.3: Reduction of $\vec{c}_1$ and $\vec{c}_2$ modulo a "bad" lattice B'.

For convenience, I added in dashed line the decryption ball of the "good" basis. As one can see, the radius of the decryption ball of B' is much smaller than the one of B. Again, I have set $\vec{c}'_1 = \vec{c}_1 \bmod B'$ and $\vec{c}'_2 = \vec{c}_2 \bmod B'$, but since the parallelogram $\mathcal{P}(B')$ is much thinner than before, $\vec{c}'_1$ and $\vec{c}'_2$ are no longer centered on the origin. For this reason, we no longer have $\vec{c}'_1 \bmod B' = \vec{e}_1$ or $\vec{c}'_2 \bmod B' = \vec{e}_2$. I now describe the complexity of solving this problem

when $\vec{c}$ is further away from the lattice than our decryption radius.

For a lattice L, I denote by $\lambda(L)$ the length of the shortest non-zero vector in L $\min_{\vec{v}\in L} \|\vec{v}\|$. It is easy to see that for any vector $\vec{v}$ (not necessarily in the lattice), there is at most one lattice point within distance $\lambda/2$ from $\vec{v}$.

I denote by $\text{dist}(L, \vec{c})$ the minimum distance between $\vec{c}$ and any point on L ($\min_{\vec{v}\in L}\{\|\vec{c} - \vec{v}\|\}$). In the "*$\alpha$-bounded distance decoding*"' problem (α-BDDP), one is given a basis B of some lattice L, and a vector $\vec{c}$ with the promise that $\text{dist}(L, \vec{c}) < \alpha\lambda(L)$ and is asked to find the point in L nearest to $\vec{c}$. This problem has been shown to be NP-hard for $\alpha > 1/\sqrt{2}$ [10], it is however an open problem whether it is NP-hard for smaller α. Extensive experiments over the true hardness of the problem can be found in [9].

Another lattice problem that will be useful for our cryptosystems is the *shortest vector problem* (SVP). Given a lattice $\mathcal{L}(B)$, find the length of the shortest non-zero vector $\lambda(\mathcal{L}(B))$. This problem has been shown to be NP-hard [11].

A.2.2 GGH-type Cryptosystems

Using the tools from the previous section, we can now see how we are going to be able to build a cryptosystem using lattices. Here, I briefly describe how GGH cryptosystems work as explained in [12]. For some lattice L, the secret is made of a "good" basis B_{sk} of L, while the public key is made of a "bad" basis B_{pk} of L. First start by generating B_{sk}. Recall that we want the decryption radius to be as large as possible. For this to happen, we need our basis vectors to be nearly orthogonal. However, choosing each coefficient of each vector as a uniform random value has shown to give good enough results [16]. For the public key, we want a basis with a particularly short decryption radius. For this reason, we choose the Hermite normal form of our lattice $B_{pk} = \text{HNF}(\mathcal{L}(B_{sk}))$, which is in some sense the least revealing basis of L. Note that given B_{pk}, finding B_{sk} requires some instances of the SVP. To encrypt a ciphertext m, the sender chooses a short error vector $\vec{e}$ within which m is encoded. The length of $\vec{e}$ must be higher than the decryption radius of B_{pk}, but lower than the decryption radius of B_{sk}. The sender then needs to pick a random point $\vec{x}$ within the lattice so that the message can be encrypted as $c = \vec{x}B_{pk} + \vec{e}$. It has been shown that rather than generating this $\vec{x}$ separately, it was enough to just take the modulo of $\vec{e}$ with the public key $c = \vec{e} \bmod B_{pk}$. In order to decrypt, the receiver needs to extract the error vector from $\vec{c}$. This can be done as mentioned above by setting $\vec{e} = \vec{c} \bmod B_{sk}$. The message m can then be retrieved from $\vec{e}$.

A.3 Mathematical demonstrations used in the implementation of Gentry's scheme

In this section, I detail some of the mathematical demonstrations I used the results of in section 3.1.

A.3.1 Decryption

We have a ciphertext $\vec{c} = \langle c, 0 \ldots, 0 \rangle$ and we want to find the short error vector $\vec{a} = 2\vec{u} + b \cdot \vec{e_1}$. We know we can retrieve $\vec{a}$ by doing

$$\begin{aligned} \vec{a} &= \vec{c} \bmod V \\ &= [\vec{c} \times W/d] \times V. \end{aligned}$$

Remember that as long as $\vec{c}$ is close enough to $\mathcal{L}(V)$, this procedure should set $\vec{a}$ to the distance to the closest point in the lattice. This means that we have $\vec{c} = \vec{y} \times V + \vec{a}$ for some integer vector $\vec{y}$, where $\vec{y} \times V \in \mathcal{L}(V)$ is the closest point to $\vec{c}$. Therefore we have

$$\begin{aligned} \vec{a} &= [\vec{c} \times W/d] \times V \\ &= [\vec{y} \times V \times W/d + \vec{a} \times W/d] \times V \\ &= [\vec{y} + \vec{a} \times W/d] \times V. \end{aligned}$$

Because $\vec{y}$ is an integer vector, its fractional part $[\vec{y}]$ should be 0. Therefore

$$\begin{aligned} &= [\vec{a} \times W/d] \times V \\ &= (\vec{a} \times W/d) \times V. \end{aligned}$$

The last line simply follows from the fact that $W/d \times V = I$; this shows that the procedure works as long as we have $[\vec{a} \times W/d] = \vec{a} \times W/d$. This is true as long as every entry of the vector $\vec{a} \times W/d$ has an absolute value below $1/2$. Equating the first and last line and removing V shows that the procedure will work as long as $[\vec{c} \times W/d] = \vec{a} \times W/d$. Since $[\vec{c} \times W/d] = [\vec{c} \times W]_d/d$, multiplying both sides by d gives:

$$[\vec{c} \times W]_d = \vec{a} \times W$$

Recall that the notation $[]_d$ means modulo d in the interval $[-d/2, d/2)$. Using this fact, we can greatly simplify the decryption procedure. First, we expand the left hand side, using the fact that $\vec{c} = \langle c, 0, \ldots, 0 \rangle$:

$$[\vec{c} \times W]_d = [c \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle]_d = \langle [cw_0]_d, [cw_1]_d, \ldots, [cw_{n-1}]_d \rangle.$$

Then expand the right hand side using $\vec{a} = 2\vec{u} + b\vec{e_1}$:

$$\vec{a} \times W = 2\vec{u} \times W + b \cdot \vec{e_1} \times W = 2\vec{u} \times W + b \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle.$$

Equating these two results, we get that any decryptable ciphertext c must satisfy

$$\begin{aligned} \langle [cw_0]_d, [cw_1]_d, \ldots, [cw_{n-1}]_d \rangle &= b \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle \pmod 2 \\ [cw_i]_d &= b \cdot w_i \pmod 2. \end{aligned}$$

A.3.2 Key Generation

A.3.2.1 Finding w and d

We know that

$$w(x) \times v(x) = d' \pmod{f_n(x)}$$

where d' is a constant. Since $f_n(x) = x^n + 1$, this means we have $d' = v_0 \times w_0 - \sum_{i=1}^{n-1} v_i w_{n-i}$. This is because the 0^{th}-degree coefficient of the polynomial $w(x) \times v(x) \bmod x^n + 1$ is the 0^{th}-degree coefficient of $w(x) \times v(x)$ minus the n^{th} degree coefficient of $w(x) \times v(x)$. Using the same logic for coefficients of different degrees, this means we have for any $k < n$, $\sum_{i+j=k} v_i w_j - \sum_{i+j=k+n} v_i w_j = 0$. Now these equations can be rewritten into matrix form:

$$\begin{bmatrix} v_0 & v_1 & v_2 & & v_{n-1} \\ -v_{n-1} & v_0 & v_1 & & v_{n-2} \\ -v_{n-2} & -v_{n-1} & v_0 & & v_{n-3} \\ & & & \ddots & \\ -v_1 & -v_2 & -v_3 & & v_0 \end{bmatrix} \times \begin{bmatrix} w_0 & w_1 & w_2 & & w_{n-1} \\ -w_{n-1} & w_0 & w_1 & & w_{n-2} \\ -w_{n-2} & -w_{n-1} & w_0 & & w_{n-3} \\ & & & \ddots & \\ -w_1 & -w_2 & -w_3 & & w_0 \end{bmatrix} = \begin{bmatrix} d' & 0 & 0 & & 0 \\ 0 & d' & 0 & & 0 \\ 0 & 0 & d' & & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & & d' \end{bmatrix}.$$

As can be seen, the two matrices on the left hand side are the two rotation bases V and W, associated with $\vec{v}$ and $\vec{w}$ respectively. Since the matrix on the right hand side is $d' \cdot I$, this means W is the scaled inverse of V and d' is the determinant of V.

A.3.2.2 Finding r

Recall that the second line of the HNF of our lattice has the form $\vec{r} = \langle -r, 1, 0, \ldots, 0 \rangle$. Hence $\vec{r}$ must be a point in our lattice, and therefore there must exist an integer vector $\vec{y}$ such that $\vec{y} \times V = \langle -r, 1, 0, \ldots, 0 \rangle$. Multiplying both sides by W we get:

$$\begin{aligned} \vec{y} \times V \times W &= \langle -r, 1, 0, \ldots, 0 \rangle \times W \\ \vec{y} \times d \cdot I &= d \cdot \vec{y} \\ &= -r \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle + \langle -w_{n-1}, w_0, \ldots, w_{n-2} \rangle. \end{aligned}$$

Reducing this modulo d, this means:

$$\begin{aligned} -r \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle + \langle -w_{n-1}, w_0, \ldots, w_{n-2} \rangle &= 0 \pmod{d} \\ \langle -w_{n-1}, w_0, \ldots, w_{n-2} \rangle &= r \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle \pmod{d}. \end{aligned}$$

A.4 Summing the large message space z_i's

In this section, I describe how to sum the fractional numbers z_i, with the representation described in section 3.3.

I first go over the procedure to sum two z_i's.

Consider z_1 and z_2, two fractional numbers with the representation I discussed in section 3.3. That is, for a fractional number z_i, one integer modulo p to represent the integer parts, which I will call z_i^{int}; and a vector of $s+1$ integers modulo p, s of which are 0 and one of which is 1, which I will call $\langle z_i^0, z_i^1, \ldots, z_i^s \rangle$. Note that if z_i^k is 1, the number represented by z_i is $z_i^{int} + \frac{k}{s+1}$. We want to compute $z_3 = z_1 + z_2$. To do this, we first compute the product of each pair of the form z_1^k and z_2^l. That is, the matrix

$$P = \begin{bmatrix} z_1^0 z_2^0 & z_1^0 z_2^1 & z_1^0 z_2^2 & & z_1^0 z_2^s \\ z_1^1 z_2^0 & z_1^1 z_2^1 & z_1^1 z_2^2 & & z_1^1 z_2^s \\ z_1^2 z_2^0 & z_1^2 z_2^1 & z_1^2 z_2^2 & & z_1^2 z_2^s \\ & & & \ddots & \\ z_1^s z_2^0 & z_1^s z_2^1 & z_1^s z_2^2 & & z_1^s z_2^s \end{bmatrix}.$$

Note that only one cell in the whole matrix is non-zero. Then we associate to each hot bit z_3^k in z_3 the cells for which if it is 1, then so is z_3^k; then z_3^k is the sum of these cells. Referring to the members of P as $p_{i,j}$, this gives $z_3^k = \sum_{i+j=k \pmod p} p_{i,j}$. For example, z_3^s is the sum of the diagonal going from bottom left to top right. Finally, we need to compute the integer part z_3^{int}. This should be the sum of the integer parts of z_1 and z_2, plus possibly an extra 1 if the sum of the fractional parts overflew. Therefore, we compute it as $z_3^{int} = z_1^{int} + z_2^{int} + \sum_{i+j>s} p_{i,j}$.

The sum $\sum_{i=1}^{s} z_i$ can then be computed by adding the z_i's to one another. However, in order to minimise the complexity of the decryption, I did not implement this by doing a straightforward linear addition.

The reason for this is that every time we do an addition, the degree of the ciphertexts in the matrix P has an impact on the degree of the integer part of the result. Therefore, since there is a fixed number of additions required to be done, we want to minimise the degree of P for every addition. To do this, I implemented the addition in a "knock-out" fashion, which makes sure that, whenever possible, the additions are performed on numbers of the same degree.

A.5 A single grade-school addition for the recryption to bits

In this section, I detail how to implement the recryption to bits in a single GSA.

In order to fit the procedure in a single GSA, we need to find a way to handle the subtraction. I do this by replacing the subtraction by two additions. Recall that in the decryption all

operations are implicitly done modulo p. Therefore

$$\begin{aligned}
\mathsf{Dec}(c) &= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) - \left\lceil\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rceil \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) + p - \left\lceil\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rceil \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) + p - s + s - \left\lceil\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rceil \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) + p - s + \left\lceil s - \sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rceil \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) + p - s + \left\lceil\sum_{i=1}^{s}1 - \sum_{j=0}^{S-1}\sigma_{i,j}z_{i,j}\right\rceil \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}\langle y_{i,j}\rangle_p\right) + p - s + \left\lceil\sum_{i=1}^{s}\sum_{j=0}^{S-1}\sigma_{i,j}(1 - z_{i,j})\right\rceil .
\end{aligned}$$

The last line comes form the fact that there only is one non-zero $\sigma_{i,j}$ per block. This means we can put both sums into one GSA by instead of encoding $z_{i,j}$ into bits, encoding $1 - z_{i,j}$ and adding into the GSA $p - s$. Note that since we know $p - s$, we can use its bit values inside the GSA along with the encrypted bit values of the other sums.

A.6 Size of the coefficient g

In this section, I give an explanation to why it should be that the size of g found in section 4.2.2 is less than 1.

Consider the two vectors $\vec{a}$ and $\vec{b}$. The multiplication in the ring of polynomials modulo $x^n + 1$ can be rewritten in the following way:

$$\vec{c} = \begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_{n-1} \end{bmatrix} \times \begin{bmatrix} b_0 & b_1 & b_2 & & b_{n-1} \\ -b_{n-1} & b_0 & b_1 & & b_{n-2} \\ -b_{n-2} & -b_{n-1} & b_0 & & b_{n-3} \\ & & & \ddots & \\ -b_1 & -b_2 & -b_3 & & b_0 \end{bmatrix}$$

where $\times$ is matrix multiplication. Consider the Euclidean length $\| \vec{c} \|$ of the vector $\vec{c}$. Notice how, given a coefficient a_i of $\vec{a}$ and a coefficient b_j of $\vec{b}$, the product $a_i b_j$ appears exactly once in $\vec{c}$. Therefore

$$\begin{aligned}
\| \vec{c} \| &= \sqrt{c_0^2 + c_1^2 + \dots + c_{n-1}^2} \\
&= \sqrt{a_0^2 b_0^2 + a_0^2 b_1^2 + \dots a_0^2 b_{n-1}^2 + a_1^2 b_0^2 + \dots + a_{n-1}^2 b_{n-1}^2 + 2\sum_{i,j,k,l} a_i a_j b_k b_l}
\end{aligned}$$

where the sum is the one obtained from the multiplications of different terms within a single coefficient c_m. Factorising this gives

$$\begin{aligned} \| \vec{c} \| &= \sqrt{(a_0^2 + a_1^2 + \dots a_{n-1}^2)(b_0^2 + b_1^2 + \dots b_{n-1}^2) + 2 \sum_{i,j,k,l} a_i a_j b_k b_l} \\ &= \sqrt{\| \vec{a} \|^2 \cdot \| \vec{b} \|^2 + 2 \sum_{i,j,k,l} a_i a_j b_k b_l}. \end{aligned}$$

Now, because the initial distribution of the coefficients of the polynomials is centered on 0, and since we only use additions and multiplications with coefficients of different polynomials to calculate the coefficients of polynomials of higher degree, it must be that the distribution of the coefficients of a polynomial of any degree is centered around 0. Notice how in the construction of the sum $\sum_{i,j,k,l} a_i a_j b_k b_l$, it must be that $i \neq j$ and $k \neq l$, meaning that there is no squaring involved. Therefore the distribution of the sum must be itself centered around 0. Since we take the square root of $\| \vec{a} \|^2 \cdot \| \vec{b} \|^2$ plus some random value centered around 0, it must be that, on average, the result is slightly less than $\| \vec{a} \| \cdot \| \vec{b} \|$.

A.7 Specifications of evaluation machines

`ganymede`: used in sections 4.1 – 4.3.2

- 2x Intel Core 2 Duo E6600, 2.4 GHz, 4 MB L2 cache, FSB 1066
- 8 GB DDR2-RAM,
- 3.8 GB swap partition,
- Running Ubuntu 10.04.2 LTS, 64-bit, kernel version 2.6.32-25-server.

`tigger`: used in section 4.3.3

- 12x AMD Opteron 6168 (quad-core), 12x 512 MB L2 cache, 2x 6 MB L3 cache, 3.2 GHz HyperTransport,
- 64 GB DDR3-RAM,
- 38 GB swap partition,
- Running Debian 7.0 testing (wheezy), 64- bit, kernel version 2.6.32-5-amd64.

Appendix B

Proposal

Valentin Dalibard
St John's College
vd241

Computer Science Tripos Part II Project Proposal

Implementing Homomorphic Encryption

October 14, 2010

Project Originator: Malte Schwarzkopf

Resources Required: See attached Project Resource Form

Project Supervisor: *Malte Schwarzkopf*

Signature:

Director of Studies: *Robert Mullins*

Signature:

Overseers: *Ann Copestake* and *Robert Harle*

Signatures:

Introduction and Description of the work

Encryption schemes that support operations on encrypted data have a wide range of applications in cryptography. The concept of "homomorphic encryption" was first introduced by Rivest et al. shortly after the discovery of public key cryptography [3], and many known public-key cryptosystems such as unpadded RSA or ElGama, support either addition or multiplication of encrypted data. However, it was only in 2009 that Craig Gentry established the feasibility of a cryptosystem supporting both operations securely ("fully homomorphic") [1].

A good metaphor for a homomorphic system is the one of a jewelery shop [2]. Alice, the shop owner, has raw precious material -gold, diamonds, silver, etc.- that she wants her worker to assemble into intricately designed rings and necklaces. But she distrusts her workers, and is afraid that they will steal her jewels if given the opportunity. In other words, she wants her workers to *process* the materials without having *access* to them. What does she do?

Here is her plan. She uses a transparent impenetrable glovebox, secured by a lock for which only she has the key. She puts the raw precious materials inside the box, locks it, and gives it to a worker. Using the gloves, the worker assembles the ring or necklace inside the box, but since the it is impenetrable, he can not get to the materials inside. When finished, the worker hands the box back to Alice who is now able to unlock it with her key and extract the ring or necklace.

Here, the materials represent the data that needs to be encrypted, while the box represent the encryption of this data. What makes this box special, aside from being impenetrable like other secure cryptosystems, is the fact that it has gloves allowing the processing of the data without accessing it.

Since Gentry's paper, a lot of work has been done towards implementing the homomorphic scheme. The principal challenge comes from the fact that the complexity of generating grows very steeply with key size, making it almost impossible to do without carefully choosing a number of optimisations. The first attempt to implement Gentry's scheme was made in 2010 by Smart and Vercauteren [4]. They managed to implement the general procedure but were unable to execute it for keys of a decent size (above 2^{12}). This was later followed by Gentry et al. who built on this work and after multiple optimisations generated a 2^{15} size key in 2.2 hours.

The aim of this project is to produce an open source implementation of Gentry's cryptosystem using the most recently published schemes, and show that it works with small examples.

Brief Description of Gentry's Cryptosystem

Imagine you have an encryption scheme with a "noise parameter" attached to every ciphertext. Encryption creates a ciphertext with a small noise, but for De-

cryption to work, the noise must be smaller than some threshold. Furthermore, imagine you have algorithms **Add** and **Mult** that can take ciphertexts $E(a)$ and $E(b)$ and compute $E(a + b)$ and $E(a * b)$, but at the cost of adding or multiplying the noise parameters. This immediately give a 'somewhat homomorphic' encryption scheme, that can handle computations with a small enough number of operations for the ciphertext to be decrypted.

Now suppose that you have an algorithm **Recrypt**, that take a ciphertext $E(a)$ with a noise below the decryption threshold N, and outputs a fresh ciphertext $E(a)$ that also encrypts a, but has a noise parameter smaller than $\sqrt{N}$. This **Recrypt** algorithm is enough to construct a fully homomorphic scheme out of our somewhat homomorphic one! Anytime we want to **Add** or **Mult** $E(a)$ and $E(b)$, we can apply Recrypt to both $E(a)$ and $E(b)$ ensuring the noise parameter of $E(a * b)$ is smaller than N, and so on recursively.

Gentry's cryptosystem follows this exact pattern. An implementation therefore requires two main parts. First, the construction of a somewhat homomorphic scheme that lets us use the homomorphic operators a certain number of times. Then, the implementation of a "boostraping" transformation to obtain a fully homomorphic scheme.

Resources Required

This project will be coded in C++ in order to make use of the high performance libraries available. There are no special resources required. I am planning to undertake development on my personal machine and the PWF computers with regular backups on the PWF servers.

Starting Point

In proposing this project, I am planning to build on the following resources:

- **Internship at Birmingham University Summer 2010** – I did a research project with the Birmingham University Computer Science department on Evolutionary Computation. Other than increasing my programming and research skill, the project required a good understanding of Fast Fourier Transforms which turn out to be a necessary tool to implement homomorphic operations.
- **Previous programming experience** – I am familiar with C++ and have already implemented small projects in it, but my understanding does not go past the Part IB lecture course.

Substance and Structure of the Project

This project will follow the same structure as the existent implementations of homomorphic encryption. There are therefore two central stages each consisting of a number of sub-tasks that need to be completed:

1. *Somewhat Homomorphic scheme*: This step will create a cryptosystem able to operate a few number of operations on the encrypted data, it implementation will consist of the following functions:

    - **KeyGen**(): Generates a public key **PK** and a secret key **SK** that can be used by the other functions.
    - **Encrypt**(M,**PK**): Encrypts the data M and outputs its cipertext c.
    - **Decrypt**(c,**SK**): Decrypts ciphertext c and outputs the message M that it contains.
    - **Add**(c_1, c_2, **PK**): Adds the contents of ciphertext c_1 and c_2 into ciphertext c_3 and output c_3.
    - **Mult**(c_1, c_2, **PK**): Multiplies the contents of ciphertext c_1 and c_2 into ciphertext c_3 and output c_3.

2. *Fully Homomorphic scheme*: This step will let us build a new cryptosystem from the somewhat homomorphic one. The new cryptosystem will support the **Recrypt** function that decreases the noise in the cipertext. It will involve the following tasks:

    - Squashing the Decryption Procedure: In order to rencrypt our ciphertext, we need to be able to apply the **Decrypt** function homomorphically. Details can be found in [1] but one of the main challenges of homomorphic encryption is to make the decryption procedure simple enough, so that it can be applied to the ciphertext (in the same way Add and Mult are applied) without making the noise parameter of the ciphertext go over the threshold.
    - **Recrypt**(c,PK): Takes a "dirty" ciphertext c and produces a "cleaner" ciphertext c_{new} of the same message but with less errors.

Possible Extension

A number of the procedures that need to be implemented appear to be highly parallelisable. A possible extension for this project will to attempt to parallelise the more costly procedures such as **KeyGen**, and **Recrypt**.

Success Criteria

The following primary criteria should be achieved:

1. A somewhat homomorphic scheme that supports some number of operations on encrypted data has been implemented
2. A decryption squashing procedure under which the decryption has a low enough complexity to be computed by the somewhat homomorphic scheme has been implemented.
3. A re-encrypting function enabling any function to be computed homomorphically on encrypted data has been devised.

   As in the recently published schemes, the data to be encrypted will be a single bit. Generalising the encryption over binary polynomial is considered as an extension.

These possible extensions would increase the success of the project but is not essential:

1. Parallelising the more computationally expensive functions such as **KeyGen**, and **Recrypt**
2. Extend the encryption to work over binary polynomials.

Evaluation

The different parts of the project such as key generation or encryption will be evaluated individually and functions that will be optimised throughout the project will be compared with their naive implementation. The set of parameters used will initially be the one used in published papers, however, I may vary some of the parameters and compare the results with the ones of existing implementations.

Timetable and Milestones

There are thirty three weeks available to do this project. I aim to finishing the implementation work by the end of the 24^{th} week, i.e. the beginning of the Easter vacation.
I estimate that the work will be partitioned into three weeks slots as follow:

Slot 0: October, 1^{th} – October, 22^{nd}

- Familiarisation with the necessary math tools.

- Find appropriate platform to use, most likely using the LAPACK and BLAS libraries.
- Understand the general structure of the recently published workschemes.

Milestone: Write project proposal.

Slot 1: October, 23^{rd} – November, 12^{th}

- More research and learning. Familiarise with the literature on the subject Have a clear understanding of the working schemes.
- Familiarise with the libraries used, attempt to execute short programs using the mathematical structures that will be used.

Slot 2: November, 13^{th} – December, 3^{rd}

- Implement a somewhat homomorphic scheme.
- Implement unit testing for each of the somewhat homomorphic functions.

Milestone: Have a working somewhat homomorphic scheme that passes unit tests.

Slot 3: December, 4^{th} – December, 24^{th}

- Start implementing the decryption squashing procedure.
- Keep on writing auxiliary code to test and debug the future parts of the project.

Slot 4: December, 25^{th} – January, 14^{th}

- Finish implementing the decryption squashing procedure.
- Implement the **Recrypt** function.
- Test and debug further progress using previously written unit testing.

Milestone: Have a first working implementation.

Slot 5: January, 15^{th} – February, 4^{th}

- Optimise previously written code using the tricks described in implementation report [**?**].
- Keep unit testing any modified piece of code.

- Write progress report and hand it in.

Milestone: Present progress, have an improvement in performance from the initial naive implementation.

Slot 6: February, 5^{th} – February, 25^{th}

- Start implementing the optional extension to the project.
- Investigate further extension possibilities that might have arisen during development and possibly implement extra features.

Slot 7: February, 26^{th} – March, 18^{th}

- Continue implementing optional extensions to the project.
- Fix remaining bugs and look for possible code optimisations.
- Start writing the dissertation by producing an outline and writing the introductory sections.

Slot 8: March, 19^{th} – April, 8^{th}

- Buffer time to catch up any delay incurred at previous stages.
- Generate figures and results that will be included in the dissertation.
- Write main parts of the dissertation and produce a draft to be handed in.

Milestone: Hand in draft of the dissertation.

Slot 9: April, 8^{th} – April, 29^{th}

- Finish writing dissertation. Generate any output from the code not yet produced.
- Incorporate comments on the draft received from supervisor and proofreaders.

Slot 10: April, 30^{th} – May, 20^{th}

- Buffer time to address any final issues with the dissertation.
- Use remaining time for exam revision.

Milestone: Submission of dissertation.

References

[1] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[2] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[3] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. pages 169–177. Academic Press, 1978.

[4] N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. Cryptology ePrint Archive, Report 2009/571, 2009. http://eprint.iacr.org/.