A study of the general number field sieve and a development of a CT2 plug-in using YAFU

Querejeta Azurmendi, I.

Award date:
2016

A study of the General Number Field Sieve and a development of a CT2 plug-in using YAFU

A thesis submitted for the degree of Master of Science.

Technical University of Eindhoven

Department of Mathematics and Computer Science

Author:
Iñigo Querejeta Azurmendi
0870120

Supervisors:
Prof. Dr. Tanja Lange
Henry de Valence

Advisors:
Cristina Balasoiu
Prof. Bernhard Esslinger
Armin Krauss

July 28, 2016

Abstract

The Number Field Sieve (NFS) is the fastest known algorithm for factoring general numbers having more than 100 decimal digits. This thesis will cover the theory behind the algorithm, going through different fields of mathematics, such as complex analysis, algebraic number theory, or theory of ideals. We present a plug-in developed for the CrypTool 2 (CT2) application. Our plug-in makes use of an already existing implementation (YAFU) of the number field sieve and finally we present some comparisons with the previously existing factoring plug-ins.

Acknowledgements

I would like to express my greatest appreciation to the people who have helped and supported me throughout my master project.

To my supervisor Prof. Dr. Tanja Lange for introducing me to CrypTool 2 and making possible an Internship abroad. Many thanks as well for the guidance during my project and the last comments in the proofreading task.

To Prof. Bernhard Esslinger. First of all for accepting me as part of this project. It is now with great pleasure that I have taken part in the CrypTool 2 group. Secondly, the critical questions and guidance throughout the thesis have been of great help as well as the time spent for feedback.

To Armin Krauss for your constant support in the development of the plug-in. Without your participation and input, the plug-in could not have been successfully implemented in that quality.

To Henry de Valence for your patience and time spent in task of proofreading.

Last, but not least, I would like to thank my family and friends for giving me unconditional support, and for having made this experience of the master as fulfilling as it has been.

Iñigo Querejeta
Eindhoven
July, 2016

Contents

List of Figures

List of Variables

$n = p \times q$	Number to be factored.
e	RSA public key. Chapter 2 onwards, exponents of factorization.
d	RSA private key. Chapter 2 onwards, degree of a polynomial.
ϕ	Eulers phi function. Chapter 2 onwards natural homomorphism $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$.
x, y	Difference of squares $x^2 \equiv y^2 \pmod{n}$.
$\mathcal{F} = \{p_1, \ldots, p_S\}$	Factor base of small primes.
$S = \lvert\mathcal{F}\rvert$	Size of the factor base.
B	Factor base bound, $p_S \leq B$.
U	Set of relations.
$\lvert U\rvert$	Size of set U.
e_i	Exponents of primes in factorization.
v_i	Binary vector related to factorization.
L	Number of relations found.
$\overline{\mathbb{Q}}$	Subfield consisting of algebraic numbers.
$\mathfrak{O}$	Set of all algebraic integers.
$\mathfrak{O}_\alpha$	Set of algebraic integers in $\mathbb{Q}(\alpha)$.
D	Dedekind domain.
m	Root of polynomial f modulo n.
(a, b)	Relations, such that $(a + b\alpha)$ and $a + bm$ are smooth.
$f(t) \in \mathbb{Z}[t]$	Monic irreducible polynomial.
$f'(t)$	Derivative of f.
α	Root of minimal polynomial.
α_i	Different roots of $f(t)$.
σ_i	Embeddings such that $\sigma_i(\alpha) = \alpha_i$.
$N(a, b)$	Norm of $(a + b\alpha)$.
$F(X, Y)$	Homogenized polynomial of f.
(r, p)	Pair representing degree one prime ideals.
$l_{p,r}(a + b\alpha)$	Power of prime ideal (r, p) such that $a + br \equiv 0 \pmod{p}$.
$\chi_{\mathfrak{q}}$	Quadratic characters.
$\mathcal{A}$	Sieving region.
$[-u, u]$	Region from which we choose the values of (a, b).

Chapter 1

Introduction

For centuries two problems in number theory have kept many mathematicians busy: determining if a given integer is prime and expressing a given whole number as a product of primes. Now, the first problem is considered to be solved using probabilistic algorithms [8], mainly because of their computational speed compared to the available deterministic algorithms (despite the fact that it is in P). Even if the second one might seem a straightforward problem for reasonably small numbers, it becomes complicated when we consider huge numbers. In fact, the difficulty of this problem is what gave rise to the well known RSA (Rivest-Shamir-Adleman) cryptosystem [53], the first public-key cryptosystem, consisting of two keys for every user, one private and one public, and offering a secure communication over an insecure channel, such as the Internet. Still today, RSA is one of the standards of secure electronic communication and is also used for digital signatures, so the interest to factorization remains one of the main topics of discrete mathematics. When we talk about RSA, we must specify that the security comes with the usage of at least 1024-bit composite numbers of the form $n = p \times q$ where p and q are primes of roughly the same size. This means that we are talking about numbers with ≈ 310 decimal digits. This number, whose factorization is unknown, together with a number $e \in \mathbb{Z}/n\mathbb{Z}$ is what forms the public key. The private key, $d \in \mathbb{Z}/n\mathbb{Z}$, is then generated by calculating the inverse of $e \mod (\phi(n))$ where ϕ is Euler's phi function, calculated by $\phi(n) = (p-1)(q-1)$. Therefore, if we manage to factor number n we can calculate the private key and the cryptosystem is broken.
Already in 1999 a 512-bit key was factored [21], but surprisingly, regardless of this factorization, 512-bit keys are still used today in some applications. In fact, Valenta, Cohney, Liao, Fried, Codduluri, and Heninger have shown in 2015 that many popular protocols still accept 512-bit keys. In the same paper [62] they present a study of the General Number Field Sieve which shows how it is possible for a non-expert to factor 512-bit RSA keys in

under 4 hours (together with the payment of $100 to access the computing power in the cloud).

A specialized version of the Number Field Sieve, named Special Number Field Sieve, also exists and has been used to factor numbers much bigger than the ones mentioned before. The downside is that the latter algorithm works only for a special kind of numbers, which must have the form $r^h \pm s$, where r, s must be "small" integers. This makes it impossible to use this algorithm for the purpose of breaking RSA.

Leaving aside the usage of Quantum Computers, which will not be discussed in this thesis, the General Number Field Sieve is the most interesting option to recover the secret key of RSA in practical use cases, and that might be enough motivation for many. Furthermore this algorithm is also academically interesting. It uses many results of different fields in mathematics, such as linear algebra, algebraic number theory, finite fields and real and complex analysis. This thesis will focus mainly on the description and explanation of the General Number Field Sieve algorithm, but will also discuss the latest factorizations, the state-of-the-art of the implementations and finally, a plug-in that I developed for the application CrypTool 2 [2]. My plug-in uses an already implemented version of the General Number Field Sieve called *Yet Another Factoring Utility*, YAFU [18], together with a user-friendly GUI under Windows.

I would like to acknowledge my use of S. H. Weintraub book, *Factorization: Unique and Otherwise* [65], but specially the thesis of M. E. Briggs, *An introduction to the General Number Field Sieve* [16]. It helped me understand the GNFS to the extent shown in this thesis, and helped me follow an organized and smooth structure. In comparing this thesis to their texts, one will notice similar arguments for some of the results contained and similar structure in the manner these results are shown.

An introduction to the factorization and the explanation of some basic algorithms, together with some background in the theory of ideals and algebraic number theory will be presented before focusing on the notions needed to understand the General Number Field Sieve. We will proceed with a study of the state-of-the-art in the implementation of the General Number Field Sieve together with some record factorizations in the past years. Some recent attacks on electronic security using the General Number Field Sieve algorithm will follow. Finally, the thesis will end with the explanation of my work in CrypTool 2.

Chapter 2

Factorizing Algorithms Using the Difference of Squares

The General Number Field Sieve algorithm uses the difference of squares method in order to find non-trivial factors of a number, but it was not the first algorithm to use this idea. This chapter will focus on earlier algorithms which also used the difference of squares. However, later in the thesis (Section 7.2), factoring algorithms based on other ideas (such as Pollard $p-1$, Williams $p+1$ or Lenstra ECM) will be covered.

Starting from the first algorithm using the difference of squares, Fermat's method, followed by Dixon's method, where the ideas of factor base and smoothness are introduced, and finishing with the Quadratic Sieve, this chapter will describe the main algorithms prior to the General Number Field Sieve (referred during the thesis as GNFS) using the difference of squares. In this chapter we will describe many of the notions used in the GNFS algorithm itself, as we will discuss factor base, smoothness of a number and a way to find a difference of squares. We will also describe what is the role of sieving, as well as using algebra to create this difference of squares.

An important theorem of integer numbers, in order to properly introduce factorization, is the fundamental theorem of arithmetic:

Theorem 2.0.1. *Every positive integer greater than one, is either a prime itself or the product of prime numbers, and that product, up to rearrangement of the factors, is unique.*

2.1 Fermat's Factorizing Algorithm

We will briefly go through this algorithm because of its historical importance. Pierre de Fermat was one of the first to suggest the idea of difference

of squares for factorizing a number [64]. This idea is based in the fact that if we find $x, y \in \mathbb{Z}/n\mathbb{Z}$ such that $x^2 \equiv y^2 \pmod{n}$, then it is likely we have found non trivial factors of n, since $(x+y)(x-y) \equiv 0 \pmod{n}$. By trivial factors, we mean that $\gcd(n, x \pm y) \in \{1, n\}$. In this case the congruence will equal zero without finding any factors of n. However, Pierre de Fermat did not use modular arithmetic, and was looking for a difference of squares such that $x^2 - y^2 = n$

To start getting our minds to number theory, we will prove that every odd number can be represented as a difference of squares.

Theorem 2.1.1. *Let n be an odd number. Then there exist two numbers $a, b \in \mathbb{Z}$ with $n \nmid a$ and $n \nmid b$ such that $n = a^2 - b^2$.*

Proof. Since n is an odd number it can be represented as $n = 2k + 1$:

$$\begin{aligned} n &= 2k+1 \\ &= (k^2 + 2k + 1 - k^2) \\ &= ((k+1)^2 - k^2) \\ &= (k+1)^2 - k^2 \end{aligned} \tag{2.1}$$

for $k \in \mathbb{Z}$. And we have our difference of squares from any odd number. □

This proof shows how a difference of squares does not guarantee non-trivial factors, however, if $x - y \neq 1$, then $(x+y)(x-y) = n$ gives a non-trivial factor of n. Fermat's algorithm, using this idea, goes as follow: Having an odd composite positive integer n to factor:

1. Let:
   $k = \lfloor \sqrt{n} \rfloor$
   $t = 2k + 1$
   $r = k^2 - n$

2. while (r is not a square)
   {
   $r = r + t$ (The i-th iteration: $r = (k+i)^2 - n + 2(k+i) + 1 = ((k+i)+1)^2 - n$)
   $t = t + 2$
   }

3. $x = (t-1)/2$
   $y = \sqrt{r}$

When r is a square, it means that $y^2 = r = ((k+i)+1)^2 - n = x^2 - n$, and therefore:

$$x^2 - y^2 = (x+y)(x-y) = n$$

The problem of this algorithm is its running time. However, this algorithm works well for numbers who have factors relatively close to $\sqrt{n}$. Fermat's factorization method will try as many as $\frac{n+1}{2} - \sqrt{n}$ steps to factor n and hence has a complexity of $O(\frac{n+1}{2} - \sqrt{n}) = O(n)$. This algorithm can therefore not be used for huge numbers. Nevertheless this idea, developed in a different manner, is what will be used in the most powerful factoring algorithm presently.

2.2 Dixon's Method and the Quadratic Sieve

In this section we will discuss the predecessors of the GNFS. None of them use algebraic numbers nor different rings from $\mathbb{Z}/n\mathbb{Z}$, but in both algorithms, the ideas of using a polynomial for finding relations, having a factor base, and a sieving step (in the case of the Quadratic Sieve) are already implemented.

The Quadratic Sieve (QS) algorithm was developed by Carl Pomerance [48]. It was popular in the 80s and early 90s, but even now, it is the optimal algorithm for factoring numbers between 50 and 100 digits. The idea is very similar to Dixon's method [27] with which we will begin, but let us first define the next two concepts:

Definition 2.2.1. *A non-empty set $\mathcal{F}$ of positive prime integers is called a* ***factor base****. An integer k is called* ***smooth*** *over $\mathcal{F}$ if all prime factors of k are in $\mathcal{F}$.*

Dixon's method relies on Fermat's factorization method, i.e.: that a congruence of squares will yield a factor of n. However, instead of looking for a pair of squares such that $x^2 - y^2 = n$, it looks for a pair of squares such that $x^2 - y^2 \equiv 0 \pmod{n}$.
However it is only with probability 2/3 that this will yield a non trivial factor of n. If p and q both divide $(x-y)$ or $(x+y)$, then $\gcd(x+y, n)$ and $\gcd(x-y, n)$ will yield 1 or n, and this will be when we have no factorization of n. We can see in the following table, as shown in [16], why this results in probability 2/3:

$p\|x+y$?	$p\|x-y$?	$q\|x+y$?	$q\|x-y$?	$\gcd(x+y,n)$	$\gcd(x-y,n)$
Yes	Yes	Yes	Yes	n	n
Yes	Yes	Yes	No	n	p
Yes	Yes	No	Yes	p	n
Yes	No	Yes	Yes	n	q
Yes	No	Yes	No	n	1
Yes	No	No	Yes	p	q
No	Yes	Yes	Yes	q	n
No	Yes	Yes	No	q	p
No	Yes	No	Yes	1	n

It is only in the first, fifth and ninth case that the factorization will yield a trivial factor.

What makes it much more efficient than Fermat's method, is that Dixon relaxes the condition of 'finding a square of an integer' to 'having an integer with small prime factors'. To see how this is relaxed, note that there are 316 squares in the range $[0, 99999]$, while there are 693 numbers with prime factors less or equal than 7, and 5157 with factors less than 30. The first step will be choosing the factor base $\mathcal{F} = \{p_1, \ldots, p_S\}$, which is usually done by considering a bound B and taking all primes below that bound. A polynomial $f(t) = t^2$ is also chosen. We then compute for a set of random integers r_i the values of $f(r_i)$ and we keep track of the ones that satisfy the following:

$$f(r_i) \equiv r_i^2 (\text{mod } n) \text{ is smooth over } \mathcal{F}.$$

The r_i's satisfying the above are called *relations*. What Dixon's method will be doing is finding a set U of integers that satisfy the following:

$$\prod_{r_i \in U} f(r_i) = (p_1^{e_1} \cdots p_S^{e_S})^2, \tag{2.2}$$

where e_i are non-negative exponents that are not all zero.
Note that the only thing we are interested in is that the whole product is a square. The exponents e_i will not make a difference as we will see when explaining the linear algebra step. Now, if we name $x = \prod_{r_i \in U} r_i$ and $y = p_1^{e_1} \cdots p_S^{e_S}$, then we get the following congruence of squares:

$$x^2 = \prod_{r_i \in U} r_i^2 \equiv \prod_{r_i \in U} f(r_i) \equiv y^2 (\text{mod } n).$$

The tricky step is to find the set U. For this we need to find a number of relations of at least the size of the factor base $S = |\mathcal{F}|$, and then, from a basic theorem of algebra, we know we can find such a set. To see this step more clearly, let us consider it in the following way:

For every $f(r_i)$ smooth over the factor base, we will assign it a vector $v_i \in \mathbb{F}_2^S$ where the j-th entry will represent the parity of the power of the j-th prime for $j \in S$. When $L > S$ such r_i's have been found we create a matrix $M \in \mathbb{Z}_2^{S \times L}$. Then, using our knowledge in algebra, we derive that there exists a linear combination of rows of the matrix M that will yield a zero vector. Let us call the group of those rows U. The fact of having that their representative vectors form the zero vector over $\mathbb{F}_2$ means that $\prod_{r_i \in U} f(r_i) = (p_1^{e_1} \cdots p_S^{e_S})^2$, and hence, this set U satisfies the condition (2.2).

What is it then that hinders the algorithm? Dixon's way of finding r_i's such that $f(r_i)$ is smooth over $\mathcal{F}$ is by choosing random integers and doing trial division until finding whether they are smooth or not. This process requires a lot of time since most of these random numbers will not be smooth over $\mathcal{F}$ and therefore we will have a lot of "wasted" operations. To be clear, the problem here is not finding the smooth numbers, but rather how to determine if they are smooth or not.

Pomerance improved this algorithm, naming it the *Quadratic Sieve*. The main differences are the polynomial used, a reduction of the factor base and a sieving procedure to run through the values of r_i. This simple changes produce huge improvements in the running time of the algorithm.
The choice of values r_i is different. In Dixon's method, random values were chosen, while here it is done in a more effective way:
Let $k = \lfloor \sqrt{n} \rfloor$, and now we choose r_i to be $k+1, k+2, \ldots$. The polynomial will be defined as:

$$f(t) = t^2 - n$$

Using this polynomial and these values for r_i will give us the following:

$$f(k+1) = \lfloor \sqrt{n} \rfloor^2 + 2\lfloor \sqrt{n} \rfloor + 1 - n = 2\lfloor \sqrt{n} \rfloor + \tau$$

where $\tau = \lfloor \sqrt{n} \rfloor^2 + 1 - n$, which is ≤ 0 in our first iteration. Note that the generation of values $f(t)$ is the same as what Fermat did in his method to generate r. This will make the values of $f(r_i)$ smaller (compared to the ones of Dixon) and therefore increase the smoothness chance. At each iteration the multiple of $\lfloor \sqrt{n} \rfloor$ will increase, and eventually the values will be big too. However this change in the polynomial does no bring us only that advantage but also sieving which we will explain later.
To explain this improvement we will use the same notions of factor base and smooth number as in Definition 2.2.1, therefore an upper bound, B, will define our factor base $\mathcal{F}$. We may assume that none of these primes divide n (it can be checked with trial division up to B). We now give a step by step explanation of how the new choice of values for r_i together with the new quadratic polynomial makes finding relations much faster.

First of all, Pomerance reduces the size of the factor base considerably by using the Legendre symbol as follows. Let $p_j \in \mathcal{F}$:

$$\text{If } p_j | f(r) \text{ then } n \equiv r^2 \pmod{p_j}$$

Which means that n is a quadratic residue modulo p_j, which can be expressed as $\left(\frac{n}{p_j}\right) = 1$, where $\left(\frac{n}{p_j}\right)$ is the Legendre symbol, defined as:

$$\left(\frac{a}{p_j}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p_j \text{ and } a \not\equiv 0 \pmod{p_j} \\ -1 & \text{if } a \text{ is a quadratic non-residue modulo } p_j \\ 0 & \text{if } a \equiv 0 \pmod{p_j} \end{cases}$$

An important thing to have in mind about the Legendre symbol, is that it is a completely multiplicative function, i.e.;

$$\left(\frac{ab}{p_j}\right) = \left(\frac{a}{p_j}\right)\left(\frac{b}{p_j}\right).$$

Therefore we can already make our factor base a bit smaller, by only choosing the primes with Legendre symbol equal to one. We can make an observation due to the fact that p_j is prime, $\mathbb{Z}/p_j\mathbb{Z}$ is a finite field, and therefore every quadratic residue can only come from two values, i.e. if n is a quadratic residue modulo $p_j \in \mathcal{F}$, it means that $n \equiv t^2 \pmod{p_j}$ or $n \equiv (-t)^2 \pmod{p_j}$, and therefore t and $-t$ are the only roots of n modulo p_j.

The naive approach to continue determining which values of $f(r_i)$ are smooth over the factor base would be to take the primes in $\mathcal{F}$, and do trial division repeatedly until we have reached either 1 or the end of the factor base. Clearly, this is a very slow procedure, since we will find ourselves doing unnecessary divisions to many numbers which will not be divisible by $p_j \in \mathcal{F}, i \in \{1, \ldots, S\}$. Therefore, we want to find a way to do this division with a higher conviction that the number will indeed be divisible by prime $p_j \in \mathcal{F}$.

Here is where the sieving procedure was introduced in the quadratic sieve, and instead of choosing the values of r_i randomly, the following was applied. We select values r_i from a range $[-u, u], u \in \mathbb{Z}/n\mathbb{Z}$, and place in an array the values of $f(r_i)$ within this range. Then, for every prime p_j, we find the first r_i for which $r_i^2 \equiv n \pmod{p_j}$. Once we have found that, then we know that that particular prime p_j only divides the values $r_i + p_j v < u$ for $v \in \mathbb{Z}$, and we avoid doing trial division for all other r_i. This is due to the modular arithmetic, as when doing calculations modulo p_j we can reduce at every step. Therefore if $r_i^2 \equiv n \pmod{p_j}$ then $f(r_i + vp_j) \equiv f(r_i) \equiv r_i^2 - n \equiv 0 \pmod{p_j}$.

For each respective value of the array, we replace $f(r_i)$ by $f(r_i)/p_j$. In order to remove all the powers of p_j dividing each value of $f(r_i)$, we repeat,

for every p_j, the sieve for $p_j^2, p_j^3, \ldots, p_j^e$, such that in every $r_i + p_j^\ell$ position we replace $f(r_i)$ by $f(r_i)/p_j$, for $1 < \ell < e$, where e is the first integer such that p_j^e is too big to fit in the sieving array. Note that for every prime p_j we will run the sieve twice, once for $(-r_i)^2 \equiv n \pmod{p_j}$ and for $(r_i)^2 \equiv n \pmod{p_j}$.
Once we finished checking for a certain value p_j, we pass to our next prime in the factor base and we do the same check as previously. In this way, we are sure of dividing by a prime $p_j \in \mathcal{F}$ which can actually divide $f(r_i)$. At the end of running through the primes in the factor base, we choose the values for which their respective entries in the array are one, i.e.: the number has been totally factored by the numbers in the factor base and therefore is $\mathcal{F}$-smooth.
Division is an expensive operation on these large numbers. A way to avoid these costly operations to numbers which end up not being smooth is by using the following property of logarithms, $\log(a/b) = \log(a) - \log(b)$, and instead of placing $f(r_i)$ in the array, we place $\log(f(r_i))$. Then one goes through the values of $\log(f(r_i))$ such that $r_i^2 \equiv n \pmod{p_j}$, (again, sieving using $r_i + p_j v < u$) and subtracts $\log(p_j)$. In order to remove all the powers of each p_j, we perform the same sieve as before. At the end of this process we divide the values of $f(r_i)$ by the primes dividing it, only for the values which its respective entry in the array is $0 = \log(1)$ in order to check that we indeed reach one. This last check is necessary due to the errors in the rounding of the logarithms to make sure that we are accepting only smooth values. Finally, the step of the linear algebra consists in the same steps used by Dixon.

Dixon's modification of finding a difference of squares was a huge advance in factoring. Before him, the idea of using modular arithmetic was already used for factoring algorithms. However Dixon was the first to develop the idea of using smooth numbers to create these squares, and as we have seen, smooth numbers are much more likely to be found than perfect squares. This led Dixon to finding an algorithm where the expected number of operations required is $O(\exp(\beta(\log n \log\log n)^{1/2}))$ for some constant $\beta > 0$.
Pomerance, with the small modification of the polynomial, the usage of the Legendre symbol and the sieving idea improved massively the step of finding relations, and developed an algorithm with conjectured complexity of $O(\exp((1 + o(1))(\log n \log\log n)^{1/2}))$.

Now the question is how can we do further improvements from here. One idea would be to optimize the search of smooth values by augmenting the size of the factor base. This way it would be easier to find smooth values, but then the problem is that you need more such values, since the number of relations needed is directly related to the size of the factor base.

The GNFS goes in a totally innovating direction, and while it stays with the same concepts as the quadratic sieve, it uses totally different notions of mathematics to achieve this, the most important being the use of rings other than $\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$.

Before starting with the GNFS we will have an interlude explaining some important concepts of algebraic number theory and theory of ideals.

Chapter 3

Interlude

3.1 Introduction

It is quite complicated to see how the use of number fields, or even complex analysis can help in solving a problem such as integer factorization. As a matter of fact, the problem of factorization has not been the only "integer" problem to use the help of number fields. This section will be introduced with two problems that were also solved using number fields. The intention is not to present a proof of these problems, but rather name them and present them to the reader as a motivation, therefore proofs will be omitted. The first of the two problems is the following:

What are the integer solutions for the following equation?

$$x^3 = y^2 + 2$$

Euler proved that this problem only has a solution with $x = 3$ and $y = 5$. As one may imagine at this point, he did it with the use of number fields. He expressed the equation as:

$$y^2 + 2 = (y + \sqrt{-2})(y - \sqrt{-2}) = x^3$$

He then did the proof by treating the factors as if they where relatively prime integers. He did not offer a proof that this was the unique prime factorization, but as we will see later in the paper, the ring $\mathbb{Z}[\sqrt{-2}]$ is indeed a unique factorization domain [51].

Another interesting proof using notions of algebraic number theory is the proof [42] of Fermat's last theorem:

$$x^l + y^l = z^l \text{ has nonzero solutions } x, y, z \text{ only if } l \leq 3$$

So we notice ourselves that algebraic number theory is a tool that can be used in "simple" problems with "complicated" solutions, and with this we mean a

statement that can be easily understood, even by non mathematicians, but a solution which requires tools not accessible to all. Factorization could be seen as one of these problems, since the idea of the problem is quite simple to understand, but when it comes to big numbers and a limited time frame to solve the problem, the solution increases in complication. In this paper we will see how algebraic numbers help in our task to factor big numbers. Despite the fact of discovering new fields in the quest of factorization, the notions of factor base and smoothness, slightly modified as we will see later, remain in the algorithm. We will first introduce the algebraic background, followed by factorization into irreducibles and finish with ideals in number fields. These two last sections are of high importance due to the following property of the field extensions. Take, for instance, $\mathbb{Z}[\sqrt{2}]$, in this extended ring, $6 = 3 \times 2$ is no longer the unique factorization, since:

$$6 = (2 + \sqrt{2})(2 - \sqrt{2}).$$

Due to this, new notions of factor base and unique factorization (different from the previously introduced *unique factorization* of Theorem 2.0.1) will have to be introduced.

3.2 Algebraic Number Theory

This section will go through some basic definitions of algebraic number theory, followed by properties of the number fields. It is a very wide subject, and we will limit the notions explained in this thesis to what will be used to explain the GNFS. For further study or other applications, we refer the interested reader to two books of algebraic number theory, [42] or [35]. In the course of this thesis more notions will be introduced, and this section is constrained as an introduction of the basics.

Definition 3.2.1. *A number $\alpha \in \mathbb{C}$ is an **algebraic number** if there exists a polynomial $f \in \mathbb{Q}[t]$ such that $f(\alpha) = 0$. A number $\beta \in \mathbb{C}$ is an **algebraic integer** if there exists a monic[1] polynomial $f \in \mathbb{Z}[t]$ such that $f(\beta) = 0$. The subfield of $\mathbb{C}$ consisting of all algebraic numbers will be denoted by $\overline{\mathbb{Q}}$, and the set of all algebraic integers of $\overline{\mathbb{Q}}$ by $\mathfrak{O}$.*

Definition 3.2.2. *An **algebraic number field** is of the form*

$$K = \mathbb{Q}(\alpha_1, \ldots, \alpha_d) \subseteq \mathbb{C} \text{ with } d \in \mathbb{N}^+ \text{ where } \alpha_j \in \overline{\mathbb{Q}} \text{ for } j = 1, \ldots, d$$

*Let $K = \mathbb{Q}(\alpha)$ be an algebraic number field, then $\mathfrak{O}_K$ is the **ring of algebraic integers** of K.*

[1] Polynomial with leading coefficient equal to one.

The following theorem will help us know more about $\mathbb{Q}(\alpha)$:

Theorem 3.2.3. *[33, Theorem 1.6, Chapter 5]*
Given a monic, irreducible polynomial $f(t) \in \mathbb{Q}[t]$ of degree d, a root $\alpha \in \mathbb{C}$ of f(t), and the associated ring $\mathbb{Q}(\alpha)$, the following hold:

1. $\mathbb{Q}(\alpha) \cong \mathbb{Q}[t]/(f(t))$.
2. *$f(t)$ divides any polynomial g(t) for which $g(\alpha) = 0$.*
3. *The set $\{1, \alpha, \ldots, \alpha^{d-1}\}$ forms a basis for $\mathbb{Q}(\alpha)$ as a vector space over $\mathbb{Q}$.*

Note: We will not go through the proof, however, it is important for later in the thesis to see that there exists a surjective ring homomorphism $\psi : \mathbb{Q}[t] \to \mathbb{Q}(\alpha)$ defined by $\psi(\mathbb{Q}) = \mathbb{Q}$ and $\psi(t) = \alpha$. It is easy to see how this map it is surjective by noting that every element in $\mathbb{Q}(\alpha)$ is represented by a linear combination of $\{1, \alpha, \ldots, \alpha^{d-1}\}$.
The following proposition will be used throughout the thesis for proofs or demonstrations.

Proposition 3.2.4. *Given a monic, irreducible polynomial f(t) of degree d with integer coefficients and a root $\alpha \in \mathbb{C}$ of f(t), the set of all $\mathbb{Z}$-linear combinations of the elements $\{1, \alpha, \ldots, \alpha^{d-1}\}$, denoted $\mathbb{Z}[\alpha]$, forms a subring of the ring of algebraic integers $\mathfrak{O}_K$ of $K = \mathbb{Q}(\alpha)$.*

To see how equality does not always hold in $\mathbb{Z}[\alpha] \subseteq \mathfrak{O}_K$, let us consider the following example:
Let $\mathbb{Q}(\sqrt{5})$ be the field formed by the irreducible polynomial in $\mathbb{Z}[t]$, $t^2 - 5$. The field is generated by the basis $S = \{1, \sqrt{5}\}$. Let $\alpha = (1 + \sqrt{5})/2$, which is in $\mathbb{Q}(\alpha)$ since it is a $\mathbb{Q}$-combination of the elements in S. It is also an algebraic integer since it is a root of the irreducible polynomial $f(t) = t^2 - t - 1$, but clearly $\alpha \notin \mathbb{Z}[\sqrt{5}]$, hence in this case we have that:

$$\mathbb{Z}[\sqrt{5}] \subsetneq \mathfrak{O}_{\mathbb{Q}(\alpha)} \subsetneq \mathbb{Q}(\sqrt{5})$$

3.3 Theory of Ideals

This thesis is aimed at students of mathematics having already knowledge in the theory of ideals. Nevertheless, I want to use this section to remind the reader of some notions such as fractional ideals, unique factorization of ideals and the study of principal ideal domains (PIDs) in the case of Dedekind domains.
We will begin by introducing what ideals are, as their role in the understanding of making the ring of algebraic integers $\mathfrak{O}_K$ of a number field K as a unique factorization domain is crucial. However, we present a general notions of ideals for integral domains:

Definition 3.3.1. *An **integral domain** is a commutative ring with an identity ($1 \neq 0$) with no zero-divisors. This is, if $ab = 0$, then either a or b equal zero.*

Note that every field is an integral domain, and therefore the notions described for integral domains, will also apply for number fields.

Definition 3.3.2. *An **ideal** $\mathfrak{p}$ of an integral domain R is a nonempty subset $\mathfrak{p}$ of R such that:*

- *$\mathfrak{p}$ is a subgroup of R (under the operation of addition),*
- *if $a \in \mathfrak{p}$ and $x \in R$ then $xa \in \mathfrak{p}$.*

Definition 3.3.3. *Let $r \in R$. The **principal ideal** with generator r is:*

$$\mathfrak{p}_r = \{r' | r' = r\beta \text{ for some } \beta \text{ in } R\}.$$

Definition 3.3.4. *Let $\mathcal{D} = \{r_i\}$ be a nonempty set of elements of R. The **ideal generated by** $\mathcal{D}$ is:*

$$\mathfrak{p}_{\mathcal{D}} = \left\{ \sum r_i \beta_i | \beta_i \in R \text{ with only finitely many } \beta_i \neq 0 \right\}$$

Definition 3.3.5. *An integral domain R in which all ideals are principal is called a **principal ideal domain** or PID.*

Definition 3.3.6. *If R is an integral domain in which every nonzero, nonunit can be represented as a finite product of irreducible elements of R, then R is called a factorization domain. If this factorization is unique up to units and order of the factors, it is called a **Unique Factorization Domain** or UFD.*

Definition 3.3.7. *An ideal $\mathfrak{p}$ is **prime** if it satisfies the following properties:*

- *If a and b are two elements of R such that their product ab is an element of $\mathfrak{p}$, then a is in $\mathfrak{p}$ or b is in $\mathfrak{p}$.*
- *$\mathfrak{p}$ is not equal to the whole ring R.*

Definition 3.3.8. *A **maximal ideal** is an ideal so that there are no other ideals contained between $\mathfrak{p}$ and R.*
This is, if $\mathfrak{q}$ is an ideal and $\mathfrak{q} \supseteq \mathfrak{p}$, then $\mathfrak{q} = \mathfrak{p}$ or $\mathfrak{q} = R$.

Now we can introduce the notion of Dedekind Domains, which will be very important throughout our explanation of algebraic number theory in the following sections and will also help us to determine how to study unique factorization with ideals. Beforehand, let us introduce the notion of *quotient field* and *integrally closed*, and the *ascending chain condition*

Definition 3.3.9. *Let R be an integral domain, and E be a field. Then E is the* ***quotient field*** *of R if:*

$$E = \left\{ \frac{a}{b} | a, b \in R, \text{ and } b \neq 0 \right\}$$

with $a/b = c/d$ in E if $ad = bc$ in R.

Note: We are not assuming that division exists, '/' is just a symbol meaning $b[a/b] = a$ and when defining E we can give it an interpretation.

Definition 3.3.10. *An integral domain R satisfies the* ***ascending chain condition*** *if every sequence of ideals $\mathfrak{p}_1 \subset \mathfrak{p}_2 \subset \mathfrak{p}_3 \ldots$ of R is finite.*

To proceed with the definition of *integrally closed*, we generalize the notion of algebraic integer. For this, let Z be any subring of $\mathbb{Q}$. Then an element θ of an algebraic number field K is Z-integral if its minimum polynomial has all its coefficients in Z. With this notion, we proceed with the definition of integrally closed:

Definition 3.3.11. *Let R be a subring of an algebraic number field K, and let $Z = R \cap \mathbb{Q}$. The integral closure of R in K is $S = \{\theta \in K | \theta$ is Z-integral$\}$. R is* ***integrally closed*** *in K if $R = S$.*

Now we have the necessary tools to define a Dedekind domain.

Definition 3.3.12. *A* ***Dedekind domain*** *is an integral domain R satisfying the following properties:*

- *R satisfies the ascending chain condition.*
- *Every nonzero prime ideal of R is maximal.*
- *R is integrally closed in its quotient field E.*

We continue with the definition of fractional ideals.

Definition 3.3.13. *Let R be an integral domain with quotient field E. A nonempty subset $\mathfrak{p}$ of E is called a* ***fractional*** *ideal of E if it satisfies the following properties:*

- *For any $\alpha, \beta \in \mathfrak{p}$, $\alpha + \beta \in \mathfrak{p}$.*
- *For any $\alpha \in \mathfrak{p}$ and $r \in R$, $r\alpha \in \mathfrak{p}$.*
- *There exists a nonzero $\gamma \in R$ such that $\gamma\mathfrak{p} \subseteq R$.*

If $\mathfrak{p}_1$ and $\mathfrak{p}_2$ are fractional ideals, then their product $\mathfrak{p}_1\mathfrak{p}_2$ is the smallest subset of K which is closed under addition and which contains all products $i_1 i_2$ where $i_1 \in \mathfrak{p}_1$ and $i_2 \in \mathfrak{p}_2$.

Definition 3.3.14. *Let $\mathfrak{p}$ be a fractional ideal. The **inverse** of $\mathfrak{p}$ is $\mathfrak{p}^{-1} = \{i \in R : i\mathfrak{p} \subseteq E\}$.*

Lemma 3.3.15. *Let $\phi : R \to S$ be a ring homomorphism.* $\ker(\phi)$ *is an ideal of R.*

Proof. Suppose that $a \in \ker(\phi)$ and $r \in R$, then

$$\phi(ra) = \phi(r)\phi(a) = \phi(r)0 = 0.$$

Therefore $ra \in \ker(\phi)$. Similarly for ar. Now we want to prove that $a + b \in \ker(\phi)$ if a and b is in the kernel.

$$\phi(a + b) = \phi(a) + \phi(b) = 0 + 0 = 0$$

Hence $a + b$ is in the kernel and therefore $\ker(\phi)$ is an ideal of R. □

Lemma 3.3.16. *If R is a commutative ring with identity 1_R, S is a commutative ring with identity 1_S, and $\phi : R \to S$ is a ring homomorphism, then $\phi(1_R) = 1_S$.*

Proof. Let $y \in S$. Since ϕ is a surjective ring homomorphism there exists $x \in R$ such that $\phi(x) = y$. Then we have $y \cdot \phi(1_R) = \phi(1_R) \cdot \phi(x) = \phi(1_R \cdot x) = \phi(x) = y$, and hence $\phi(1_R) = 1_S$. □

Now we present two important theorems, for which the proof is accessible in [65, p.153]. These results determine the conditions for the unique prime factorization of ideals to happen.

Theorem 3.3.17. *Every nonzero ideal in a Dedekind domain D is uniquely representable as a product of prime ideals. In other words, any ideal I has a unique expression (up to order of the factors) of the form:*

$$I = \mathfrak{p}_1^{a_1}\mathfrak{p}_2^{a_2} \cdots \mathfrak{p}_\ell^{a_\ell}$$

where the $\mathfrak{p}_i$ are the distinct prime ideals containing I, and $a_j \in \mathbb{N}^+$ for $j = 1, \ldots, \ell$ and $a_j \geq 1$.

Theorem 3.3.18. *If D is a Dedekind domain, then D is a unique factorization domain (UFD) if and only if D is a PID*

Finally, a very important result accessible in [65, p.155], states the following:

Theorem 3.3.19. *Let K be an algebraic number field and let $R = \mathfrak{O}_K$, then R is a Dedekind domain.*

Finally we end up with an important notion which will help us build the GNFS, the norm of an ideal. Later in the thesis we will see how this is related to the notion of a norm of an element $\theta \in \mathbb{Q}(\alpha)$.

Definition 3.3.20. *Given a ring R and an ideal $\mathfrak{p}$ of R, the* ***norm*** *of $\mathfrak{p}$ is defined to be $[R : \mathfrak{p}]$, the number of cosets of $\mathfrak{p}$ in R.*

We will develop the presented ideas further in the thesis, but for now we have a basis to start with the GNFS and see how we can use these concepts to create a difference of squares.

Chapter 4

The General Number Field Sieve

The General Number Field Sieve (GNFS) uses the same idea as Fermat's method, Dixon's method or the Quadratic Sieve, namely the difference of squares to factor a number. The quadratic sieve algorithm arrived to a point which was very hard to optimize factorization. Some testing tried optimizing the algorithm by means of augmenting the factor base (to make the search of smooth numbers easier) or using multiple polynomials for the sieve [56]. The usage of a quadratic polynomial was believed to be the best choice to produce the difference of squares desired in order to factor a number, but as the GNFS shows by using polynomials of higher degree, this was not the optimal choice for factoring numbers with big factors. However the most important improvement in the GNFS algorithm is using rings other than $\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$ that have the notion of unique prime factorization and smoothness imposed on them. In this section we will describe how the GNFS exploits this idea. We will begin by explaining how the GNFS creates the difference of squares, followed by the updated notions of factor base and smoothness. The norm of ideals plays a very important role in the GNFS, so we will dedicate a section to explain this notion and relate it with our sieving procedure. Finally, the task of determining whether a number is a perfect square in $\mathbb{Z}[\alpha]$ will need the help of quadratic characters, with which we will finish the section.

Note: Throughout this section, we will refer to the ring of algebraic integers of $\mathbb{Q}(\alpha)$ *as* $\mathfrak{O}_\alpha$.

4.1 Creating a Difference of Squares

The idea in the quadratic sieve was to use a quadratic polynomial, $f(t) = t^2 - n$ in order to produce squares modulo n. Another way to think about this

polynomial is as a ring homomorphism which sends an element of the ring $\mathbb{Z}$ to a square in the ring $\mathbb{Z}/n\mathbb{Z}$, $f : \mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$. Note that it is not the unique polynomial which can be seen as such a map, for instance $f(t) = t^2 + tn - n$ has the same effect. Up to now we have mentioned that the GNFS makes use of a different ring other that $\mathbb{Z}/n\mathbb{Z}$ in order to find smooth elements in a faster manner. After the interlude we might have an idea which ring we will be using, but before that, let us call the new ring R and assume there exists a natural homomorphism $\phi : R \to \mathbb{Z}/n\mathbb{Z}$. Now, let $\beta \in R$. If we manage to find $y, x \in \mathbb{Z}/n\mathbb{Z}$ such that $\phi(\beta^2) = y^2 \pmod{n}$ and $x = \phi(\beta)$ then we can produce a difference of squares in the following manner:

$$x^2 \equiv \phi(\beta)^2 \equiv \phi(\beta^2) \equiv y^2 \pmod{n} \tag{4.1}$$

In the previous section, we already introduced the extensions $\mathbb{Z}[\alpha]$ and $\mathbb{Q}(\alpha)$, where $\alpha \notin \mathbb{Z}, \mathbb{Q}$. The reason for having introduced these notions of such rings is due to the following property presented in [40, p.53]:

Theorem 4.1.1. *Let $f \in \mathbb{Z}[t]$ be a monic, irreducible polynomial of degree $d > 1$. Let $\alpha \in \mathbb{C}$ be a root of $f(t)$. Let $n \in \mathbb{N}$, $m \in \mathbb{Z}$ be such that $f(m) \equiv 0 \pmod{n}$. Then there exists a natural ring homomorphism $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$ induced by $\phi(\alpha) \equiv m \pmod{n}$ with $\phi(1) \equiv 1 \pmod{n}$.*

Proof. To construct this homomorphism we choose f, m and d as above, and we combine the ordinary reduction map on $\mathbb{Z}$ with $\mathbb{Z}[\alpha]$ to obtain the following ring homomorphism:

$$\begin{aligned} \phi : \mathbb{Z}[\alpha] &\to \mathbb{Z}/n\mathbb{Z} \\ \left(\sum_{i=0}^{d-1} a_i \alpha^i\right) &\mapsto \left(\sum_{i=0}^{d-1} a_i m^i \mod n\right) \end{aligned}$$

In this way, we have that $\phi(1) = \phi(1\alpha^0) \equiv 1 \pmod{n}$.
Now we need to check that ϕ is well defined. Let $g(t), h(t) \in \mathbb{Z}[t]$. Note that f is a monic, irreducible polynomial, which means that f is the minimal polynomial of α. In order to see if ϕ is well defined, we need to show that if $g(\alpha)$ and $h(\alpha)$ are equal, then $\phi(h(\alpha)) = \phi(g(\alpha))$. Start by noting that if $g(\alpha) = h(\alpha)$ then we have that α is a root of $g - h$, and as we have mentioned before, f is the minimal polynomial of α, therefore we have that $f | g - h$. So we have that $g - h = fl$, for $l(\alpha) \in \mathbb{Z}[\alpha]$. With this deduced, we have that:

$$g(m) - h(m) = f(m)l(m)$$

But remember that we have created our polynomial f in a way that $f(m) \equiv 0 \mod n$, therefore we have that $g(m) - h(m) = f(m)l(m) \equiv 0 \mod n$, and therefore $g(m) - h(m)$ is divisible by n so that $g(m) \equiv h(m) \mod n$ and we have that ϕ is well defined. □

This is the homomorphism that the GNFS will make use of. To see how it is possible to create a difference of squares with this, we apply the homomorphism to the set U of pairs (a, b) chosen such that the following holds:

$$\prod_{(a,b)\in U}(a+b\alpha) = \beta^2 \text{ and } \prod_{(a,b)\in U}(a+bm) = y^2 \tag{4.2}$$

To see how we can create a difference of squares with these notions, let $x = \phi(\beta)$, where $\beta \in \mathbb{Z}[\alpha]$ and $y \in \mathbb{Z}$ as in (4.2). Then:

$$\begin{aligned} x^2 \equiv \phi(\beta)^2 \equiv \phi(\beta^2) \equiv \phi\left(\prod_{(a,b)\in U}(a+b\alpha)\right) \\ \equiv \prod_{(a,b)\in U}\phi(a+b\alpha) \equiv \prod_{(a,b)\in U}(a+bm) \equiv y^2 \pmod{n} \end{aligned} \tag{4.3}$$

It is important to note that the condition that the first square produced by U in (4.2) is a perfect square in $\mathbb{Z}[\alpha]$ is imposed because the ring homomorphism is only defined in $\mathbb{Z}[\alpha]$. But in practice, in order to make the condition less restrictive, we relax the condition for the product to be a perfect square in $\mathbb{Q}(\alpha)$. Then the just defined way to produce a square cannot be used.

Let us assume that the pairs (a, b), in the first product of equation (4.2) create a square $\theta^2 \in \mathbb{Q}(\alpha)$. To solve this obstruction in an easy manner we consider the following lemma shown in [40, p.60]:

Lemma 4.1.2.

If $\prod_{(a,b)\in U}(a+b\alpha) = \theta^2$ *for* $\theta \in \mathbb{Q}(\alpha)$ *then* $\theta \in \mathfrak{O}_\alpha$ *and* $f'(\alpha)\cdot\theta \in \mathbb{Z}[\alpha]$

where $f'(t)$ *is the derivative of* $f(t)$.

Therefore, if we have a set U similar to the one in (4.2) such that:

$$\prod_{(a,b)\in U}(a+b\alpha) = \theta^2 \text{ and } \prod_{(a,b)\in U}(a+bm) = z^2$$

with $\theta \in \mathfrak{O}_\alpha$ and $z \in \mathbb{Z}$, we let $\beta = f'(\alpha)\cdot\theta \in \mathbb{Z}[\alpha], y = f'(m)\cdot z$ and

$x = \phi(\beta) \in \mathbb{Z}/n\mathbb{Z}$, then:

$$\begin{aligned} x^2 &\equiv \phi(\beta)^2 \equiv \phi(\beta^2) \\ &\equiv \phi\left(f'(\alpha))^2 \prod_{(a,b)\in U} (a+b\alpha)\right) \\ &\equiv \phi(f'(\alpha))^2 \prod_{(a,b)\in U} \phi(a+b\alpha) \\ &\equiv f'(m)^2 \prod_{(a,b)\in U} (a+bm) \\ &\equiv y^2 \pmod{n} \end{aligned} \tag{4.4}$$

and a difference of squares is reached.

We will expand the polynomial choice idea in section 5.1, but for the moment, we will see how such a polynomial is easy to create in a simple manner. We begin by choosing the integer m, which can be defined by $m = \lfloor \sqrt[d]{n} \rfloor$, then we write n in base m, i.e.:

$$n = c_d m^d + c_{d-1} m^{d-1} + \cdots + c_0$$

with $c_d = 1$ and we let

$$f(t) = t^d + c_{d-1} t^{d-1} + \cdots + c_0.$$

In order to satisfy assumptions made to create the homomorphism, the polynomial f must be irreducible.
Many studies have been focusing on how to determine whether a polynomial is indeed irreducible. One well known such method is the Eisenstein criteria:

Theorem 4.1.3. *Let $f(t) = \sum_{i=0}^{d} a_i t^i$. If there exists a prime number p such that $p \nmid a_n$, $p | a_i$ for all $i = 0, 1, \ldots, n-1$ and $p^2 \nmid a_0$, then $f(t)$ is irreducible over $\mathbb{Q}$.*

Another widely used result comes from probabilistic Galois theory, which states that almost all polynomials with integer coefficients are irreducible [59]. This will be assumed to be the case. In case it is reducible, the algorithm would end here with a factorization of n. To see how this is the case imagine that the polynomial created is indeed reducible. Then, by using algorithms to factor polynomials (such as Cantor–Zassenhaus algorithm [20] or Berlekamp's algorithm [14]) we reduce $f(t) = g(t) \cdot h(t)$, but note that, by construction, $f(m) = n$ and therefore $g(m) \cdot h(m) = n$, and if the factorization of f is not trivial, neither would be the factorization of n and we are therefore done. One may ask now, if it is so simple to factor a polynomial, why not just factor $f(m) = n$ and get in this manner the factors

of n? The obstruction here is to create the polynomial $f(m) = n$ such that it is reducible. So throughout the thesis we assume that $f(t)$ is irreducible over $\mathbb{Z}[t]$, and therefore a homomorphism can be created.
Now the problem is to find the set U such that (4.2) is satisfied. In order to accomplish this, we will use the same method used in the quadratic sieve. We will sieve over integers until we find a big enough set of numbers U such that $a+\alpha b$ is smooth over a factor base defined in $\mathbb{Z}[\alpha]$ and $a+mb$ is smooth over a factor base defined in $\mathbb{Z}/n\mathbb{Z}$. We will only consider pairs (a, b) that do not share a factor. In the explanation of Dixon's method and the quadratic sieve we have already introduced the notions of smoothness and factor base, but since we are working in an extension of the ring of integers, we need to redefine these two notions, starting by redefining the notion of unique factorization.

4.2 Smoothness and the Algebraic Factor Base

At this point, we might ask ourselves why is there so much interest in using such a complicated algorithm instead of using for instance the quadratic sieve. The answer is easy: it is faster for larger numbers. But why is it faster? The step that requires time in both Dixon's and Pomerance's method is sieving in order to find enough relations to produce the difference of squares. This step is directly related to the bit length of n. In the quadratic sieve we need to consider whether x^2-n is smooth, but if $x \approx \sqrt{n}$, then $x^2 - n$ is of magnitude about $\sqrt{n}$. The GNFS works in a different manner. On the one hand we need to determine whether $a + bm \mod n$ is smooth. The values of $a, b \in \mathbb{Z}$ are chosen within a range $-u < a, b < u$ with $u \in \mathbb{Z}$, and the number m is chosen to be of order $n^{1/d}$. This results in a number of order $n^{1/d}$ [35], which with a sufficiently large d (say $d \geq 5$) makes it more likely to be smooth than in the quadratic sieve. A good choice is $d = 5$, as mentioned in [15]. On the other hand one needs to determine whether $(a+b\alpha)$ is smooth over a factor base consisting of prime ideals, and this section will explain specifically which ideals and how to determine smoothness. As we will see, the notion of smoothness is directly related to the homogenized polynomial, and therefore, the size of the values to check for smoothness is related to the exponent of the polynomial. This is the reason why we must limit the size of d and not make it too big.

The new notion of smoothness becomes pretty simple. What we want to do is determine whether $a + b\alpha$ is smooth over an algebraic factor base. To do this we could use a map that sends elements from $\mathbb{Z}[\alpha]$ to elements of $\mathbb{Z}$. This algebraic factor base and map must have the property that the smoothness of the mapped element will determine the smoothness of the element in the algebraic factor base. This is exactly what we will do with

the help of the norm defined in $\mathbb{Q}(\alpha)$. Before defining the norm, we will present the following theorem following the steps presented in [16]:

Theorem 4.2.1. *There are exactly d ring injective homomorphisms (embeddings) from the field $\mathbb{Q}(\alpha)$ into the field $\mathbb{C}$. These embeddings are given by $\sigma_i(\tau) = \tau$ for all $\tau \in \mathbb{Q}$ and $\sigma_i(\alpha) = \alpha_i$, for $1 \leq i \leq d$, assuming f(x) splits over $\mathbb{C}$ as:*

$$f(t) = (t - \alpha_1) \cdots (t - \alpha_d).$$

Proof. The canonical mapping $\sigma_i : \mathbb{Q}(\alpha) \to \mathbb{Q}(\alpha_i)$ which sends α to α_i for $1 \leq i \leq d$ is an isomorphism of fields, which means that each σ_i is a distinct isomorphic copy of $\mathbb{Q}(\alpha)$ in $\mathbb{C}$, which means that there are at least d embeddings from $\mathbb{Q}(\alpha)$ into $\mathbb{C}$. To show that these are all possible embeddings, suppose that $\sigma : \mathbb{Q}(\alpha) \to \mathbb{C}$ is a injective ring homomorphism, then $\sigma(\mathbb{Q}) = \mathbb{Q}$. If $\sigma(\alpha) = \theta \in \mathbb{C}$ and $f(t) = t^d + a_{d-1}t^{d-1} + \cdots + a_1 t + a_0$, then

$$\begin{aligned} f(\theta) &= \theta^d + a_{d-1}\theta^{d-1} + \cdots + a_1\theta + a_0 \\ &= \phi(\alpha)^d + a_{d-1}\phi(\alpha)^{d-1} + \cdots + a_1\phi(\alpha) + a_0 \\ &= \phi(\alpha^d + \cdots + a_1\alpha + a_0) \\ &= \phi(0) \\ &= 0 \end{aligned} \tag{4.5}$$

and therefore $\theta = \alpha_i$ and $\sigma = \sigma_i$ for some $1 \leq i \leq d$. Hence, the σ_i are the only embeddings. □

Now we are ready to define the norm:

Definition 4.2.2. *The **norm** of the element $\theta \in \mathbb{Q}(\alpha)$, denoted by $N(\theta)$, is defined as:*

$$N(\theta) = \sigma_1(\theta)\sigma_2(\theta) \cdots \sigma_d(\theta)$$

where σ_i for $i \in \{1, \ldots, d\}$ are the embeddings described in the previous theorem.
We now redefine what will be considered as algebraic smoothness. We do not have enough information in order to totally understand this explanation. However, this section will be used in order to get the necessary notions in order to comprehend the following definition:

Definition 4.2.3. *Consider $\mathbb{Q}(\alpha)$, a number field, then an algebraic number $a + b\alpha \in \mathbb{Z}[\alpha]$ is **B-smooth** if $|N(a + b\alpha)|$ is B-smooth.*

Note that we have defined our new notions of smoothness for elements of $\mathbb{Z}[\alpha]$ while we have the norm defined for elements of $\mathbb{Q}(\alpha)$. For this we need to know more about this norm, which will be studied further. As we saw, our interest of smoothness is for elements of $\mathbb{Z}[\alpha]$, therefore, first of all, the norm must be defined for elements in $\mathbb{Z}[\alpha]$, and it must map

elements to $\mathbb{Z}$ and not to $\mathbb{C}$. Further in this chapter we will see how we can indeed relate the factorization of the norm of an element $\beta \in \mathbb{Z}[\alpha]$ to the factorization of the element itself into ideals.

Our next step is to show that the norm is a map to elements of $\mathbb{Z}$ and we have to study how the norm reacts when we apply it to elements of $\mathbb{Z}[\alpha]$ instead of $\mathbb{Q}(\alpha)$. For this we will make use of the notions presented in the interlude (Chapter 3), together with other concepts presented throughout this section. We will begin by an important property, but beforehand we need to introduce two results and a definition from [42, p.65-p.67].

Definition 4.2.4. *Let $\mathbb{Q}(\alpha)$ be an algebraic number field of degree d over $\mathbb{Q}$, and let $\sigma_1, \ldots, \sigma_d$ be the embeddings of $\mathbb{Q}(\alpha)$ in $\mathbb{C}$. For each element $\beta \in \mathbb{Q}(\alpha)$,*

$$T_{\mathbb{Q}(\alpha)}(\beta) = \sum_{j=1}^{d} \sigma_j(\beta)$$

*is called the **trace** of β.*

Lemma 4.2.5. *Let $\mathbb{Q}(\alpha)$ be an algebraic number field, and let $\beta \in \mathbb{Q}(\alpha)$ with $[\mathbb{Q}(\beta) : \mathbb{Q}] = \ell$, then we can express $m_{\beta,\mathbb{Q}}(t)$ (the minimal polynomial of β over $\mathbb{Q}$) as :*

$$m_{\beta,\mathbb{Q}}(t) = t^{\ell} - T_{\mathbb{Q}(\alpha)}(\beta)t^{\ell-1} + \cdots \pm N_{\mathbb{Q}(\alpha)}(\beta).$$

Lemma 4.2.6. *Let $\theta \in \mathbb{Q}(\alpha)$, and let $m_{\theta,\mathbb{Q}}(t)$ be the minimal polynomial of θ over $\mathbb{Q}$. Then $\theta \in \mathfrak{O}_\alpha$ if and only if $m_{\theta,\mathbb{Q}}(t) \in \mathbb{Z}[t]$.*

Now we have enough information to start with the proof of the following proposition:

Proposition 4.2.7. *The norm defined previously, Definition 4.2.2, is a multiplicative function that maps elements of $\mathbb{Q}(\alpha)$ to $\mathbb{Q} \subset \mathbb{C}$. Furthermore, algebraic integers in $\mathbb{Q}(\alpha)$ are mapped to elements of $\mathbb{Z}$.*

Proof. For the first part of this proof we use the form of the minimal polynomial of θ over $\mathbb{Q}$ as in Lemma 4.2.5:

$$m_{\theta,\mathbb{Q}}(t) = t^{\ell} - T_{\mathbb{Q}(\alpha)}(\theta)t^{\ell-1} + \cdots \pm N_{\mathbb{Q}(\alpha)}(\theta)$$

where $T_{\mathbb{Q}(\alpha)}$ is the trace defined in Definition 4.2.4.
It is clear from the definition of the minimal polynomial that $m_{\theta,\mathbb{Q}}(x) \in \mathbb{Q}[t]$, and therefore it follows that the norm, as well as the trace, of θ from $\mathbb{Q}(\alpha)$ maps elements from $\mathbb{Q}(\alpha)$ to $\mathbb{Q}$.
To prove the second statement we use Lemma 4.2.6 that $\theta \in \mathfrak{O}_\alpha$ if and only if $m_{\theta,\mathbb{Q}}(t) \in \mathbb{Z}[t]$, which means its coefficients are integers. With this result

and with the definition of the minimal polynomial of θ over $\mathbb{Q}$, we have that if θ is an algebraic integer, then the norm maps elements from $\mathbb{Q}(\alpha)$ to $\mathbb{Z}$. □

Which now leads us to the following important corollary, which is trivial by the fact that $\mathbb{Z}[\alpha] \subseteq \mathfrak{O}_\alpha$:

Corollary 4.2.8. *The norm function is a multiplicative function that sends elements of $\mathbb{Z}[\alpha]$ to elements of $\mathbb{Z}$.*

Now that we see how the norm sends elements from $\mathbb{Z}[\alpha]$ to elements of $\mathbb{Z}$, we use the following definition which will help us present an easy way of computing the norm.

Definition 4.2.9. *Let $f \in \mathbb{Z}[t]$ be a monic, irreducible polynomial of degree $d > 1$. The **homogenized** polynomial, $F(X, Y) \in \mathbb{Z}[X, Y]$ is defined as:*

$$F(X, Y) = (Y)^d f(X/Y)$$

Let $a + b\alpha$ be the number we want to calculate the norm of. Then using the embeddings of Theorem 4.2.1 and Definition 4.2.2 we are able to get the following result:

Proposition 4.2.10. *Given an element $\beta \in \mathbb{Z}[\alpha]$ of the form $\beta = a + b\alpha$ for coprime integers a and b, polynomial $f(t) \in \mathbb{Z}[t]$ be a monic, irreducible polynomial of degree d, the norm defined in Definition 4.2.2 can be represented as $N(a + b\alpha) = (-b)^d f(-a/b)$.*

Proof.

$$\begin{aligned} N(a + b\alpha) &= \sigma_1(a + b\alpha) \cdot \sigma_2(a + b\alpha) \cdots \sigma_d(a + b\alpha) \\ &= (a + b\alpha_1) \cdot (a + b\alpha_2) \cdots (a + b\alpha_d) \\ &= b^d \left[(a/b + \alpha_1) \cdots (a/b + \alpha_d)\right] \\ &= (-b)^d \left[(-a/b - \alpha_1) \cdots (-a/b - \alpha_d)\right] \\ &= (-b)^d f(-a/b) \end{aligned} \tag{4.6}$$

□

In some literature, such as [15], the pair (a, b) is represented as $a - b\alpha$, which results in $N(a, b) = b^d f(a/b)$, the homogenized polynomial of $f(t)$. Note that having any of these representations, it is now easy to calculate the norm of a value $\beta \in \mathbb{Z}[\alpha]$.
Having already defined smoothness, we know how to find the relations, and assuming enough have been found, the next step would be applying linear algebra as we would do in the quadratic sieve with vectors $v \in \mathbb{F}_2^n$ where each entry represents the parity of the power of the respective prime dividing $N(a + b\alpha)$ to reach a zero vector. However, we encounter the problem that the fact of $N(a + b\alpha)$ being a square is a necessary condition, but not

sufficient, as we will see now. In order to solve this obstruction we have to relate the norm of an element $\theta \in \mathbb{Q}(\alpha)$ with the norm of an ideal but first we will present a necessary result for squareness in $\mathbb{Z}$ that we will try to make sufficient by the means just mentioned.
First, let

$$R(p_j) = \{r \in \{0, 1, \ldots, p_j - 1\} | f(r) \equiv 0 \mod p_j\}.$$

We now express the norm in a different manner, using the values in $R(p_j)$ in the following way:

$$l_{p_j,r}(a + b\alpha) = \begin{cases} \text{ord}_{p_j}(N(a + b\alpha)) & \text{if } a + br \equiv 0 \mod p_j \\ 0 & \text{otherwise} \end{cases}$$

where $\text{ord}_{p_j}(k)$ is the exponent of the highest power of p_j dividing k. Then, we clearly have, as stated in [17]

$$N(a + b\alpha) = \pm \prod_{p,r} p^{l_{p,r}(a+b\alpha)},$$

which gives rise to the following proposition.

Proposition 4.2.11. *Let U be a set as in (4.2). Then for each prime number p_j and each $r \in R(p_j)$, we have:*

$$\sum_{(a,b)\in U} l_{p_j,r}(a + b\alpha) \equiv 0 \mod 2.$$

To be able to prove this result, we need to understand exactly what the pairs (p, r) are. The new notation of the norm just presented will be more clear once we start working with ideals.
As previously mentioned, our intention to solve this obstruction is to relate the norm of an element of $\mathbb{Q}(\alpha)$ with the norm of an ideal (as defined in Chapter 3). The following two propositions taken from [58, p.116-118] will finally relate the norm of an element of a Dedekind domain to the norm of the principal ideal generated by that element, and prime ideals to prime integers respectively:

Proposition 4.2.12. *The norm of ideals is a multiplicative function that maps ideals of $\mathfrak{O}_\alpha$ to positive integers. Moreover, if $\theta \in \mathfrak{O}_\alpha$ then $N(\langle\theta\rangle) = |N(\theta)|$, where $\langle\theta\rangle$ is the ideal generated by θ.*

Proposition 4.2.13. *Let D be a Dedekind domain. If $\mathfrak{p}$ is an ideal of D with $N(\mathfrak{p}) = p$ for some prime integer p, then $\mathfrak{p}$ is a prime ideal of D. Conversely, if $\mathfrak{p}$ is a prime ideal of D then $N(\mathfrak{p}) = p^u$ for some prime integer p and a positive integer u.*

If we follow the properties of the Dedekind domain described earlier, we have that having $\beta \in \mathfrak{O}_\alpha$, it follows that the ideal generated by β factors uniquely as:

$$\langle \beta \rangle = \mathfrak{p}_1^{e_1} \mathfrak{p}_2^{e_2} \cdots \mathfrak{p}_k^{e_k}$$

for distinct prime ideals $\mathfrak{p}_i$ of $\mathfrak{O}_\alpha$ and positive integers e_i with $1 \leq i \leq k$.
It is now that the two previously stated propositions come into play. Following the properties described above, we can get the following:

$$|N(\beta)| = N(\langle \beta \rangle) = N(\mathfrak{p}_1^{e_1} \cdots \mathfrak{p}_k^{e_k}) = N(\mathfrak{p}_1)^{e_1} \cdots N(\mathfrak{p}_k)^{e_k} = (p_1^{u_1})^{e_1} \cdots (p_k^{u_k})^{e_k} \quad (4.7)$$

for not necessarily distinct primes p_i and positive integers e_i and u_i with $1 \leq i \leq k$.
Using this will be essential in order to determine whether an ideal generated by $a + b\alpha$ is smooth over the factor base. But now, as mentioned in [16], a practical problem arises. The problem comes when storing the prime ideals in a computer, since the representation of them takes a lot of disk space. To solve this problem the GNFS restricts the prime ideals used in the algebraic factor base to ones of a special form. We start with the definition of a first degree prime ideal:

Definition 4.2.14. *A **first degree** prime ideal $\mathfrak{p}$ of a Dedekind domain D is a prime ideal of D such that $N(\mathfrak{p}) = p$ for some prime p.*

We want to represent the first degree prime ideals as pairs (p, r). For that we need to use lemma 3.3.16.
Using the following theorem, we will define what is the representation used for our first degree prime ideals:

Theorem 4.2.15. *The set of pairs (r, p) where p is a prime integer and $r \in \mathbb{Z}/p\mathbb{Z}$ with $f(r) \equiv 0 \pmod{p}$ is in bijective correspondence with the set of all first degree prime ideals of $\mathbb{Z}[\alpha]$.*

Proof. Let $\mathfrak{p}$ be a first degree prime ideal of $\mathbb{Z}[\alpha]$. Then, by definition, the norm of $\mathfrak{p}$ equals p for some prime p. With this, we have that the order of $\mathfrak{p}$ with respect to $\mathbb{Z}[\alpha]$ is the same as the order of $p\mathbb{Z}$ with respect to $\mathbb{Z}$, which implies that $\mathbb{Z}[\alpha]/\mathfrak{p} \cong \mathbb{Z}/p\mathbb{Z}$, i.e. they are isomorphic.
Using the fact that there is a surjective homomorphism of rings $\psi : \mathbb{Z}[\alpha] \to \mathbb{Z}[\alpha]/\mathfrak{p}$ with $\ker \psi = \mathfrak{p}$, and because of the isomorphism between both fields, we can compose with the isomorphism to get a map $\psi : \mathbb{Z}[\alpha] \to \mathbb{Z}/p\mathbb{Z}$, with the same kernel, which means that the elements in $\mathfrak{p}$ map to elements which are divisible by p, and any such integer is the image of an element in $\mathfrak{p}$. Using Lemma 3.3.16, we have that any integer a will be mapped to itself modulo p.
First we will prove the injective correspondence. Let $r = \psi(\alpha) \in \mathbb{Z}/p\mathbb{Z}$,

where ψ is the map generated by the ideal $\mathfrak{p}$. We have that $f(t)$ is a monic, irreducible, polynomial with integer coefficients and α a root, of the form $f(t) = t^d + a_{d-1}t^{d-1} + \cdots + a_0$, then by $f(\alpha) = 0$ we have that $\psi(f(\alpha)) = 0 \pmod{p}$, and hence:

$$\begin{aligned} 0 &\equiv \psi(f(\alpha)) \\ &\equiv \psi(\alpha^d + a_{d-1}\alpha^{d-1} + \cdots + a_1\alpha + a_0) \\ &\equiv \psi(\alpha)^d + \psi(a_{d-1}\alpha)^{d-1} + \cdots + \psi(a_1\alpha) + \psi(a_0) \\ &\equiv r^d + a_{d-1}r^{d-1} + \cdots + a_1 r + a_0 \\ &\equiv f(r) \pmod{p} \end{aligned} \tag{4.8}$$

so that r is a root of $f(t) \pmod{p}$ and the ideal $\mathfrak{p}$ determines the pair (r, p). Note that there does not need to be only one root of f modulo p, and normally there is more than one. However, any such root, with p, will represent the same first degree prime ideal, and will be good for our intentions.

Now we need to prove the surjectivity. Let p be a prime integer and $r \in \mathbb{Z}/p\mathbb{Z}$ with $f(r) \equiv 0 \pmod{p}$. Then there is a natural ring surjective homomorphism, analogous to the note alongside Theorem 3.2.3 that maps polynomials in α to polynomials in r, in particular $\psi(\alpha) \equiv r \pmod{p}$. By Theorem 3.3.15 we have that the kernel of a ring homomorphism is an ideal. Let $\mathfrak{p} = \ker\psi$ so that $\mathfrak{p}$ is an ideal of $\mathbb{Z}[\alpha]$. Since ψ is surjective and $\ker\psi = \mathfrak{p}$ it follows that $\mathbb{Z}[\alpha]/\mathfrak{p} \cong \mathbb{Z}/p\mathbb{Z}$ and hence the norm of the ideal is p, and $\mathfrak{p}$ is therefore a first degree prime ideal. Hence the pair (r, p) determines a unique first degree prime ideal $\mathfrak{p}$. □

The *algebraic factor base* will consist of these kind of ideals, namely, first degree prime ideals of $\mathbb{Z}[\alpha]$. Now that we have a clear view of which are the ideals that we will be using, we have to see how the use of Proposition 4.2.11 will help to determine the squareness in the algebraic factor base. For this we begin by considering the exponents $l_{p,r}$ in Proposition 4.2.11 in a different manner, mainly as group homomorphism $l_{\mathfrak{p}_i} : \mathbb{Q}(\alpha)^* \to \mathbb{Z}$, where $\mathbb{Q}(\alpha)^*$ denotes the multiplicative group of non-zero elements in the field $\mathbb{Q}(\alpha)$, for a fixed prime ideal $\mathfrak{p}_i$.
Following a result derived by Briggs in [16, p.17], we have that these exponent homomorphisms follow the next proposition:

Proposition 4.2.16. *For every prime ideal $\mathfrak{p}_i$ of $\mathbb{Z}[\alpha]$, there exists a group homomorphism $l_{\mathfrak{p}_i} : \mathbb{Q}(\alpha) \to \mathbb{Z}$ that possesses the following properties:*

- $l_{\mathfrak{p}_i}(\beta) \geq 0$ *for all* $\beta \in \mathbb{Z}[\alpha]$.
- $l_{\mathfrak{p}_i}(\beta) > 0$ *if and only if the ideal* $\mathfrak{p}_i$ *divides the principal ideal* $\langle\beta\rangle$.

- $l_{\mathfrak{p}_i}(\beta) = 0$ *for all but a finite number of prime ideals* $\mathfrak{p}_i$ *of* $\mathbb{Z}[\alpha]$*, and* $|N(\beta)| = \prod N(\mathfrak{p}_i)^{l_{\mathfrak{p}_i}}$ *for all prime ideals* $\mathfrak{p}_i$ *of* $\mathbb{Z}[\alpha]$.

As we have said previously, we are interested only in the principal ideals of $\mathbb{Z}[\alpha]$ for the GNFS, furthermore, the elements we are interested in are $\langle a + b\alpha \rangle$. Therefore, the only homomorphisms to be considered are those of the first degree prime ideals of $\mathbb{Z}[\alpha]$.

Theorem 4.2.17. *Given an element* $\beta \in \mathbb{Z}[\alpha]$ *of the form* $\beta = a + b\alpha$ *for coprime integers a and b and a prime ideal* $\mathfrak{p}$ *of* $\mathbb{Z}[\alpha]$*, then the homomorphism* $l_{\mathfrak{p}}$ *of Proposition 4.2.16 corresponding to* $\mathfrak{p}$ *has* $l_{\mathfrak{p}}(\beta) = 0$ *if* $\mathfrak{p}$ *is not a first degree prime ideal of* $\mathbb{Z}[\alpha]$*. Furthermore, if* $\mathfrak{p}$ *is a first degree prime ideal of* $\mathbb{Z}[\alpha]$ *corresponding to the pair* (r, p) *as in Theorem 4.2.15, then:*

$$l_{\mathfrak{p}}(\beta) = \begin{cases} \mathrm{ord}_p(N(\beta)) & \text{if } a \equiv -br \pmod{p} \\ 0 & \text{otherwise} \end{cases}$$

where $\mathrm{ord}_p(N(\beta))$ *denotes the exponent of the prime integer p occurring in the unique factorization of the integer* $N(\beta)$ *into distinct primes.*

We will not prove this result, however, the interested reader might refer to Theorem 3.1.9 of [16]. Little by little we get closer to the final result, but we can already start seeing how indeed a difference of squares using a number field and a finite field is indeed possible. Theorem 4.2.16 is of high importance, since it gives us a simple and fast manner to determine whether a prime ideal corresponding to (r, p) occurs in the factorization of $\langle a + b\alpha \rangle$, mainly if and only if $a \equiv -br \pmod{p}$. This property is what gives rise to the fast sieving in the algebraic base.

Now we have seen how finding a smooth element in the number field sums up to finding an element $a + b\alpha$ such that its norm factors completely over the primes in (r, p) corresponding to the first degree prime ideals of the factor base. For the smoothness bound there is a consensus [42] that it is best chosen empirically, however, in [40], theoretical reasons are given to motivate the choice of the bound being of size:

$$B = \exp((2/3)^{2/3}(\log n)^{1/3}(\log\log n)^{2/3})$$

The next step is producing a square in $\mathfrak{O}_\alpha$, in order to arrive at the difference of squares in (4.3). In the quadratic sieve algorithm this step sums up to representing the powers of the primes in the prime factorization by their parities, in a vector of length S, where S is the length of the factor base. Once the number of smooth numbers has surpassed the size S, we are able to follow, using a matrix, to the final step of finding the difference of squares. In the GNFS, the process is similar. Using the notions of factorization described earlier, and the homomorphisms related to the exponents of the

prime ideal factorization we reach the desired difference of squares. But before this, we need to see how to produce squares in $\mathfrak{O}_\alpha$ since it is not as obvious as for $\mathbb{Z}/n\mathbb{Z}$. The next section will face this starting with the proof of Proposition 4.2.11.

4.3 Creating Squares in $\mathfrak{O}_\alpha$

In the GNFS algorithm, by the form of the difference of squares, we need to create a square both in $\mathbb{Z}[\alpha]$ and $\mathbb{Z}/n\mathbb{Z}$. The condition for the latter will be the same as the one used in the quadratic sieve, i.e., a linear combination of vectors representing the parity of the exponents of the primes dividing the respective number. A square will be reached when a linear combination gives rise to a zero vector $v \in \mathbb{F}_2^S$. We need to find a similar way to represent the squareness of numbers in the number field $\mathbb{Q}(\alpha)$. In other words, we need a way to represent squareness of numbers such that a zero vector in $\mathbb{F}_2^S$ is a necessary and sufficient condition for squareness in $\mathbb{Q}(\alpha)$. We will start proving the necessary condition of Proposition 4.2.11 by using the new notions of exponents derived in the previous section, and we will finish by making this condition sufficient in probabilistic grounds.

Theorem 4.3.1. *If U is a set of pairs of integers (a, b) such that the product of all elements $a + b\alpha \in \mathbb{Z}[\alpha]$ is a perfect square $\theta^2 \in \mathbb{Q}(\alpha)$, then*

$$\sum_{(a,b)\in U} l_{\mathfrak{p}_i}(a + b\alpha) \equiv 0 \pmod 2$$

for all prime ideals $\mathfrak{p}_i$ of $\mathbb{Z}[\alpha]$.

Proof. By Proposition 4.2.16, $l_{\mathfrak{p}_i}$ is a homomorphism from $\mathbb{Q}(\alpha)^* \to \mathbb{Z}$, and hence:

$$\sum_{(a,b)\in U} l_{\mathfrak{p}_i}(a + b\alpha) = l_{\mathfrak{p}_i}\left(\prod_{(a,b)\in U} (a + b\alpha)\right) = l_{\mathfrak{p}_i}(\theta^2) = 2l_{\mathfrak{p}_i}(\theta) \equiv 0 \pmod 2.$$

□

However, this condition is not sufficient, since $\mathbb{Z}[\alpha] \subsetneq \mathfrak{O}_\alpha$ in most of the cases, and therefore, the prime ideals of $\mathbb{Z}[\alpha]$ might differ from the prime ideals of $\mathfrak{O}_\alpha$. For instance, use the following example:
Take $\mathbb{Z}[\sqrt{3}]$ generated by $f(t) = t^2 - 3$, and consider $2 + \sqrt{3}$. We have that $N(2 + \sqrt{3}) = F(2, -1) = (-1)^2((\frac{2}{-1}^2 - 3) = 4 - 3 = 1$, which is square in $\mathbb{Z}$, but if we solve $x^2 = 2 + \sqrt{3}$, we get $x = \pm(\sqrt{6} + \sqrt{2})/2$, which is not in $\mathbb{Z}[\sqrt{3}]$. Furthermore, it is not even in $\mathbb{Q}(\sqrt{3})$.

In order to make this condition sufficient we will use the notion of

quadratic characters but first let us have a look at the following property of perfect squares. If i is an integer such that $i = \ell^2$, then i is also a perfect square modulo every prime p. This is due to Euler's Criterion:

$$\left(\frac{i}{p}\right) \equiv i^{\frac{p-1}{2}} \equiv \ell^{\frac{2(p-1)}{2}} \equiv \ell^{p-1} \equiv 1 \pmod{p}$$

where $\left(\frac{i}{p}\right)$ is again the Legendre symbol defined in Section 2.2. With this property, we have that if i is a perfect square, then it is a perfect square modulo any set of primes, and even if the converse is not true, probabilistically speaking, the more primes we check this property for, the more likely it is that i is a perfect square. To transport this idea to $\mathbb{Q}(\alpha)$ we introduce the following theorem:

Theorem 4.3.2. *Let U be a set of (a, b) pairs such that*

$$\prod_{(a,b)\in U} (a + b\alpha) = \theta^2$$

for some $\theta \in \mathbb{Q}(\alpha)$. Given a first degree prime ideal $\mathfrak{q}$ corresponding to a pair (s, q) that does not divide $\langle a + b\alpha\rangle$ for any pair (a,b) and for which $f'(s) \not\equiv 0 \pmod{q}$, it follows that

$$\prod_{(a,b)\in U} \left(\frac{a + bs}{q}\right) = 1$$

Proof. Let $\psi : \mathbb{Z}[\alpha] \to \mathbb{Z}/q\mathbb{Z}$ be the homomorphism as defined in Theorem 4.2.15, mapping α to s modulo q, and let $\mathfrak{q}$ be its kernel, which is the first degree prime corresponding to the pair s, q. Let $\chi_{\mathfrak{q}} : \mathbb{Z}[\alpha] \setminus \mathfrak{q} \to \{\pm 1\}$, defined as $\chi_{\mathfrak{q}}(\beta) = \left(\frac{\psi(\beta)}{q}\right)$.

As we saw in (4.4), we have that

$$f'(\alpha)^2 \cdot \prod_{(a,b)\in S} (a + b\alpha) = \gamma^2$$

for some $\gamma \in \mathbb{Z}[\alpha]$. By the hypothesis of the theorem that (s, q) (pair representing $\mathfrak{p}$) does not divide $(a + b\alpha)$ for any pair (a, b) nor $f'(s) \equiv 0 \pmod{q}$, we have that $\gamma \notin \mathfrak{q}$ (i.e. γ is not divisible by the ideal $\mathfrak{q}$). The proposition follows by applying $\chi_{\mathfrak{q}}$ to the equation and by using the multiplicative property of Legendre symbol:

$$\chi_{\mathfrak{q}}(\gamma^2) = \left(\frac{\psi(\gamma^2)}{q}\right) = \left(\frac{\psi(\gamma)\psi(\gamma)}{q}\right) = \left(\frac{\psi(\gamma)}{q}\right)^2 = 1$$

$$\begin{aligned}
1 &= \chi_{\mathfrak{q}}(\gamma^2) \\
&= \chi_{\mathfrak{q}}\left(f'(\alpha)^2 \cdot \prod_{(a,b)\in U}(a+b\alpha)\right) \\
&= \left(\frac{\psi\left(f'(\alpha)^2 \cdot \prod_{(a,b)\in U}(a+b\alpha)\right)}{q}\right) \\
&= \left(\frac{\psi(f'(\alpha)^2) \cdot \prod_{(a,b)\in U}\psi(a+b\alpha)}{q}\right) \\
&= \left(\frac{\psi(f'(\alpha)^2)}{q}\right) \cdot \left(\frac{\prod_{(a,b)\in U}\psi(a+b\alpha)}{q}\right) \\
&= \chi_{\mathfrak{q}}(f'(\alpha))^2 \cdot \left(\frac{\prod_{(a,b)\in U}(a+bs)}{q}\right) \\
&= 1 \cdot \prod_{(a,b)\in U}\left(\frac{a+bs}{q}\right).
\end{aligned} \tag{4.9}$$

□

For further reference in the thesis, the set

$$\begin{aligned}
Q = \{(s,q) |\ &\text{for some pairs } (s,q) \text{ with } a+bs \not\equiv 0 \pmod q \\
&\text{for any pair } (a,b) \text{ and } f'(s) \not\equiv 0 \pmod q\}
\end{aligned} \tag{4.10}$$

will be referred as the *quadratic factor base* and the elements $\chi_{\mathfrak{q}}$ as *quadratic characters*.
This does not make the condition sufficient, but as it is mentioned in [17, p.70], it is "overwhelmingly likely" that it will be the case and the more characters are checked, the more likely it will be a square.

Now, for the last step of the matrix reduction, we need to get these results in a manner that will keep our representation with binary vectors. For this, the entry will be a zero if the Legendre symbol is 1 and one otherwise. With this additional information, the relation (a, b) is most probably square if all entries related to a quadratic character in the array are 0. As we explained before, the Legendre symbol is a completely multiplicative function and therefore addition modulo 2 of binary vectors correctly reflects the Legendre symbol of $(a + b\alpha) \cdot (c + d\alpha)$.

Having this in mind, and having proved these properties we can explain how the dependencies will work in order to create the difference of squares by finding a set of pairs (a, b) with $a+bm$ and $a+b\alpha$ smooth. As we previously said, in order to create that, we will use binary vectors. Let k, l, m denote

the sizes of the rational, algebraic and quadratic factor base respectively. The rational factor base will be created by prime numbers, the algebraic factor base by the first degree prime ideals represented by the pairs (p, r) as defined in Theorem 4.2.15. Finally the quadratic factor base will consist of the quadratic characters as previously defined. Then the size of the binary vectors would be $1+k+l+m$. We represent these vectors for the pair (a, b) where $a + bm$ and $a + b\alpha$ are smooth over the rational and algebraic factor base respectively as:

$$\begin{aligned} e_{(a,b)} = \big\{ \big(&\mathrm{Sign}(a+bm), e_{p_i}(a,b), e_{\mathfrak{p}_j}(a,b), e_{\mathfrak{q}_h}(a,b)\big) \mid \\ &i \in \{1,\dots,k\}, j \in \{1,\dots,l\}, h \in \{1,\dots,m\}\big\} \end{aligned} \tag{4.11}$$

where $\mathrm{Sign}(a + bm)$ is the sign of $a + bm$, i.e. the power of the factor (-1), $e_{p_i}(a, b)$ is the parity of the exponents of every prime in the rational base, $e_{\mathfrak{p}_j}(a, b) = l_{\mathfrak{p}_j}(a, b)$ as defined in Theorem 4.2.17 and finally, $e_{\mathfrak{q}_h}(a, b)$ are such that $(-1)^{e_{\mathfrak{q}_h}(a,b)} = \chi_{\mathfrak{q}}(a + b\alpha)$, with $\mathfrak{q}$ uniquely representing (q, s). Having this definition of the binary vectors, and having the properties of Theorem 4.3.1 and Theorem 4.3.2, a non-trivial dependence among these vectors will very likely generate a difference of squares in $\mathbb{Z}[\alpha]$ as required in (4.3).

Chapter 5

Details of the General Number Field Sieve

Until now we have studied the theory behind the GNFS, backing up the explanations with different fields in mathematics. This chapter will explain the parts of the algorithm that were skipped previously for sake of simplicity. The parts we will cover throughout this chapter address a simple way of solving the problem proposed in each section and an introduction to a more advanced method to solve the problem. This chapter will be divided in the four different stages of the GNFS:

- Polynomial Selection
- Sieving
- Matrix Reduction (Linear Algebra)
- Square Root

Out of these four, the sieving step and the square root are well studied steps in the research of the GNFS. The sieving is the most expensive step in terms of time and memory for the GNFS, while the computation of the square root is the last and cheapest part of the algorithm. However, the square root problem is not obvious since it is not the squares in $\mathbb{Q}(\alpha)$ that are required for $x^2 \equiv y^2 \pmod{n}$ but the homomorphic images in $\mathbb{Z}/n\mathbb{Z}$ of their square roots.
Regardless of the importance in the complexity of the sieving step, the ongoing research is more directed to the first part of the algorithm, the polynomial selection. This is due to the influence of the polynomials selected on the likelihood of pairs being smooth. We will present a basic method for this problem, known as the base-m method. Then we will present a better method. Finally, the matrix reduction step is also well studied, and research is mostly directed in the study of how to parallelize

this step. As opposed to the polynomial selection and the sieving step, the matrix reduction is very hard to parallelize.

This chapter will follow the flow of the GNFS, starting with the polynomial selection, continuing with the sieving step followed by the matrix reduction and finally the square root.

5.1 The Polynomial Selection Problem

In the previous chapter it was mentioned that the great innovation in the GNFS that outpassed the quadratic sieve in running time was using a ring other than $\mathbb{Z}/n\mathbb{Z}$. The new ring we consider is $\mathbb{Z}[\alpha]$, which is created with the complex root α of an irreducible polynomial. We discussed that the most time consuming part of the algorithm is the sieving step, i.e. finding enough relations in order to find a non trivial dependence which will lead to a difference of squares (4.3). Therefore there are two ways to approach this situation: first, to find an optimization in the sieving procedure, second, to spend a certain amount of time in making the values $(a + bm)$ and $(a + b\alpha)$ more likely to be smooth, and therefore ease the task of the sieving procedure.

The determination of whether a value is smooth or not (which we will consider to be part of the sieving step) is also time consuming and the running time of the algorithm could be improved here. We will see however, that while it has been optimized from the first one used in Dixon's method, it is not much different than the one of the Quadratic Sieve. It is indeed the selection of the polynomial (which is directly related to the field in which we work, and to the norm since $N(a, b) = b^d f(a/b)$) which will define the running time of the sieving step to a big extent.

We can divide the polynomial selection methods in two groups, the *linear search* and the *non-linear search* methods. The linear methods consist in finding two polynomials with one linear and the other of higher degree, while the non-linear looks for two non-linear polynomials. In the first group we can find methods as the *base-m* method, Murphy's improvement of the latter or Kleinjung's method. As for *non-linear* methods, the most studied proposals are from Zimmerman [50] or Williams [68]. In this thesis we discuss only linear methods.

The first step of the algorithm, in order to find a difference of squares, starts by looking for a pair of polynomials, f_1, f_2, which share a common

root m modulo n, so that:

$$\prod_{(a,b)\in U} \phi_1(a+b\alpha_1) = \prod_{(a,b)\in U} a+bm = \prod_{(a,b)\in U} \phi_2(a+b\alpha_2)$$

where α_1, α_2 are the complex roots of polynomials f_1 and f_2 respectively, and ϕ_1, ϕ_2 are the homomorphism of f_1, f_2 respectively as defined in Theorem 4.1.1.
We begin by an obvious method for generating such polynomials, called the *base-m method*, which is a simple way of defining polynomials with the required properties and which we used so far in Chapter 4.

Definition 5.1.1 (The base-m Method [45]). *The **base-m representation** of n is as follows:*

$$n = \sum_{i=0}^{d} a_i m^i,$$

with $a_i \in \{-\lfloor \frac{m-1}{2} \rfloor, \lceil \frac{m-1}{2} \rceil\}$. We define the polynomials as follows:

$$f_1(t) = \sum_{i=0}^{d} a_i t^i, \text{ and } f_2(t) = t - m.$$

Note at this point, that the homogenized polynomial of $f_2(t)$ is $F_2(a,b) = a - mb$.

In this way we have a nonlinear polynomial $f_1(t)$ and a monic and linear polynomial $f_2(t)$. If the linear polynomial is non-monic, then the size of the nonlinear polynomial can be greatly reduced as mentioned in [36]. Note that in order to make the nonlinear polynomial non-monic, we need to keep the property of it having a root at m modulo n, and therefore if it is of the form $f_2(t) = b_1 t - b_0$, then b_0/b_1 must be equal to m.

However this is not the best method to find a good polynomial which will help us find more relations. What the polynomial selection step tries to do in more advanced methods is find a set of pairs of polynomials $(f_1(t), f_2(t))$, satisfying the conditions of being irreducible, coprime and sharing a root modulo n, and then choose the most suitable pair. By suitable polynomials we mean polynomials that have a norm $N_1(a,b)$ and $N_2(a,b)$ (where N_1 and N_2 are $F_1(a,b)$ and $F_2(a,b)$ respectively) simultaneously smooth for many coprime integer pairs (a,b) in the sieving region. This is what defines the *yield* of a polynomial (or also referred as the yield of $N_i(a,b)$).
There are two factors that influence the yield of $N_i, i \in \{0,1\}$, the *size* and the *root properties*. The size refers to the values taken by N_i, and small values are more likely to be smooth over the factor base. The root properties describe how the roots of N_i affect the likelyhood of smooth

values. To study the latter, we check whether the homogeneous form of a polynomial has many roots modulo small p^k, for p prime and $k \geq 1$. If it is the case, then the values taken by $F_i(a, b)$ "behave" as if they where smaller than they actually are [69].
We will start with a sketch of Montgomery and Murphy's method [21], but beforehand, we define what is known as the sieving region.

Definition 5.1.2. *The pairs (a, b) searched during the sieving step to satisfy equation (4.2) are inspected in some region $\mathcal{A} \subset \mathbb{Z}^2$. This region is known as* ***sieving region****.*

For the case of line sieving (which we will explain in next section), the sieving region is of the form $[-A, A] \times [1, B] \cap \mathbb{Z}^2$ for some A and B.
The reason why we introduce these terms in this section, is because in the polynomial selection step, one of the tests made to each pair of polynomials is the yield of $N_i(a, b)$ for $i \in \{1, 2\}x$. We now use the sieving region to present three modifications to a pair of polynomials which keeps them coprime and sharing a root modulo n if they had these properties previously.

- Re-defining the pair of polynomials as (f_1', f_2') with $f_i'(t + l) = f_i(t)$
- Adding a $\mathbb{Z}[t]-$multiple of one polynomial to another
- Changing the shape of the sieving region $\mathcal{A}$, while maintaining its area.

Note that a rectangle of a given area only changes $s = A/B$ (where A, B are the length of its sides) which is named the skewness of a sieving region. The last modification together with the first one preserve the value of α_i while the second one might change it. Nevertheless, the shared root modulo n and keeping both polynomials coprime will be preserved.
Montgomery and Murphy's method can be divided in two main stages. The first where different pairs of polynomials are being searched, and the second where, among these pairs, one is selected. This method is a *linear* method, and therefore the effort is focused on the first polynomial $f_1(t)$. The method used for the factorization of the first 512-bit RSA modulus [21] starts with input n, and chooses a degree d and a bound $a_{d,\max}$ denoting the maximum value of the leading coefficient. Begin by setting $a_d = 0$. Then one formulates the following steps:

(i) Choose the next good a_d, i.e., choose an a_d with small prime divisors. If the bound $a_{d,\max}$ has been reached, terminate.

(ii) Set $m = \left\lfloor \sqrt[d]{\frac{n}{a_d}} \right\rfloor$. Start determining the base-m expansion of n, but if you encounter that a_{d-2} is too big, return to step (i).

(iii) Next step is determining the complete base-m expansion from which you obtain f_1 and, as in the base-m method, we define $f_2 = t - m$.

Using the three methods defined above, optimize in order to get a pair of polynomials with small coefficients.

(iv) Finally, using a sieve as will be further explained, determine which $f_1 + cf_2$ have good root properties, and output $(f_1 + cf_2, f_2)$. Go to the first step.

In the last step, c is a polynomial of degree one, which is used to find polynomials $f_1(t) + cf_2(t) = f_1(t) + (j_1 t - j_0)f_2(t)$, where j_1 and j_0 are small compared to a_2, a_1 respectively and where the resulting polynomial has good root properties modulo many small primes.

In the first three steps, this method is trying to make the coefficients a_i of the non-linear polynomial as small as possible. This makes the *size* of F_i smaller, and therefore having a better *yield*. Then, in step (iv) the root properties are checked. As explained in [21], the effect of the root properties is measured using a parameter while for the smoothness of the values of $F_1(a, b)$ and $F_2(a, b)$ a probabilistic method is used. Among the top ranked candidates, a short sieving is performed in order to choose the best suited pair of polynomials. This short sieving consists in running the previously describe sieve (for a small number of pairs (a, b) in order to determine which pair is more interesting for the next step).

Another polynomial selection method to mention, is Kleinjung's polynomial selection algorithm. He makes an improvement of the method just explained by changing the two first steps and keeping the last two as Montgomery and Murphy do. One of the changes in these two steps is to substitute the creation of a monic polynomial f_2 and replace it by a non-monic one. However, we will not go into detail of this algorithm in this thesis, and the interested reader may refer to [36].

5.2 The Sieving Step

We will stick with the choice of $f_2 = x - m$ with norm $N_2(a, b) = a + bm$ and the sieving region $\mathcal{A} = [-u, u] \times [1, u]$. To keep notation as in the previous chapter, we refer to $N_1(a, b)$ as $N(a, b)$, and to $N_2(a, b)$ simply as $a + bm$. After having chosen a "good" polynomial, the GNFS proceeds with the sieving step. This is, finding a set U, such that equation (4.2) is satisfied, by sieving through different values of (a, b) pairs. The construction of such a set proceeds in two steps. In a first place, the algorithm sieves through pairs in order to find (a, b) pairs with $a + bm$ and $a + b\alpha$ smooth over the rational and algebraic factor base respectively. Secondly, after having stored the parity of the powers of the primes dividing $a + bm$ and $N(a + b\alpha)$ and their exponents in binary vectors, it calculates the quadratic

characters as described in Section 4.4 for every (q, s) in the quadratic base, and places the binary result in its respective entry of the vector, to increase the possibility of indeed making it a square in $\mathbb{Z}[\alpha]$. As we saw before, this does not ensure that we will indeed find a square, but at least it increases the probabilities considerably. After this, as we will see in the next section, one uses linear algebra in order to find a dependency among these pairs to construct squares.
Recall that the pairs (a, b) must be relatively prime. Let us define the set of "accepted" pairs.

$$\mathcal{B} = \{(a, b) : a, b \in \mathbb{Z}, \gcd(a, b) = 1, |a| \leq u, 0 < b \leq u\}. \tag{5.1}$$

where u is a large positive integer to be chosen depending on n.
Now, it is clear from (4.2) that we have two different cases in our search for the set U. First we have the rational side, i.e., finding pairs such that $\prod_{(a,b)\in U}(a+bm)$ is a square, and second the algebraic side, which consists in finding pairs such that $\prod_{(a,b)\in U}(a+b\alpha)$ is a square. As presented in [17] we will proceed by dividing these two cases and explaining two different sieving procedures. First the rational sieve, followed by the algebraic sieve. To be able to find squares in $\mathbb{Z}[\alpha]$ and $\mathbb{Z}/n\mathbb{Z}$ simultaneously we just need to keep the pairs which are in the intersection of these two groups i.e., we need to find pairs (a, b) which are smooth over both the rational and the algebraic factor base.
For determining whether the pairs (a, b) are smooth over a factor base, a simple method consists in keeping the value of b fixed and calculating smoothness for all values of a ranging from $-u \leq a \leq u$. This could be done with trial division, by trying to divide all values of $a + bm$ or $N(a, b)$ by all primes in the factor base and then accepting for the ones terminated with 1. However, there are methods to solve this problem known as sieving method where the total number of pairs (a, b) checked is much smaller than the numbers of pairs checked with trial division. We begin by showing a procedure to solve the problem in the rational and algebraic side, which is very similar to the previously explained in the Quadratic Sieve. Then, we finish the chapter with a sketch of a more complex method, known as the Lattice Sieving.

5.2.1 The Rational Sieve

An alternative procedure for finding a smooth set $(a, b) \in \mathbb{Z}^2$ will be explained in this subsection. We need to find a set T_1:

$$T_1 := \{(a, b) \in \mathcal{B} : a + bm \text{ is } B\text{-smooth}\}$$

In order to find this set, we use the fact that a prime p_j divides $a+bm$ if and only if $a + bm \equiv 0 \pmod{p_j}$. The steps of the rational sieve go as follows:

(i) Create sieve arrays by creating for each $b < u$ an array with $2u$ positions consisting of the values $a + bm$ for $-u < a < u$.

(ii) For every p_j in the factor base:

- sieve through the elements of each array such that $a = -bm + kp_j$ for some integer k and $-u < a < u$.
- replace the value of $a + bm$ by $(a + bm)/p_j$.
- in order to remove all powers of p_j, we do a similar procedure to the one of the QS. We begin the sieve for $p_j^2, p_j^3, \ldots, p_j^e$, where e is the first integer with p_j^e too big for the sieving array. And we replace again $(a + bm)$ by $(a + bm)/p_j$.

(iii) Every location containing 1 or -1 is smooth over the factor base and can be added to T_1.

The reason of doing this method instead of trial division with the primes of the factor base is to avoid useless division where the prime does not divide $a+bm$. However, the method just described also has a problem. Many pairs, after all divisions of the respective diving primes, might end with a value greater than one, which means that the value $a+bm$ is divisible by a number higher than the smoothness bound, and we have again produced unnecessary divisions. Then the same as in the QS method is used, using the fact that the calculation of the logarithm base a prime p_j is a computationally cheaper calculation compared to division:

(i) Create sieve arrays by creating for each $0 < b < u$ an array with $2u$ positions consisting of the values $\log(a + bm)$ for $-u < a < u$.

(ii) For every p_j in the factor base:

- sieve through the elements of each array such that $a = -bm + kp_j$ for some integer k and $-u < a < u$.
- replace the value of $\log(a + bm)$ by $\log(a + bm) - \log(p_j)$.
- in order to remove all powers of p_j dividing $(a + bm)$ we do the same procedure as in the previously presented sieve.

(iii) Every location containing 0 is most likely to be smooth over the factor base. However, before adding it to T_1, for every location containing 0 we divide $a + bm$ by every p_j dividing it, to make sure it is indeed B-smooth. It can then be added to T_1. At this point we can also begin by creating the binary vectors representing the parity of each prime of the rational factor base dividing $a + bm$. These vectors will be completed in the rational sieve step.

This will avoid unnecessary divisions. A similar check will be implemented for the algebraic sieve.

5.2.2 The Algebraic Sieve

Finding a square in $\mathbb{Z}[\alpha]$ will be slightly different than the previously defined procedure, but the idea is to follow as much as possible the strategy of the rational sieve. As we have seen in the section of algebraic number theory, we define $\beta \in \mathbb{Z}[\alpha]$ to be B-smooth if its norm $N(\beta)$ is B-smooth. One way to calculate the norm is by calculating the homogeneous polynomial of $f(x)$, $F(a,b) = (-b)^d f(-a/b)$. In other notation:

$$N(a + b\alpha) = a^d - c_{d-1}a^{d-1}b + \cdots + (-1)^d c_0 b^d$$

where c_i is the i-th coefficient of f_1. Following the steps of the rational sieve, we must find a set

$$T_2 := \{(a,b) \in \mathcal{B} : a + b\alpha \text{ is } B\text{-smooth}\}$$

As it has been shown in the section of Algebraic Number Theory in Chapter 3, the first degree prime ideals, $\mathfrak{p}$, of $\mathfrak{O}_\alpha$ are in one-to-one correspondence with pairs (p,r), where p are prime numbers satisfying $N(\mathfrak{p}) = p$ for some first degree prime ideal and r is a root of f modulo p. Also, it has been proven that $a + b\alpha$ is divisible by $\mathfrak{p}$ if and only if $N(a + b\alpha) \equiv 0 \pmod{p}$, which is similar to saying that $a \equiv -br \pmod{p}$ for r as defined above. This later property is what will be used for our algebraic sieve.
Note: If $b \equiv 0 \mod p$ then there does not exist an a with $(a,b) \in \mathcal{B}$ and $N(a + b\alpha) \equiv 0 \mod p$, due to the condition of a and b being coprime. If $b \equiv 0 \mod p$, then $a \equiv -r0 \mod p$ and therefore a and b share a factor, contradicting our condition of $\gcd(a,b) = 1$.
Now, as for the previous algorithm, we do the following:

(i) Create sieve arrays by creating for each $0 < b < u$ an array with $2u$ positions consisting of the values $\log(N(a + b\alpha))$ for $-u < a < u$.

(ii) For every p in the factor base:

- sieve through the elements of each array such that $a = -br + kp$ for some integer k and $-u < a < u$.
- replace the value of $\log(N(a + b\alpha))$ by $\log(N(a + b\alpha)) - \log(p)$.
- we again repeat this procedure for the powers of p as in the rational sieve.

(iii) Every location containing 0 is most likely to be smooth over the factor base. However, before adding it to T_2, for every location containing 0, we perform division to $N(a + b\alpha)$ for every p dividing it, to make sure it is indeed B-smooth. It can then be added to T_2. In this step we continue the construction of our binary vectors with the parity of the prime (p,r) dividing $N(a + b\alpha)$.

Our final set of accepted pairs will be the intersection $T_1 \cap T_2$.
In order to handle the fact that the square does not necessarily fall in $\mathbb{Z}[\alpha]$, we just need to calculate the Legendre symbol of $\left(\frac{a+bs}{q}\right)$ for pairs (q, s) in the quadratic factor base and $(a, b) \in T_1 \cap T_2$ as defined in Section 4.3. Then the binary vectors will finally be constructed by adding the binary value related to the output of the Legendre symbol, and in this manner, we have finished constructing the binary vectors used for the linear algebra step. Before proceeding with the linear algebra step, we will give an overview of the Lattice sieving.

5.2.3 Lattice Sieving

This method is the one used by the implemented version of the GNFS considered in this thesis, YAFU [18]. It was published by John Pollard in 1991 [47]. We will give an overview of the method, but we will not go into detail due to the need to understand lattice theory, which we will not cover in this thesis.
The goal of this stage of the GNFS is to find enough pairs (a, b) such that $a + bm$ and $a + b\alpha$ are smooth.

Pollard's algorithm is divided in the following steps:

(i) Divide the rational factor base $\mathcal{F}$ into two parts:

$$\begin{aligned} S:&\ \text{the small primes:} && p \leq B_0 \\ M:&\ \text{the medium primes:} && B_0 < p \leq B_1 \end{aligned}$$

where the ratio between both bounds should be between 0.1 and 0.5 as specified in [47]. We also use a set of large primes:

$$L:\ \text{the large primes:} \quad B_1 < p \leq B_2$$

where B_2 is much larger.

(ii) Choose a region $\mathcal{A}$ from which the pairs (a, b) will be chosen (known as the sieving region).

(iii) Fix a prime q from M, and sieve only the (a, b) pairs from $\mathcal{A}$ with:

$$a + bm \equiv 0 \pmod{q}$$

For the set of the pairs

$$T = \{(a, b) | a + br \equiv 0 \mod p, \text{ with } (r, p) \text{ a unique ideal, and } (a, b) \in \mathcal{A}\}$$

the sieve goes through two stages. In a first place to determine whether $a + bm$ is smooth, in this case it only checks for primes $p < q$. Secondly to know whether $N(a, b)$ is smooth checking with all primes of S and M. With this method, the smoothness check step accepts a value of $a + bm$ and $N(a, b)$ which factors completely in the sets S and M and at most one factor in L.
The idea of this method is to create a lattice with the pairs (a, b) and sieve through the points of this lattice depending on certain properties. To get more understanding of how the lattice is created, and how we are sieving through these elements, knowledge of lattice theory is required, and the interested reader may refer to [19]. In [47] it is claimed that the number of integers sieved by this method is much less than the linear sieve by a factor:

$$\lambda = \sum_{q \in M} \frac{1}{q} \approx \frac{\log(1/k)}{\log(B_1)}$$

and still get most of the solutions found by the other method.

5.3 The Linear Algebra Step (Matrix Reduction)

Once enough relations have been found, and having the related binary vectors associated with them, a non-trivial dependency must be found, i.e. solving the system formed by $Ax = 0$, where A is the matrix formed with the binary vectors. To create this matrix, we create binary vectors in the way explained in Section 4.3.
A naïve way to do this, would be Gaussian elimination, but this would mean a time complexity of $O(u^2 s)$, where u is the number of relations, and s is the number of non-zero entries. As it turns out there are more advanced algorithms which are used in the GNFS such as the block Wiedemann or block Lanzcos algorithms. We will use this section to describe Gaussian elimination, followed by a brief explanation of the Lanzcos algorithm. In theory, a matrix with one relation more than the size of the binary vector would suffice to find a dependency among the vectors, but in practice one is interested in creating a matrix which will contain a high number of dependencies, not "just above" the length of the binary vectors. This is due, partly, to the ending of a trivial relation of the two squares, resulting in a trivial factor of n (either 1 or n itself).

5.3.1 Gaussian Elimination

In this section we explain the Gaussian elimination for matrices. In our case, these matrices will contain only elements in $\mathbb{Z}/2\mathbb{Z}$, and therefore the algorithm will be much simpler, as we will see. We start however with the general case.

As input, we have a system of linear equations of the form $Ax = w$, with $A \in \mathbb{Z}^{c\times\ell}, x \in \mathbb{Z}^{\ell}, w \in \mathbb{Z}^{c}$. The algorithm goes as follows:

- Create a matrix of the form $[A|w]$.
- Use the following row operations to reach *row echelon form*:

  Type 1: Exchange two rows.

  Type 2: Multiply a row by a scalar not equal to zero.

  Type 3: Add to one row a scalar multiple of another.

- Once reached row echelon form we are ready to determine whether there are no solutions, one, or infinitely many, and therefore solve the linear system, in case it is feasible. This is done by reaching reduced row echelon form, and then solving for each element.

Now follows an example of a matrix in row echelon form:

$$\left(\begin{array}{ccc|c} 2 & 4 & 1 & 5 \\ 0 & 3 & -1 & 2 \\ 0 & 0 & 2 & 4 \end{array}\right)$$

To solve this, we start by reaching reduced row echelon form:

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -7/6 \\ 0 & 1 & 0 & 4/3 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

which then corresponds to:

$$\begin{aligned} x_1 &= -7/6 \\ x_2 &= 4/3 \\ x_3 &= 2 \end{aligned}$$

and the system of linear equations is solved.
For the case of the GNFS it is slightly different. Our interest in the linear algebra step is to determine which linear combinations of vectors reach a zero vector, where all entries in the matrix are in $\mathbb{Z}/2\mathbb{Z}$. In this case, we have as input $A \in \mathbb{Z}_2^{\ell\times c}$ and we want to solve the following system of equations:

$$Ax = \mathbf{0}.$$

We refer to the columns of A as a_i for $i \in \{1, \ldots, c\}$. To reach the upper triangular matrix we perform addition among the rows[1] until a matrix in

[1]Note that in the case of $\mathbb{F}_2$ the only row operation is addition among two rows, since multiplication by a scalar $\not\equiv 1 \pmod 2$ will result in a zero vector

row echelon form form is reached:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,c} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,c} \\ 0 & 0 & \ddots & \cdots & \vdots \\ 0 & 0 & \cdots & a_{c-1,c-1} & a_{c-1,c} \\ 0 & 0 & \cdots & 0 & a_{c,c} \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & & & & \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

If we keep track of the row operations done to produce the zero vectors at the bottom of the matrix, then we know which vectors can be combined in order to produce a zero vector. In order to keep track of the row operations, we can use a binary vector of size ℓ and we introduce a 1 for every row appearing in the addition to create a zero vector. Since the entries in the rows of A represent the exponents of the primes in the factor base dividing different $(a+bm)$ and $(a+b\alpha)$, the addition among rows represents multiplication among different $(a + bm), (a + b\alpha)$. Hence the resulting vectors which are zero modulo two, represent that the exponents have even power, and therefore we have probably reached our difference of squares.
However, the Gaussian Elimination is very expensive computationally, and for the implementations of the GNFS, alternatives as block Wiedemann's [66] or block Lanczos [44] algorithms are used.

Before starting defining the Lanczos algorithm, it is important to note that in the case of the GNFS, the matrices that have to be reduced are sparse. For example, in [21] which factored a 512-bit RSA modulus, the matrix M had $\ell \approx 7 \times 10^6$ and on average 62 non-zero coefficients per row, and for the RSA-768 factorization in 2009 [37], the matrix M had $\ell \approx 2 \times 10^8$ and on average 144 non-zero coefficients per row. This guides us to work with algorithms which apply to sparse matrices.

5.3.2 Standard Lanczos Algorithm

For the explanation of the standard Lanczos algorithm, we will assume throughout the section that the field we are working in is $\mathbb{F}_p$. There exists also a Lanczos algorithm to solve linear systems in the ring of integers, but for our purpose it is more interesting to concentrate in finite fields. It is also important to mention that the Lanczos algorithm is defined for symmetric matrices. In order to generalize this condition, the system of equations will be solved for $M = A^T A$. However, the matrix M would actually never be computed.

For the simplicity of the explanation of the algorithm, we will assume that we are trying to solve the following equation:

$$Ax = w,$$

with $A \in \mathbb{F}_p^{\ell \times \ell}, w \in \mathbb{F}_p^{\ell}$. And therefore, this will be our input. For clarification, we start by showing the steps of the algorithm and then we follow with a proper explanation:

(i) Define a sequence of vectors v_i which satisfy the following:

$$v_i \text{ is } A\text{-orthogonal to } v_j \text{ whenever } i \neq j, \tag{5.2}$$

$$V_i = \langle v_0, \ldots, v_i \rangle. \tag{5.3}$$

where $\langle \cdot \rangle$ means the span of vectors.

(ii) Proceed by induction to calculate v_{i+1} as follows:

$$v_{i+1} = Av_i - \sum_{j \leq i} \frac{v_j^T A^2 v_i}{v_j A v_j} v_j.$$

(iii) We end defining x in the following way:

$$x = \sum_{i<k} \frac{v_i^T w}{v_i A v_i} v_i$$

where k is the index when $v_k = 0$.

Now we proceed to the explanation of these steps. Let the inner product defined from A be $v_i^T A v_j$ for $i \neq j$. We say that two vectors are **A-orthogonal** if their inner product defined from A is zero, and we call a vector **A-isotropic** if it is A-orthogonal to itself.
The Lanczos algorithm focuses on the sequence of Krylov subspaces, which are defined as:

$$V_i = \langle v_0, Av_0, A^2 v_0, \ldots, A^i v_0 \rangle,$$

where $v_0 = w$. This gives us that $V_i \subseteq \mathbb{F}_p^{\ell}$
It is clear that the sequence of subspaces is strictly increasing, until a certain point where it remains stationary, i.e.: $V_0 \subseteq V_1 \subseteq V_2 \subseteq \cdots \subseteq V_s = V_{s+1} = V_{s+2} = \cdots$.

The first step is to define

$$V_i = \langle v_0, Av_0, A^2 v_0, \ldots, A^i v_0 \rangle,$$

and then define a sequence of vectors v_s for $0 \leq s \leq i$ which satisfy (5.2) and (5.3).

Theorem 5.3.1. *The vectors created with the following recurrence:*

$$v_{s+1} = Av_s - \sum_{j \leq s} \frac{v_j^T A^2 v_s}{v_j^T A v_j} v_j \tag{5.4}$$

with $v_0 = w$ and $0 \leq s < i$, form an orthogonal, linearly independent basis that span V_i.

Proof. We will begin by showing that these vectors are A-orthogonal. First, we begin showing the base case, namely that v_0 and v_1 are orthogonal:

We have v_0 and $v_1 = Av_0 - \frac{v_0^T A^2 v_0}{v_0^T A V_0} v_0$. Then:

$$v_0^T A A v_0 \; - \; \frac{v_0^T A^2 v_0}{v_0^T A V_0} v_0^T A v_0 \quad = \quad v_0^T A A v_0 \; - \; v_0^T A^2 v_0 \quad = \quad 0 \tag{5.5}$$

therefore, the base case is satisfied. Now we proceed to show by induction that v_{i+1} is orthogonal to all v_j,$j \leq i$, and therefore our induction hypothesis is that it holds for all such v_j. Let v_{i+1} as in (5.4). Then:

$$\begin{aligned} v_j^T A v_{i+1} &= v_j^T A \left(A v_i - \sum_{s \leq i} \frac{v_s^T A^2 v_i}{v_s^T A v_s} v_s \right) \\ &= v_j^T A A v_i - \sum_{s \leq i} \frac{v_s^T A^2 v_i}{v_s^T A v_s} v_j^T A v_s \\ &= v_j^T A A v_i - \frac{v_j^T A^2 v_i}{v_j^T A v_j} v_j^T A v_j \\ &= 0 \end{aligned} \tag{5.6}$$

and we have that the vectors are orthogonal.

Our next step is to show that a set of such vectors is linearly independent. We assume that $v_h^T A v_j = 0$ for $h \neq j$ and $v_j^T A v_j \neq 0$. Let $a_0 v_0 + \cdots + a_i v_i = 0$ with $a_s \in \mathbb{Z}$ for $0 \leq s \leq i$. Then $A(a_0 v_0 + \cdots + a_i v_i) = 0$ and hence, for any j with $0 \leq j \leq i$ we have:

$$\begin{aligned} 0 = v_j^T \cdot 0 = v_j^T A(a_0 v_0 + \cdots + a_i v_i) = v_j^T (a_0 A v_0 + \cdots + a_i A v_i) = \\ a_0 (v_j^T A v_0) + \cdots + a_i (v_j^T A v_i) = a_j (v_j^T A v_j). \end{aligned} \tag{5.7}$$

Since $v_j^T A v_j \neq 0$, we have that $a_j = 0$, and since the value of j is arbitrary, it holds for all j. Hence the set created by vectors formed in such a manner is linearly independent.

Now we need to show that (5.3) is satisfied. For $i = 0$ it is easy to see. We assume it is true for i, i.e.:

$$V_i = \langle v_0, \ldots, v_i \rangle.$$

Note that:

$$V_{i+1} = \langle v_0 \rangle + AV_i = \langle v_0 \rangle + AV_{i-1} + \langle Av_i \rangle = V_i + \langle Av_i \rangle.$$

It is easy to see that indeed $v_{i+1} \in V_{i+1}$ by condition (5.4), and therefore $\langle v_0, \ldots, v_{i+1} \rangle \subseteq V_{i+1}$. But since the dimension of V_{i+1} and the dimension of $\langle v_0, \ldots, v_{i+1} \rangle$ are equal (due to the independence of the vectors v_j for $j \in \{0, \ldots, i+1\}$), we have that $V_{i+1} = \langle v_0, \ldots, v_{i+1} \rangle$. Note that we mentioned before that at some point the dimension of V_i does not increase, therefore we have that

$$V_i + \langle Av_i \rangle = V_i$$

which means that Av_i is a linear combination of v_j for $j \leq i$. Therefore $Av_i = \sum_{j \leq i} a_j v_j$ for not all $a_j = 0$. Then, using the definition of v_{i+1}:

$$\begin{aligned} v_{i+1} = Av_i - \sum_{j \leq i} \frac{v_j^T A^2 v_i}{v_j^T A v_j} v_j = \sum_{j \leq i} a_j v_j - \sum_{j \leq i} \frac{v_j^T A (\sum_{j \leq i} a_j v_j)}{v_j^T A v_j} v_j = \\ = \sum_{j \leq i} a_j v_j - \sum_{j \leq i} a_j \frac{v_j^T A v_j}{v_j^T A v_j} v_j = \sum_{j \leq i} a_j v_j - \sum_{j \leq i} a_j v_j = 0 \quad (5.8) \end{aligned}$$

and therefore when the dimension of V_{i+1} does not increase, neither does the one of $\langle v_0, \ldots, v_i, v_{i+1} \rangle$. □

The way to calculate v_s for $1 \leq s \leq i$ can be simplified as follows:

$$v_{s+1} = Av_s - c_{s+1,s} v_s - c_{c+1,s-1} v_{s-1},$$

where,

$$c_{s+1,s} = \frac{v_s^T A^2 v_s}{v_s^T A v_s}, \qquad c_{s+1,s-1} = \frac{v_{s-1}^T A^2 v_s}{v_{s-1}^T A v_{s-1}},$$

due to the orthogonality of the basis. Furthermore, we have that $Av_{s-1} \in v_s + V_{s-1}$, which implies $v_{s-1}^T A^2 v_s = v_s^T A v_s$, and therefore we can do the following further simplification:

$$c_{s+1,s-1} = \frac{v_s^T A v_s}{v_{s-1}^T A v_{s-1}}.$$

So it is clear that the values of v_s can be easily computed, but there is one more condition which is necessary in order to be able to compute the values of $c_{s+1,s}$ and $c_{s+1,s-1}$, namely that

$$\forall j \leq s, v_j^T A v_j \neq 0. \quad (5.9)$$

For this last condition, we will assume it is true until some index k, so that $v_k = 0$, which means $V_k = V_{k-1}$, and therefore the space will not increase with new vectors.
The final step is to define x as:

$$x = \sum_{i<k} \frac{v_i^T w}{v_i^T A v_i} v_i \tag{5.10}$$

The following theorem proves that constructing x in such a way gives indeed a solution to $Ax = w$.

Theorem 5.3.2. *Let $\langle v_0, v_1, \ldots, v_{k-1}\rangle$ be a basis of the Krylov subspace generated by $v_0 = w$*

$$V_{k-1} = \langle v_0, Av_0, \ldots, A^{k-1}v_0\rangle$$

with $v_h A v_j = 0$ for $h \neq j$ and $v_h A v_h \neq 0$ for $h \in \{0, \ldots, k-1\}$. Then the vector defined as in (5.10) satisfies $Ax = w$.

Proof. We begin by showing that $v^T(Ax - w) = 0$ for all $v \in V_{k-1}$. Notice that:

$$v_j^T Ax = v_j^T A \left(\sum_{s<k} \frac{v_s^T w}{v_s^T A v_s} v_s \right) = \frac{v_0^T w}{v_0^T A v_0} v_j^T A v_0 + \cdots + \frac{v_j^T w}{v_j^T A v_j} v_j^T A v_j + \\ + \cdots + \frac{v_{k-1}^T w}{v_{k-1}^T A v_{k-1}} v_j^T A v_{k-1} = v_j^T w \tag{5.11}$$

for all $v_j \in \{v_0, \ldots, v_{k-1}\}$. This gives us that

$$v_j^T Ax - v_j^T w = v_j^T (Ax - w) = 0$$

for all $v_j \in \{v_0, \ldots, v_{k-1}\}$. Since $\{v_0, \ldots, v_{k-1}\}$ is a basis of V_{k-1}, we have that

$$v^T Ax - v^T w = v^T (Ax - w) = 0$$

for all $v \in V_{k-1}$.
Now note that $Av_h \in V_{k-1}$ for $0 \leq h < k-1$. By our previous step, we then have that $(Av_h)^T(Ax - w) = 0$, and therefore $v_h^T(A(Ax - w)) = 0$ because A is self-adjoint[2].
Since we have that the Krylov subspace is generated by $v_0 = w$, we have that $w \in V_{k-1}$. By construction of x we have also that $Ax \in V_k = V_{k-1}$ and therefore $Ax - w \in V_{k-1}$, which means it is a linear combination of $\{v_0, \ldots, v_{k-1}\}$, i.e. $Ax - w = a_0 v_0 + a_1 v_1 + \cdots + a_{k-1} v_{k-1}$, therefore:

$$0 = v_h^T(A(Ax - w)) = v_h^T(A(a_0 v_0 + \cdots + a_{k-1} v_{k-1})) \\ = a_0(v_h^T A v_0) + \cdots + a_h(v_h^T A v_h) + \cdots + a_{k-1}(v_h^T A v_{k-1}) \\ = a_h(v_h^T A v_h) \tag{5.12}$$

[2]It is equal to its conjugate transpose (in this case, symmetric).

for $a_0, \ldots, a_{k-1} \in \mathbb{Z}$, and therefore $a_h = 0$. Note that h was chosen arbitrarily, therefore $a_0, \ldots, a_{k-1}$ are all equal to zero and hence $Ax - w = 0$. $\square$

Therefore we have a method which will find a solution for $Ax = w$. However, there is the possibility that we have $v_k^T A v_k = 0$ without $v_k = 0$, i.e. v_k is an A-isotropic vector. In this case, the algorithm fails.

Let d be the number of nonzero elements per row in matrix A. Then the cost per iteration of Lanczos algorithm is $O(d\ell)$ to multiply by A and $O(\ell)$ for the other vector arithmetic. The total running time is $O(d\ell^2) + O(\ell^2)$.

5.3.3 Lanczos in GF(2)

However, when applying Lanczos over the field $\mathbb{Z}/2\mathbb{Z}$, there is an important issue that must be addressed. The problem is that half of vectors are $A-$isotropic, i.e.: $v^T A v = 0$. This forces condition (5.9) to be replaced. To solve this problem, Montgomery uses sequences of orthogonal subspaces. So instead of using the A-orthogonality of vectors, he divides the space $(\mathbb{Z}/2\mathbb{Z})^n$ into a sequence of subspaces $\{\mathcal{W}_i\}_{i=0}^{m-1}$ which are pairwise orthogonal. The condition (5.9) is replaced by the condition

$$\nexists \text{ non zero } z \in \mathcal{W}_i \text{ such that } z^T A z_i = 0 \quad \forall z_i \in \mathcal{W}_i.$$

He also uses the binary nature of the matrix in its favour, since by having a matrix consisting of 0s and 1s, a numerical computation can be converted to logical operations, which means that the computer can operate on N vectors at a time (N being the word size of the computer in question). On a high level, the algorithm starts by selecting an $\ell \times N$ matrix Y over GF(2), computes AY and attempts to find an $\ell \times N$ matrix X s.t $AX = AY$, which makes us end up with $A(X - Y) = 0$. Therefore the columns of $(X - Y)$ will be random vectors of the null space of A. These vectors of $(X - Y)$ will also be vectors in the null space of B, where $A = B^T B$.

Independently of what method is chosen to do the matrix reduction, at the end we will end up with a vector x that will determine which combination of vectors will yield a zero vector. This will give away the relations that have to be selected in order to find a product which will yield a square. As previously shown in the thesis, the zero vector in the positions of the quadratic characters will augment the probability of yielding a square. Once we have this, we can follow to calculate the square root.

5.4 Computing the Square Root

In this section, we will describe the last part of the algorithm, namely, computing a square root. It is now considered to be an "easy" step of the GNFS for computing complexity. Moreover this problem has been solved since the beginning of research for the GNFS. Even if the running time has been optimized, the most used algorithms date back to 1993, one included in 'The development of the Number Field Sieve' [40] by Couveignes [24], and another one by Montgomery [43] presented in [63]. In our Thesis we will not go through any of these algorithms due to its complexity. However, the interested reader might refer to the algorithm presented by Peter L. Montgomery in [43] which was used for the latest RSA-challenge factorization record of a 768-bit RSA modulus [37]. Some alternatives to this method can be found in [61].
This step of the algorithm, after having found a set U such that (4.2) holds, proceeds to compute the respective square roots. For the rational case, it is a simple task, since we know the prime factorization of the product. However for the algebraic side it is more complicated. Even if the prime ideal factorization of $\prod_{(a,b)\in U}$ is known, this is not useful to find its square root. What is known as the brute force method, consists in computing a root of the polynomial $t^2 - \beta^2$ in $\mathbb{Q}(\alpha)$ [57]. However, methods such as the previously mentioned by Montgomery try to reduce the size of β^2 by means of complex approximations which results in better practical performance.

Once the square root is calculated, the algorithm outputs the factor of the number inputed. What could go wrong now is that γ does not yield a square or that the difference of squares yields a trivial factors (n or 1). This is the reason why number of relations we seek is higher than the number of columns of the matrix, in order to have more than one solution for the linear equation and therefore have the choice with other dependencies of vectors in the matrix to yield the zero vector.

Chapter 6

The GNFS Algorithm and the Security of Cryptographic Keys

This chapter consists of an analysis of the state of the art of factorization together with the best implementations of the GNFS. A discussion of two recent vulnerabilities in RSA encryption (FREAK) and Diffie Helmann key exchange (LogJam) will follow. Both of the attacks were made using the GNFS, for factorization and discrete logarithms respectively. This chapter will go through both attacks and explaining briefly the Number Field Sieve for Discrete Logarithms. We will continue with an explanation of Telsacrypt ransomware showing that thanks to a flaw in its implementation and today's power of factorization, it was possible to recover compromised files without paying the ransom. The chapter will finish by presenting what key size is considered to be secure for RSA cryptosystem.

6.1 State of the Art of the GNFS

The GNFS is the most efficient algorithm known for factoring integers of more than 100 decimal digits, with heuristic complexity [57] of

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}}+O(1)\right)(\log(n))^{1/3}(\log(\log(n)))^{2/3}\right)=L_n\left[1/3,\sqrt[3]{\frac{64}{9}}\right].$$

Since the algorithm was developed by Pollard in 1993 the idea of "hard to factor numbers" has totally changed. In 1977, in the original memo of Rivest, Shamir and Adleman presenting RSA [52], the following was said: *"We estimate that factoring a number of that size [125-digit] could require 40 quadrillion years"*

In this paper it was mentioned that choosing a composite number which factors into two prime numbers of length 40 (decimal digits) seemed to be satisfactory at present. This means using a number n of 80 digits, resulting in bit size slightly more than 256-bit, which nowadays is an accessible to factor number to anyone willing to spend a few minutes.

The estimates of Rivest, Shamir and Adleman were made at a time where the best factoring algorithm was Pollard's method (explained in Section 7). Later on, with the Quadratic Sieve and the Number Field Sieve development, these estimates were rising, as Carl Pomerance said in [49]:
"Twenty years ago, at the dawn of the continued factoring algorithm of Brillhart and Morrison, factoring hard 50-digit numbers seemed barely possible, while 100-digit numbers could not even be dreamed about. Ten years ago, when my quadratic sieve factoring algorithm first began to enjoy some success, we indeed did dream of 100-digit numbers, and within a few more years, they were falling regularly. Today, with the number field sieve, our dreams have moved on the hard 150-digit numbers".

This was in 1994, and after the factorization in 2009 of a 768-bit number, the public record made by a research group [37] is set to 232 digits and it would not be unreasonable to believe that efforts made by nation states are ahead of academic efforts.
It is to be expected that the computer power will keep developing in the future which means that the advances in factorization will as well follow. As we can see in Figure 6.1, the factorization is always advancing hand in hand with the computer resources available[1].

It is important to mention that in this section we will only analyse the records and factorizations of RSA numbers, which are of the form $n = p \cdot q$ with p, q random primes of similar size. Otherwise, the Special Number Field Sieve algorithm could be used to factor numbers up to 1199-bits of the form $2^n - 1$ [38].

The RSA laboratories put up the so known *"RSA factoring challenge"* in order to encourage research groups to factor RSA-numbers[2]. We will mention some of the numbers present in the RSA factoring challenge [9]. In 1999 Cavallar, Dodson, Lenstra, Lioen, Montgomery, Murphy, te Riele, Aardal, Gilchrist, Guillerm, Leyland, Marchand, Morain, Muffett, Putnam, Putnam and Zimmermann factored the first 512-bit number [21] in seven months and hundreds of computers. Currently the record is a 768-bit number factored in 2.5 calendar years, made by the group of Kleinjung,

[1] Note that for these results the same algorithm, GNFS, was used. Some modifications were made to it through time, but the complexity remains almost the same

[2] These are the numbers presented in Figure 6.1.

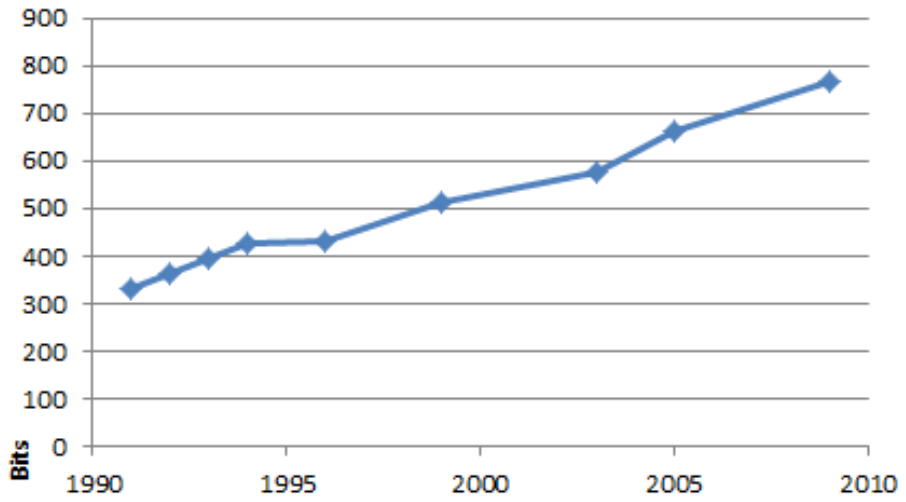

Figure 6.1: Advances in factorization through time, with the size of keys in bits (y-axis) for RSA numbers. Data taken from [9]

Aoki, Franke, Lenstra, Thomé, Bos, Gaudry, Kruppa, Montgomery, Osvik, Te Riele, Timofeev and Zimmermann presented in [37]. A very important advance in making factoring accessible is the paper of Valenta, Cohney, Liao, Fried, Bodduluri and Heninger in [62]. The reason of the importance of the paper, is the ease in which a non-expert can factor a 512-bit key, by using the Amazon Elastic Compute Cloud (EC2). As we mentioned before, the part of the GNFS which has most investigation dedicated is the polynomial selection, but there is also a huge amount of research on how is it possible to parallelize the steps of the GNFS, and this is exactly what has been optimized in [62].

There are many implementations of the number field sieve which have been published, such as CADO-NFS [60], msieve [46] or ggnfs [39]. These implementations have been used by different software packages, such as *factMsieve.pl* [28] which uses the Msieve polynomial selection and postprocessing and the ggnfs sieving, or YAFU which uses the same as factMsieve but does not use a perl script.
The CADO-NFS implementation has been in many of the latest factorizations. However, in this particular case of factorization using EC2 [62], CADO-NFS was used only for the stages of polynomial selection and sieving, while the matrix step and square root where performed with msieve. This decision was made by the group of the University of Pennsylvania due to better results of the msieve implementation for the linear algebra step in Amazon EC2.

It is important to note that even if the record in factorization currently is at 768-bits, this has been made by research groups five years ago so it would be cautious to stop using 1024-bit due to the difference in available resources between research groups and governmental agencies such as the NSA. It has not been confirmed that such agencies are indeed capable of doing so (or that have interest to do it...), but one is never 'too' careful.

6.2 LogJam and FREAK Attack

The reason why both attacks will be commented in the same section, is due to the fact that both vulnerabilities are due to a very similar reason and both of the attacks are against the TLS protocol. In a last layer view, what these attacks do is downgrade the exchange keys to export cipher suites, which moves away from the "believed to be" secure keys of today's ciphersuites, to less secure ones. Let us go a bit further in this explanation:

In short, export cipher suites were created by the US government in order to avoid showing their believed to be better encryption to the outside of the US [11]. Therefore, in order to distribute crypto out of the U.S., the security had to deliberately be weakened. But it was still possible to use strong crypto in the U.S., which raised the problem to U.S. servers to have to accept both strong and weak cipher suites. Therefore, the SSL negotiation mechanisms allowed a server to accept weakened cryptography, and this is where the vulnerabilities were found in order to produce both attacks. At the end of last century, the U.S. removed most of its export policies. Nevertheless, the export cipher suites never really disappeared. Thanks to this and a flaw in TLS, a *man in the middle* (MITM) attack could weaken the agreed cryptography between the client and the server, without the client being aware of this. If there is a connection where the client is vulnerable and the server supports export ciphersuites (either RSA or Diffie-Hellman) then the attacker can force down the security of the connection as shown in Figure 6.2.

The LogJam attack consists of downgrading the DH exchange keys to DHE_EXPORT, which consists of 512-bit primes. Similarly the FREAK attack consists in downgrading the RSA security to 512-bit RSA numbers. We now focus on the LogJam attack by briefly describing the Number Field Sieve for discrete logarithms, and explain how the attack works.

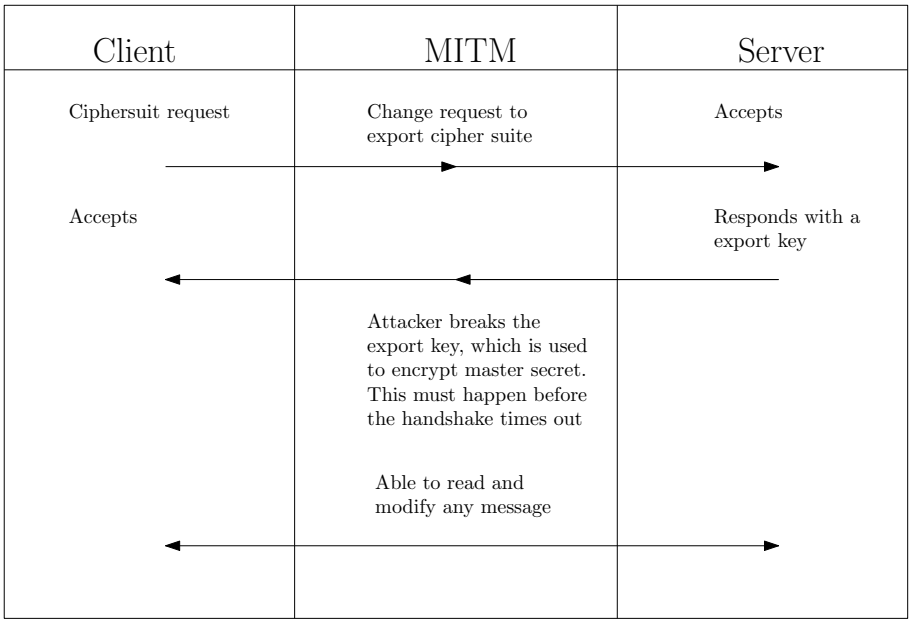

Figure 6.2: MITM attack overview

6.2.1 LogJam attack

There are several ways of using the downgrade to DHE_EXPORT, as explained in [6], but the scope of this chapter is to focus in the theoretical side of these attacks, i.e. the usage of the number field sieve for the attack.

Let us first understand how the Diffie Hellman key exchange works in practice.
Alice and Bob agree on a prime p and a generator g, of a multiplicative subgroup modulo p. Alice sends g^a mod p, Bob sends g^b mod p and both compute g^{ab} mod p. The DH assumption says that the most efficient cryptanalytic attack is computing the discrete log of g^a or g^b, and the DL assumption says that this is a hard problem. For this, the most efficient algorithm is the Number Field Sieve for Discrete Logarithms. Explaining this algorithm is out of the scope of this paper, but we will briefly see how this algorithm is divided in order to understand the attack.
The algorithm can be divided into four stages:

- Polynomial Selection
- Sieving
- Linear algebra
- Descent

Source	Popularity	Prime
Apache	82%	9fdb8b8a004544f0045f1737d0ba2e0b 274cdf1a9f588218fb435316a16e3741 71fd19d8d8f37c39bf863fd60e3e3006 80a3030c6e4c3757d08f70e6aa871033
mod_ssl	10%	d4bcd52406f69b35994b88de5db89682 c8157f62d8f33633ee5772f11f05ab22 d6b5145b9f241e5acc31ff090a4bc711 48976f76795094e71e7903529f5a824b
(others)	8%	(463 distinct primes)

Figure 6.3: Top 512-bit DH primes for TLS. 8.4% of Alexa Top 1M HTTPS domains allow DHE_EXPORT, of which 92.3% use one of the two most popular primes, shown here. Taken from [12]

The algorithm is different from the factorization one, but we can find many similarities. The important fact we need to point out in this chapter, without going into detail, is the following:
"The algorithm can be divided into two sub processes. The first (pre computation) will consist on the first three stages, which are only dependent on the number p, and finally the descent, that is the only one that involves $y = g^x mod\ p$*."*[12]
And here is where the problem comes. Let us look at Figure 6.3. What the attacker does, is calculate all the pre-computation steps for many 512-bit primes p, known to be used in many servers (this is the time consuming part of the algorithm), and then hope that the DHE happens with one of these primes. In case it does, the attacker, being a man in the middle, will have access to the information $y = g^x \mod p$ needed to extract the secret key, and the last step, using $y = g^x \mod p$ takes on average 70 seconds [3] and therefore, the extraction of the secret key can be computed before the handshake has to finish. The attacker can decrypt the private key of the symmetric encryption scheme and will have access to read and modify any message between the client and the server.

The FREAK attack downgrades the RSA key to a 512-bit key. However, we have seen that a 512-bit key can be factored in around four hours, but an attacker clearly does not have so much time during a TLS key exchange. The reason why this attack is successful is due to an implementa-

[3] Data taken from [12]

tion shortcut that generates RSA server keys only on application start, not on every connection. This same key could last for hours, days or months. It is due to this that an attack can be successful and the MITM can have access to the private key of the RSA key encryption. As Nadia Heninger explained in the talk "Security in Times of Surveillance", the factorization needs to be made in real time, due to the change of the ciphersuite name from "RSA" to "RSA_EXPORT" [31].

6.3 TeslaCrypt Malware

The reason why I will be presenting this attack is because the factorization utility we are using in CrypTool 2 was used in order to exploit the ransomware[4] flaw to recover data. Unfortunately the flaw was fixed and solved by the newer version of TelsaCrypt. However we will discuss how the ransomware works and what were the flaws that made possible to recover the files of the victim.

When a computer is infected with the TelsaCrypt Ransomware it encrypts all files of the victim using the AES (Advanced Encryption Scheme) encryption algorithm, which is a symmetric encryption scheme. Each time TeslaCrypt was run, a new AES key was generated. Then the ransomware would open an HTML page in the user's browser explaining the instructions to follow, which mainly were paying a ransom of a certain amount in Bitcoins in exchange of the key to decrypt the files. In this description of the virus, it also explained that the keys were encrypted using RSA-2048, so that in case the victim tried to find a way to solve the problem, she would realize how infeasible it is. However, this was only information to misguide the victim, since the real encryption method used was AES in the following way:

- TeslaCrypt creates a symmetric AES key to encrypt your files.
- It then asymmetrically ECDH-encrypts the AES-key, sending the private key to the operators of the ransomware (in order to be able to decrypt the key once the payment has been effectuated).
- It starts encrypting the files in the system one by one.

The ransomware uses one AES key per session, which means that if the infection is interrupted, new keys will be generated. However, this is irrelevant for the scope of this chapter.
The AES [26] and ECDH [3] encryption schemes are safe encryption schemes,

[4]Type of malicious software designed to block access to a computer system until a sum of money is paid.

when used with caution and when properly implemented. The implementation of the encryption schemes was properly done by the attackers, the problem was how they were storing the keys. As we previously said, the AES key was encrypted using the ECDH scheme, which is a public key encryption scheme, and then stored along each encrypted file[5]. When a computer is infected, the ransomware generates both the public and private keys of ECDH, and then sends the private key to the attackers. Then the AES key is encrypted with ECDH and stored in the encrypted file. Until now, everything is good for the attacker, and probably quite complicated to restore any encrypted information. However the attackers did not stop there regarding information of the private key. In order to maintain their credibility (by making sure they could decrypt the files for the person paying the ransom) they stored the key of AES together with the secret of the ECDH in the files, so that if the private key did not reach the attackers upon generation, they could still restore the files, and therefore fullfil their promise to the victim who paid. To see this better refer to Figure 6.4.

It is with session_ecdh_secret_mul (which obfuscates the AES key) where

Session Key Generation

session_priv	Secret key to encrypt files (AES). Not saved anywhere
session_pub	AES key encrypted with ECDH Stored in encrypted files
session_ecdh_secret	Value used to restore the private key in case of loss. Generated with a master private key.
session_ecdh_secret_mul	session_priv $*$ session_ecdh_secret Stored in the encrypted file

Figure 6.4: An overview of the session key generation

the victim can benefit from today's power of factorization. Using a factoring algorithm, the values can be extracted, and therefore proceed to generate the AES key with the help of Googulator Python scripts (that also created scripts to accomplish the decrypting automatically [29]). The session_ecdh_secret_mul is around a 512-bit number. As we have seen before, 512-bit hard numbers (with two factors of similar size) are possible to factor for a non-expert from home, but that is not even close to the problem of

[5]First lines of the encrypted file will be the encrypted AES key.

factoring session_ecdh_secret_mul. These numbers not only are 512-bit, but they are not hard to factor, or in other words, their factors are not of similar size, nor there are only two. Therefore other methods which do not depend on the size of n, but on the size of their smaller factors such as ECM[6], will do the factorization much more efficiently. Therefore factorization can be produced from any modern computer using programs such as factMsieve [28] or YAFU. In many of the help pages we can see that these are the main programmes proposed (for easy use) see [4] or [5]. In the YAFU sourceforge page [18] we can see that once the method to decrypting the infected files was brought to light, the number of downloads hugely increased as seen in Figure 6.5:

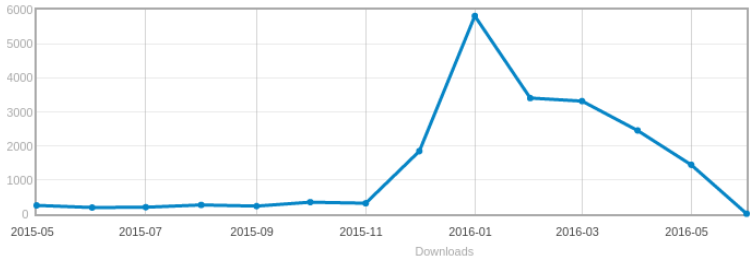

Figure 6.5: Number of YAFU downloads in the last year. The method to decrypt infected files was given in December 2015

And as said before, the rest of the procedure to decrypt the files (from the key extraction till the actual decryption of the files) is done by codes developed by [29].

6.4 Conclusion

Overall, we could say that usage of RSA modulus of 1024-bits is safe against a not super powerful attacker. However it is necessary to start using 2048-bit keys since cryptography also must be applied against super powerful attackers. Today's security flaws, as we have seen in the past chapter, are not the cryptosystems themselves, but rather the storage of the private key, the size of the later, the key exchange, the creation of secure modulus (not as random as we would wish) or the implementation of cryptosystems.
The National Institute of Standars and Technology (NIST) of the United States describes in [13] that the key strength to be used from 2014 through to 2030, is as shown in Figure 6.4 presented in [13, p.67]. With *"applying"* or *"processing"* it is meant whether it is used to encrypt data or to decrypt data respectively. *'Deprecated'* means that it can be used if risk is accepted,

[6]We give an explanation of the ECM method in Section 7.2

<table>
<tr><td colspan="2">Security Strength</td><td>2011 through 2013</td><td>2014 through 2030</td><td>2031 and beyond</td></tr>
<tr><td rowspan="2">80</td><td>Applying</td><td>Deprecated</td><td colspan="2">Disallowed</td></tr>
<tr><td>Processing</td><td colspan="3">Legacy use</td></tr>
<tr><td rowspan="2">112</td><td>Applying</td><td rowspan="2">Acceptable</td><td rowspan="2">Acceptable</td><td>Disallowed</td></tr>
<tr><td>Processing</td><td>Legacy use</td></tr>
<tr><td>128</td><td rowspan="3">Applying/ Processing</td><td colspan="3" rowspan="3">Acceptable</td></tr>
<tr><td>192</td></tr>
<tr><td>256</td></tr>
</table>

Figure 6.6: Security strength through different time frames

while *'disallowed'* means that it should not be used for applying cryptographic protection. *'Legacy use'* means that the key can be used to process cryptographically protected data. Finally with *'acceptable'* it is meant that the security strength is not known to be insecure. The first column represents the bits of security where an algorithm that takes 2^X operations to break is said to have a "security strength of X bits" or to provide "X bits of security". In particular, the security of keys for integer factorization cryptography (IFC) (e.g. RSA), are as follows

Bits of security	IFC
80	n=1024
112	n=2048
128	n=3072
192	n=7680
256	n=15360

which means, first of all, that 1024-bit keys should be left aside and secondly that 2048-bit keys should be useful for at most the next 15 years.

It is also with big concern that the developing of quantum computers is seen from the cryptographic community. In 1994 Peter Shor developed an algorithm for integer factorization for usage in quantum computers [55]. This algorithm runs in polynomial time (as opposed to the GNFS which runs in sub-exponential time) which implies a huge risk for RSA cryptosystem. However, this does not mean that the only solution is to cross fingers and hope that no quantum computer is ever invented. There is much post-quantum cryptography research ongoing [7] which studies schemes which are claimed to be safe even against quantum computers.

The direction of the research is in the development of such schemes which are reliable in terms of security but at the same time meet practical performance requirements. What is of big importance is to manage to start implementing and using such schemes before quantum computers are invented. The question is how soon do we need to worry. As very clearly explained by Andreas Hülsing in the talk of *Security in times of surveillance*, using Theorem 1 of slide 9 in [32]:

Theorem 1: If $x + y > z$, then worry.

Here x is the time in years we need for our keys to be secure, y the number of years it would take to re-tool the infrastructure with post-quantum cryptographic schemes and z the time until a quantum computer is invented. Since we cannot tell what z is, it is a must to start thinking about the solutions for a possible post-quantum infrastructure.

Chapter 7

CrypTool 2 Plug-in

Besides giving a theoretical description of the GNFS, my thesis consists in developing a plug-in for the Windows application CrypTool 2 (CT2). CrypTool is a widespread open-source e-learning program for cryptography and cryptanalysis. For this task it is necessary to know C# and .xaml (programming languages), because CT2 is developed as .NET application building its user interface with the Windows Presentation Foundation (WPF). The new plug-in is aimed for factorization using the GNFS – in addition it also offers several other factoring algorithms.

This chapter introduces the application as a whole, followed by an explanation of all the algorithms available in the plug-in which have not yet been explained yet. A description of the tools I used for the development of the plug-in will follow. Then I will compare the functionality and performance of the factoring plug-ins already available in CT2 with the one I developed. I will finish the section with a presentation of the results and the functionality of the plug-in.

7.1 Introduction to CrypTool 2

Back in 1998, with the development and use spread of computer throughout financial companies, the financial institute Deutsche Bank decided to launch a world-wide awareness program for its employees about computer security, for which they developed CrypTool 1 (CT1). This cryptologic e-learning program was intended to be interactive with the users, which made it one of a kind at the time. It was then used by the German Information Security Agency in 2002, again, for awareness of the citizens, and finally, in 2003, together with the Technical University Darmstadt, it became open source software. Since then, the CrypTool project has been in close cooperation with many universities. What started to be a company awareness program became a cryptographic e-learning tool for everyone which offers GUI-based

user interaction with state-of-the-art cryptographic methods. By 2007 CT1 was available in 5 different languages. The wide-spread usage encouraged the developing team to magnify the project and develop .NET and Java versions, as these offered more modern technologies and allowed a plugin-based architecture. So there CT2 was born [1]. CT2 offers a more modern user interface which is based on WPF, and implements the concept of visual programming (the according prototype has been developed by the University of Koblenz). So CT2 can visualize the reactions of the algorithms in real time. The cryptanalytic tools available in CrypTool 2 help to analyze or even break classical and modern ciphers [2]
One of the most used ciphers today is RSA (Rivest, Shamir, Adleman [53]). As a cryptographic learning tool, CT2 already offered an interactive plug-in of the RSA cryptosystem, as well as factorization with the Quadratic Sieve and factorization with brute force. Nevertheless, it lacked a plug-in having the algorithm of the state-of-the-art in factorizing big RSA-moduli, the GNFS. (To be accurate, there was a draft of a plug-in using msieve [46] and GGNFS [39], but this was not stable). The plug-in presented below offers to use GNFS, but also to use other factoring algorithms such as the Quadratic Sieve or Lenstra's ECM among others.
Before this, I will present the other seven methods which are now available and have not yet been explained.

7.2 Factoring Algorithms besides GNFS, QS, or Fermat's Made Newly Available in CT2

This section will discuss the algorithms available in the plug-in which are not explained in the first part of this thesis. The aim of this section is not to go into deep theoretical detail of any method, but rather explain with simple concepts the algorithms that have not been explained yet. For the two more complex methods (Multi-Polynomial Quadratic Sieve (MPQS) and Self-Initializing Quadratic Sieve (SIQS), an understanding of the quadratic sieve (see Section 2.2) should be sufficient for the usage of the plug-in, and we will limit the explanations to referencing. We will start with the methods which are easier to understand from a mathematical perspective, such as Pollard's $p-1$ method, Pollard's ρ method, or Shank's method. We will finish with two concepts a bit more complex: the Lucas sequences in order to explain William's $p+1$ method, and elliptic curves, in order to explain the Elliptic Curve Method (ECM).

(1) Pollard $p-1$ method

From Fermat's little theorem, we know that if p is an odd prime, then $2^{p-1} \equiv 1 \pmod{p}$. From this, we can derive that if $p-1|M$, where $M \in \mathbb{Z}$,

then $2^M \equiv 1 \pmod{p}$. Therefore, we have that, if p is a prime factor of a number n, p will divide $\gcd(2^M - 1, n)$, and this is the result that Pollard makes use in his $p-1$ method for factorizing [25].

Algorithm 7.3. [25] Input is a composite number n, and we select a high bound B. The output is either none or a non-trivial factor of n.

1. Calculate the list of all primes up to a previously selected bound B. This may be done using the sieve of Eratosthenes [10].
2. For each prime selected in the previous step, calculate the maximum integer c_i such that $p_i^{c_i} \leq B$.
3. Let $a = 2$
4. For each prime p_i compute $a = a^{p_i} \mod n$ c_i times.
5. Calculate $\gcd(a - 1, n)$.
6. If $g \neq 1$ or n, output g. It is a non trivial factor.
7. Else, replace a by next prime and go to step 4.

An upper bound can be given to a (say 10) in order for the algorithm not to run indefinitely in case it does not find a non-trivial factor. Alternatively, one can increase B.

(2) Pollard's rho meethod

Another method introduced by J. Pollard is the rho (ρ) method. Let f be a random function from S to itself, where S is a finite set, and consider the following sequence:

$$s, f(s), f(f(s)), \ldots$$

Then, it is obvious that at some point, the values will be repeated. This is from what the algorithm takes the name of "rho", since if we graph this behavior, the whole picture looks like the shape of rho (ρ) where the intersection point is when the two values of f are the same. If the finite set is $S = \{0, \ldots, p-1\}$ and the function $f(x)$ is random enough, we expect the sequence $(f^{(i)}(x))$, where $f^{(i)}(x)$ is the ith iteration, to repeat within $O(\sqrt{p})$ steps. Suppose we want to factor $n = pq$. We clearly do not have access to p, but if we define F to follow the same function as f but taken modulo n instead of modulo p then clearly $F(x) \equiv f(x) \mod p$, thus $F^{(j)}(a) \equiv F^{(k)}(a) \pmod{p}$, and therefore $\gcd(F^{(j)}(a) - F^{(k)}(a), n)$ will be divisible by p. If the output of the gcd is not n, we have found a non-trivial factor.

For this factoring method, the most common function f used is $f(x) = x^2+1$.

The algorithm consists of the following steps:

Algorithm 7.4. As input, number n to factor.

1. Choose random $s \in [0, n-1]$;
   $U = V = s$;
   Define function $f(x) = x^2 + 1 \pmod{n}$;

2. $U = f(U)$;
   $V = f(f(V))$;
   $g = \gcd(U - V, n)$;
   if $g = 1$ repeat step;

3. if $g = n$ choose different s and start from step 1. If this step is reached many times, try other polynomial as for example $f(x) = x^2 - 1$ or $f(x) = x^2 + 3$.

4. else: return g

This simple algorithm, as explained before, depends on the value of the smallest prime factor of n, and therefore is a good algorithm to find small factors. Furthermore, the Pollard rho method stores only the value n and the current values of U, V, so it uses very little space. However, when the number n is big and has no small values, it is better to try other algorithms such as ECM or QS.

(3) Shanks' method

Shanks' square forms factorization method (sometimes called squfof) also makes use of the idea of a difference of squares. Before we present the algorithm start defining a binary quadratic form, as in Definition 5.2.2 [22, p.220]:

Definition 7.4.1. *A **binary quadratic form** f is a function $f(x, y) = ax^2 + bxy + cy^2$, where a, b, c are integers. It is denoted as $f = (a, b, c)$. $D = b^2 - 4ac$ is the **discriminant** of f.*

Definition 7.4.2. *(Definition 5.6.4 [22, p.258] Let $f = (a, b, c)$ be a quadratic form. Let $a \neq 0$ and b be integers. The **rho function** is defined by:*

$$\rho(f) = \left(c, r(-b, c), \frac{r(-b, c)^2 - D}{4c}\right)$$

where $r(b, a)$ is defined by $r \equiv b \pmod{2a}$ with $-|a| < r \leq |a|$ if $|a| > \sqrt{D}$ and $\sqrt{D} - 2|a| < r < \sqrt{D}$ if $|a| < \sqrt{D}$.

Definition 7.4.3. *Let $f = (a, b, c)$ be a quadratic form with discriminant D. We say $f = (a, b, c)$ is* ***reduced*** *if we have $|\sqrt{D} - 2|a||< b < \sqrt{D}$.*

Regarding the quadratic forms, I will not spend more time in them because they are not really relevant to see what the algorithm does. Nevertheless, in order to understand why it works, knowledge of quadratic forms is needed. For more information on them, refer to [22].

Algorithm 7.5. *[22, Alg. 8.7.2]* As input, the number n which is not either a square or a prime.

1. If $n \equiv 1 \mod 4, D := n, d := \lfloor\sqrt{D}\rfloor, b := 2\lfloor(d-1)/2\rfloor + 1$,
   else $D := 4n, d := \lfloor\sqrt{D}\rfloor, b := 2\lfloor d/2\rfloor$.
   Set $f := (1, b, (b^2 - D)/4), Q := \emptyset, i := 0, L := \lfloor\sqrt{d}\rfloor$.
   Q will be used as our queue.
2. $f = (A, B, C) = \rho(f), i = i + 1$. If i is odd, go to step 5.
3. Test if A is a square. If it is, let $a = +\sqrt{A}$ and if $a \notin Q$ go to step 6.
4. If $A = 1$ break, and output that the algorithm did not find a non-trivial square form.
5. If $|A| \leq L$, set $Q = Q \cup \{|A|\}$ and go to step 2.
6. *Note: at this point we have found a non-trivial square form.*
   Set $s := \gcd(a, B, D)$. If $s > 1$, output s^2 as a factor of n. Else $g := (a, -B, aC)$ and apply the rho function to g until g is reduced, and write $g = (a, b, c)$
7. Let $b_1 := b$ and $g = (a, b, c) = \rho(g)$. If $b_1 \neq b$, repeat step 7, else output $|a|$ if a is odd, else, output $|a/2|$.

As we can see, this algorithm is very simple in terms of programming, since its code length is really small, and this is one of the reasons why it is attractive. Another reason is that is works exclusively with reduced quadratic forms, of discriminant at most a small multiple of n, so values a, b, c are of order $n^{1/2}$. The downside of the algorithm is that, unlike Pollard's rho method or ECM, it depends only on size of n, therefore when the number to factor becomes bigger and bigger, Shanks' method will not be useful even if factors are small. For a more detailed explanation of the algorithm, please refer to [54].

(4) Williams p plus 1

This method was discovered by H.C. Williams and it is based on the $p - 1$ method. To understand how this algorithm works, we will first define

the following Lucas sequences (which is what replaces the multiplications modulo p in our previous algorithm):

$$U_0 = 0, U_1 = 1, V_0 = 2, V_1 = u$$

$$U_r = uU_{r-1} - U_{r-2}, V_r = uV_{r-1} - V_{r-2} \text{ for } r \geq 2,$$

where the only condition for u is that it is bigger than 2.

Let $D = u^2 - 4$. Williams proved that an odd prime p divides both $\gcd(n, U_M)$ and $\gcd(n, V_M - 2)$ whenever M is a multiple of $p - \left(\frac{D}{p}\right)$, where $\left(\frac{D}{p}\right)$ is the Legendre symbol [67]. Of course, the problem is that we cannot compute the Legendre symbol, since we do not have the value of p. What we do is compute the gcd's with different values of u, to have higher probability to have $\left(\frac{D}{p}\right) = -1$.

Firstly, instead of computing $a^m - 1$, we compute $V_m \pmod n$, and instead of $\gcd(a^m - 1, n)$, we compute $\gcd(V_m - 2, n)$.
In this step, what we are interested in doing is to find $V_{m\ell}$ from V_ℓ (index multiplication), and to do that, we'll use the following two formulas:

$$V_{2\ell} = V_\ell^2 - 2 \text{ (duplication formula)}$$

$$V_{m+\ell} = V_m V_\ell - V_{m-\ell} \text{ (addition formula)}$$

Now, we will see the approach we can use to perform index multiplication by a natural number M.
First we represent the value of M in binary form. Once we have this we proceed using a method known as Montgomery's ladder. It goes as follows:

Algorithm 7.6. Montgomery's ladder.

1. $V_A \leftarrow V_0$ and $V_B \leftarrow V_\ell$.
2. Then for every position of the binary form:

   If it is zero then

   $$V_B \leftarrow V_{A+B} \text{ and } V_A \leftarrow V_{2A}$$

   else

   $$V_A \leftarrow V_{A+B} \text{ and } V_B \leftarrow V_{2B}$$

3. Return V_A.

So for instance, 23, in binary form, is 10111, and therefore the index multiplication to calculate $V_{23\ell}$ will go as follows:

Algorithm 7.7. *Example:*

1. $$V_A \leftarrow V_0; V_B \leftarrow V_\ell$$
2. $$V_A \leftarrow V_l; V_B \leftarrow V_{2\ell}$$
3. $$V_B \leftarrow V_{\ell+2\ell}; V_A \leftarrow V_{2\ell}$$
4. $$V_A \leftarrow V_{3\ell+2\ell}; V_B \leftarrow V_{6\ell}$$
5. $$V_A \leftarrow V_{5\ell+6\ell}; V_B \leftarrow V_{12\ell}$$
6. $$V_A \leftarrow V_{11\ell+12\ell}$$

and we are done.

The number M will be at each step a factorial until we reach a bound. Once this bound is reached, and using the property previously presented, we try computing $\gcd(n, V_M - 2)$ in order to determine a non-trivial factorization of n. For a proper explanation of the whole procedure, we again need to go into too much detail for the scope of this thesis, so the interested reader might refer to [67]. This algorithm will be successful if the representation of p, a factor of n is as follows:

$$p = \prod_{i=1}^{r} q_i^{\alpha_i} - 1, \text{ with } q_i^{\alpha_i} \leq B_1 \text{ for all } i.$$

(5) Lenstra ECM

This method, discovered by H. W. Lenstra [41], is the most powerful algorithm for big numbers that we will discuss in this section. It is considered to be weaker than the Quadratic Sieve for RSA numbers, but is still used to factor some gigantic numbers compared with what the previous algorithms could do.
Recall that Pollard's $p - 1$ method calculates a^{m-1} where m is the least common multiple of a set of numbers under a bound B.
Lenstra will do the same, but instead of using the multiplicative group of integers modulo p, he makes use of an elliptic curve group $E_{a,b}$ modulo p, with operation of elliptic curve addition. A main advantage of this is that

there exist many elliptic curves $E_{a,b}$ modulo p, and therefore if we arrive to a point where we are stuck, we can just choose a different elliptic curve and start over. We will see this step in the algorithm. Since the paper is directed to students with knowledge in mathematics, we will avoid going through the basics of elliptic curves. For further information [30] and [34] can be followed. Nevertheless, we will recall the definition of elliptic curve addition and elliptic curve doubling, since it plays an important role in this algorithm. Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two points on an elliptic curve $E : y^2 = x^3 + ax + b$ with $x_P \neq \pm x_Q$.

Definition 7.7.1. *Then the **addition**, $P + Q = R$ is defined as follows:*

$$\text{Let } s = (y_P - y_Q)/(x_P - x_Q), \text{ then we have}$$

$$x_R = s^2 - x_P - x_Q \text{ and } y_R = -y_P + s(x_P - x_R).$$

Definition 7.7.2. *Let P be as before, and let $y_P \neq 0$, then the **doubling** of P, $2P = R$, is defined by:*

$$\text{Let } s = (3x_P^2 + a)/(2y_P), \text{ then we have}$$

$$x_R = s^2 - 2x_P \text{ and } y_R = -y_P + s(x_P - x_R).$$

The trick in this algorithm is to define an elliptic curve over the ring $\mathbb{Z}/n\mathbb{Z}$, where n is the number to factor. However, as it is well known, a curve is well defined over a finite field. Therefore, in this case we will not have a properly defined curve (since n is a composite number). This group-law failure will turn out to be the main tool of the factorization algorithm. In this case, some elements are not invertible, due to the fact that $\gcd(n, v) \neq 1, n$, where v is the element of which the inverse is computed. Here we have arrived to a good output of the algorithm.
Let us go step by step through the algorithm:

The elliptic curve (or pseudo-curve, since it is not well defined) will be $E_{a,b}(\mathbb{Z}_n) = \{(x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n : y^2 = x^3 + ax + b\} \cup \{0\}$, where 0 is the point at infinity.

Algorithm 7.8. As input, number n to factor.

1. We begin by choosing B_1, for instance $B_1 = 10000$
2. Next step is to define a curve $E_{a,b}(\mathbb{Z}_n)$ and a non-trivial point $(x, y) \in E$. An easy and effective way to do this is to randomly pick $x, y, a \in \{0, \ldots, n-1\}$ and then compute $b = (y^2 - x^3 - ax) \mod n$. At this point we compute $g = \gcd(4a^3 + 27b^2, n)$. If g is equal to n, then we have to redefine the curve and point. This is because we want a non-singular elliptic curve. If $1 < g < n$ then we have found our factor. Else we keep the curve E and the point P.

3. for i in range$(1, B_1)$:

   Find largest integer a_i such that $p_i^{a_i} \leq B1$

   for j in range$(1, a_i)$

   $P = p_i P$

   at each step, at each step, when performing addition, we calculate $\gcd(d, n)$, for d the denominator in the addition formula.

   If non-trivial, return g

4. Else, if the end of the loops come out without a successful factor, we reset the point and curve, and maybe the bound as well.

For the step in the second loop, we calculate the new values of P using the addition and doubling formulas that we explained earlier. It is then when we have to calculate inverse elements to find s. Here is where we will calculate the $\gcd(d, n)$, and if it is not 1 or n, then we exit the loop successfully. This is due to the fact of working with a non well defined elliptic curve.
Note, however, that this is a simplified explanation of the method. In practice ECM is implemented in a different manner, where projective coordinates are used, and we only check once at the end if the computation gave 0 in $\mathbb{F}_p$ by computing only one gcd.

As explained in [25], the heuristic complexity of ECM is $L(p)^{\sqrt{2}+o(1)}$, where p is the smallest prime factor. Therefore, the worst case scenario for this algorithm is when both factors of n have roughly the same size. If we know that the number to be factored is of this type, then it is recommended to use QS or GNFS. On the other hand, if we have a random number which is way too big to consider using the QS or GNFS, then it is recommended to start with ECM and see if we are lucky. This is one interesting feature of ECM, that the number of steps may vary largely due to the fact that we are only expecting one successful event to occur [25].

(6) SIQS and MPQS

These two algorithms are modifications of the Quadratic Sieve algorithm. The multi-polynomial Quadratic Sieve (MPQS) tries to surpass one of the main problems of the QS: the sieving region. As we need to have more smooth values than the size of the factor base, we have to sieve through a huge interval. A solution to this problem is to use multiple polynomials to generate values, and sieve each polynomial over a smaller interval. For more information about this algorithm, please refer to [56]. For the self-initializing QS (SIQS), the idea remains the same of using different polynomials, but in

their paper they offer a way to calculate them in an efficient way, and their claim is that with such a method, the sieving area can be even smaller than in the MPQS, and therefore results in a faster algorithm [23].

7.9 The New GeneralFactorizer

The idea how to broaden CT2 was to add the GNFS for factorization as an additional feature. For this, some research in the already implemented versions of the GNFS was done, to analyze which was the best choice for the Windows application. There were many available choices, from which I explain the most promising three.

The first one was motivated by the latest record factorizations, CADO-NFS [60], and was the favorite for this task. However, it has been developed only for Unix systems, and our interest (usage in Windows) is only partly supported. During my internship at the headquarters of Deutsche Bank, we tried to compile CADO-NFS for Windows, but after spending a long period for this task and not arriving to a solution, we then decided to leave aside this option.

Then there was GGNFS [39] together with msieve [46], but for usage in Windows it needed a Python middle layer. This was to be avoided for CrypTool 2 in order to bypass the requirement of a Python dependency for the application.

Finally there was YAFU (Yet Another Factorization Utility) [18], which seemed pretty simple to add to the .NET application and which offered quite good functionality of, not only GNFS, but also other factoring algorithms. YAFU makes usage of the msieve library together with GGNFS binaries (without a Python middle layer), so the decision to re-write the Python middle layer as a C# code manually was also left aside, and the final decision of using YAFU was taken.

7.9.1 YAFU (Yet Another Factorization Utility)

YAFU, available for download at [18], offers the following list of 9 factoring algorithms:

- Self Initializing Quadratic Sieve (SIQS)
- Multiple Polynomial Quadratic Sieve (MPQS)
- Elliptic Curve Method (ECM)
- Pollard's $p-1$ method $(p-1)$
- Williams' $p+1$ method $(p+1)$
- Shanks' square forms factorization (squfof)

- Pollard's rho method (ρ-method)
- Fermat's algorithm
- Number Field Sieve (both GNFS and SNFS)

Together with this, there is an implementation which uses several factoring algorithms instead of sticking to a single one for factorization which is very useful as we will see in the presentation of the results. It is to mention that the GNFS algorithm implementation in YAFU is mainly a usage of other public domain software such as msieve or the lattice sievers of Franke and Kleinjung [18].
YAFU also provides the user with a prime sieve, prime generations, logarithm calculations, greatest common divisor or RSA moduli generator among others. It has a very simple and straightforward usage, and it is mainly a command-line driven tool. This is one of the reasons it was the best choice in relation to the integration to CrypTool 2. We used the latest available version of YAFU, version 1.34 (released in 2013).

7.9.2 The Plug-in

The intention of this part of the work was to offer the users of CT2 a plug-in where the state-of-the-art in factoring could be used. The difference between the state-of-the-art-factorizations (as mentioned before, 512-bit keys in under four hours) and what is reached with the available plug-in are still huge, as we will see in the presentation of the results later on the chapter. Nevertheless, the algorithm in use remains the same, and therefore the user can already get a gasp of how these record factorizations are being attained. Not only we aim to offer the usage of the GNFS for the user, but also give a small explanation of what is happening. Following the line of all other plug-ins in the CT2 application, a documentation is available.

As we have explained before, CT2 is an e-learning platform, so we assume that not all the users will have deep understanding in mathematics, and hence, the documentation will be both pedagogical and user friendly.

As previously mentioned, CrypTool was in the first place created as a program to raise awareness. This plug-in not only wants to enable the user to factor huge numbers, but also to show what a single computer can do. It can be learned from the experience with the plug-in (to those interested in following the information given in the app) that 512-bit keys are not secure at all anymore, and if the user wants to remain really private, a 1024-bit key might be something to start leaving aside as well. It is to remember that the results given in the previous chapters are from research groups, which have much less funding and computer power than government agencies. Of course, we have no idea

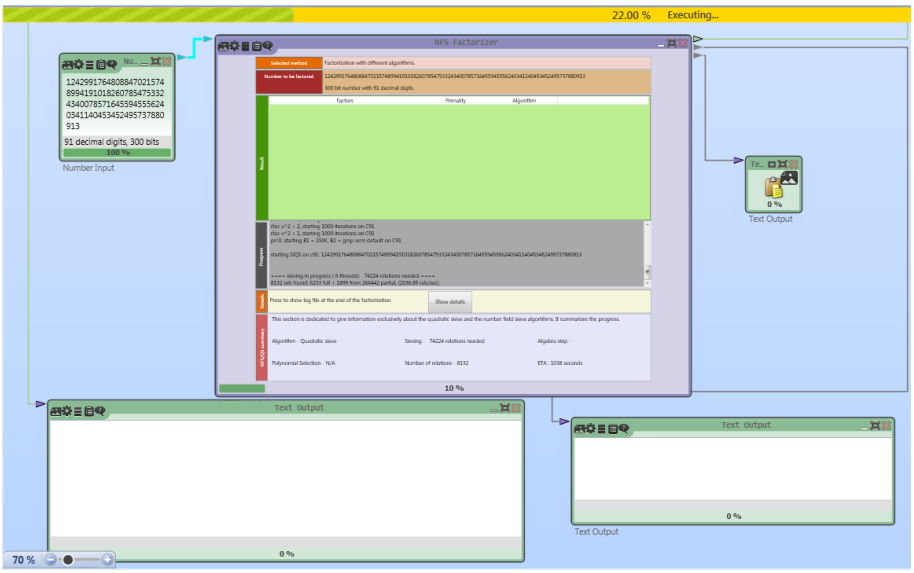

Figure 7.1: GUI of the plug-in

at what point they really are in the developing of computers and algorithms.

The user will have parameters to determine for the factorization to take place. These parameters will depend on the chosen algorithm, so for instance, if the chosen factorizer is Trial Division, the user will choose the bound of the primes with which to try the factorizations. If, on the other hand, the Quadratic Sieve is chosen, then the number of threads, or the maximum time for factorizing are some of the possible parameters to select. For the case of the general factorization (which is the choice where many algorithms are involved) the user will only decide whether to use ECM or not, and the number of threads involved. However, for each algorithm there is a meaningful default already chosen when the plug-in starts.

The plug-in offers information about the progress of the selected algorithms, and for the case of Quadratic Sieve and the GNFS, a summary is also presented. The reason for offering a summary only for these two algorithms is because they are the most complicated ones to follow. We want to present an easy to understand interface. One of the differences between CT1 and CT2 is, that CT2 offers a progress bar, representing how far the algorithm proceeded. However, it is quite complicated to create a totally accurate bar. First of all, this is due to the number of selections available for the user, but mainly because of the two most complex algorithms in the list: Quadratic Sieve and the GNFS. The progress of these two algorithms is not always incrementing. For instance, in the sieving step of both algorithms, we can-

not really know how long it will take in order to find all the relations needed.

The plug-in is intended for factorizations of big numbers. The downside of this is that in methods aimed for smaller numbers, such as Pollard rho or Fermat's factorization method, the factorization might not be complete, in the sense that the factors outputted might be composite. The plug-in will output these factors and will specify that they are composite. The user then has to take these numbers manually and place them back in the factorization window (as explained in 7.10.2, the automation of this process is meant as further work).

7.10 Results and Further Work

In this section we present the performance of the new plug-in. We compare running times of other factoring plug-ins of the CT1 and CT2 applications with the running times of the GeneralFactorizer. The results presented in this section are not meant to be compared with the study of the University of Pennsylvania [62] discussed in the previous chapter. The latter used much higher computational power, as for those results, they made use of the Amazon Elastic Compute Cloud (Amazon EC2). Amazon EC2 is a web service that provides computing capacity over the cloud. The University of Pennsylvania team made use of this using the largest type of compute-optimized instances available when their study happened. This instance type has two Intel Xeon E5-2666 v3 processor chips, with 36 vCPUs in a NUMA (Non-Uniform Memory Access) configuration with 60GB of RAM. On the other hand, the machine I run the tests on has one Intel Core i7-3630QM chip with one CPU at 2.40GHz, with 8 logical processors.

However, the CrypTool 2 team is developing a cloud service where a user can get computing power in the cloud by using distributed computing. This service is still in progress, so I could not make use of it as for now, but once this is finished the GeneralFactorizer could be used in coordination with this. This will augment the functionality of the plug-in, mainly because the GNFS is an easy to parallelize algorithm and the running times could be greatly improved.

7.10.1 Performance of the Plug-in

This subsection covers an analysis of the running times of the plug-in, however before presenting the running times, we begin by shortly describing other programs and plug-ins used: mainly the CT1 Factorizer [1], the QS Factorizer and the old NFS Factorizer in CT2 [2]. Finally we describe the settings used in the new GeneralFactorizer.

The *CT1 factorizer*, similar to the GeneralFactorizer, offers a choice of different algorithms to the user, such as trial factorization, Brent's method,

ECM or the Quadratic Sieve. The CT1 factorizer runs all chosen algorithms in parallel (however, the user can easily stop a thread later) – as opposed to the GeneralFactorizer that runs them one after the other. The parallelization of algorithms can be useful when the number is of unknown size, nonetheless, it might also slow down for certain cases, e.g. when memory is tight. Further in the subsection will follow a specific case of this type of situation.

The *QS Factorizer* also makes use of other algorithms different to the QS, such as trial division. However, it does not have the ECM method implemented which results in a poor performance for certain numbers.

The old *NFS Factorizer* factorizes uniquely with the NFS algorithm, which results in an error for small numbers, and unnecessarily large running times for numbers of a specific case.

Finally, the *GeneralFactorizer*, as we have previously seen, has different algorithms to choose from, but we will aim our study to the choice of the General Factorization, as shown in Figure 7.2. This choice makes use of

Figure 7.2: Drop down box of available choices

several algorithms, but sequentially in this case. There are also some settings available to the user, such as the number of threads, whether we want the process to be in idle priority or not, or the factoring plan[1]. The last setting allows the user to choose whether the factorizer should avoid running ECM or do a light, medium or deep search with ECM as shown in Figure 7.3. For our results, due to the knowledge of the type of numbers, we chose the plan to avoid running ECM unless specified otherwise.

The GeneralFactorizer plug-in is the first stable plug-in in CrypTool 2 being able to factor arbitrary numbers with the General Number Field Sieve. Moreover, it outperforms most of the other plug-ins accessible for

[1]This last choice is available only for the General Factorization method, which offers to the user the choice of how deep the search with ECM should be.

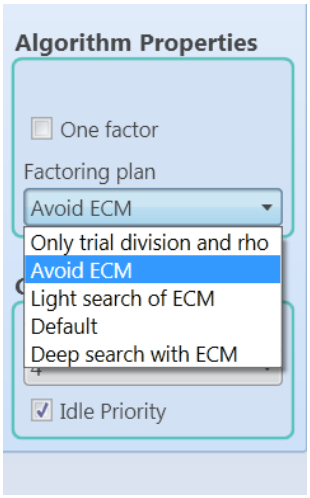

Figure 7.3: Drop down box for determining search of ECM

factorization, and has a similar running time as the NFS Factorizer for big numbers[2]. We will now present these results in the form of screen shots and charts. We begin by comparing the performance with 300-bit numbers which are products of two primes of similar size. Figures 7.4 and 7.5 show the performance of the GeneralFactorizer and the QS factorizer respectively[3]. We can see that the performance of the GeneralFactorizer plug-in is faster by a third of the running time of the QS plug-in. For smaller numbers, such as 128837284670563056926945319649507306384894847834165908817610888209826043588 3 (250 bit number) the GeneralFactorizer has a slightly better performance compared to the QS factorizer, with running times of 32 and 48 seconds respectively.

We now proceed with a more general analysis of the performances of CT1 Factorizer, CT2 QS, CT2 NFS and the CT2 GeneralFactorizer. In order to perform these tests we used two different sets of numbers. One of them consists in hard to factor numbers, which are numbers that have two factors of similar size (numbers used as RSA modulus). These numbers will range from 200-bit till 370-bit numbers, shown in Table A.1. The second set consists of two special numbers whose factors are chosen in a way that the Elliptic Curve Method is the most suitable algorithm to factor (i.e. factors of different size, with one significantly smaller than the other). The results of the first kind of numbers are presented in Figure 7.6 with the running

[2]Both plug-ins use the same NFS implementation, mainly the msieve library with the ggnfs binaries.

[3]We do not include a screen shot of the NFS plug-in due to the lack of a timer in the plug-in, which means we had to measure the time manually.

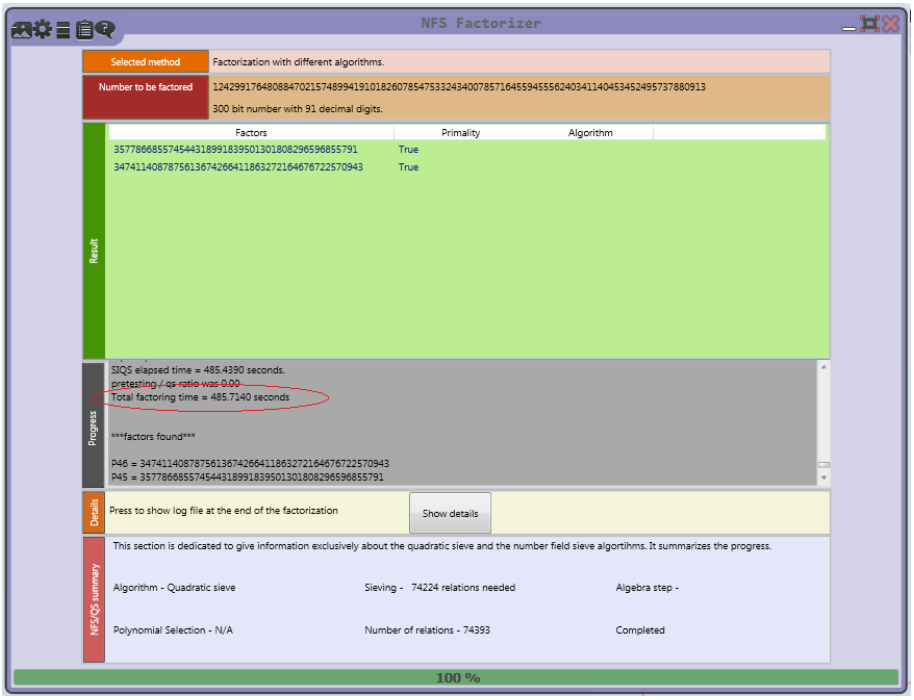

Figure 7.4: Performance of the GeneralFactorizer for a 300-bit number, lasting slightly above 8 minutes

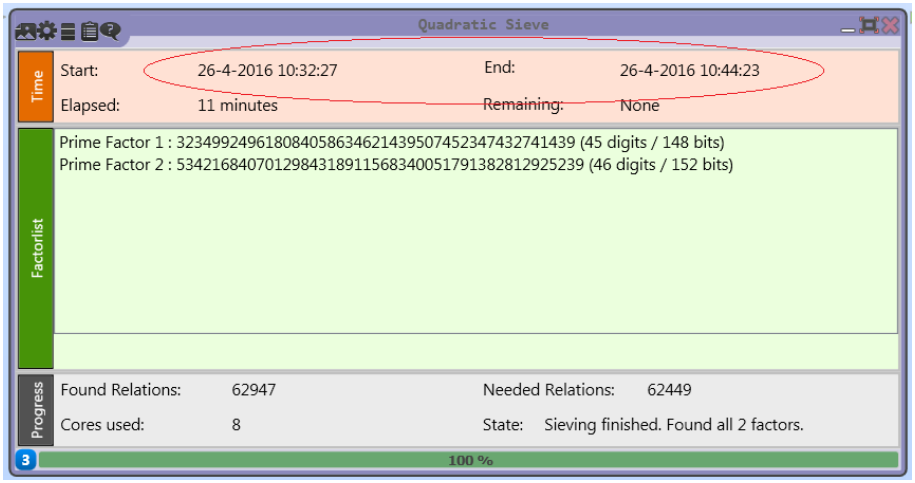

Figure 7.5: Performance of the old Quadratic Sieve plug-in for a 300-bit number, lasting slightly below 12 minutes

times in the y-axis and the bit length in the x-axis, while the results for the

second group are presented in Figure 7.8.

One notices that performances of the QS plug-in and the NFS plug-in

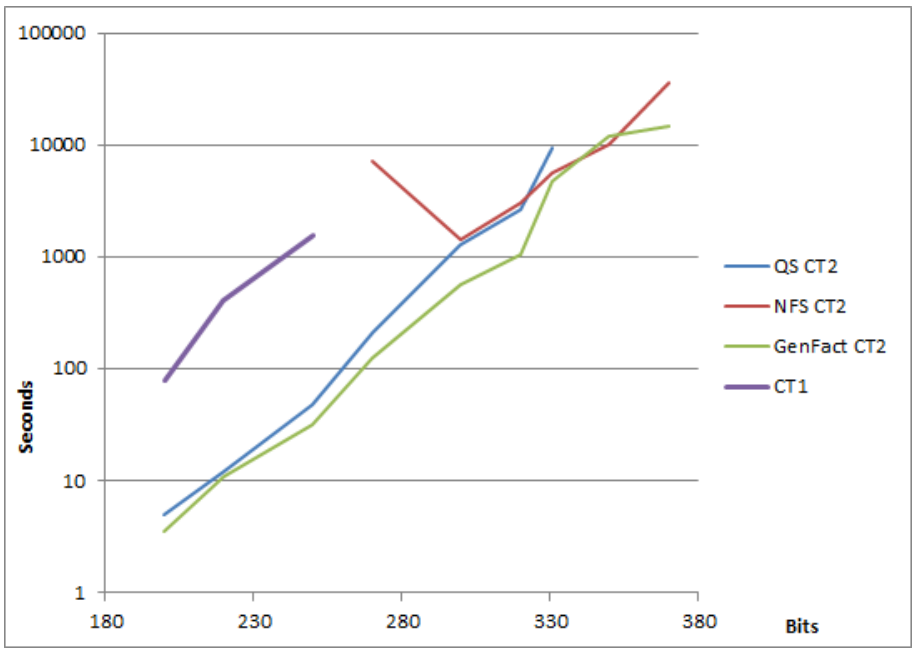

Figure 7.6: Comparison of performance with time in logarithmic scale. Numbers can be found in Table A.1

depend on the size of the number. For numbers of 320-bits or smaller, the QS is a far better choice[4], while with numbers bigger than 320-bit, the NFS is more performant. For smaller numbers the NFS CT2 was taking way too long, and it was decided not to run tests.

The GeneralFactorizer not only performs in a better manner in most of the cases, but it also makes the choice of which algorithm to choose automatically (depending on the size of the number to factor) and eases the task to the user.

It is clear that the bigger the number is, the bigger the gap of the running time becomes compared with the QS. In the case of the 331-bit number the QS CT2 runs twice as slow. The difference between the GeneralFactorizer and the NFS CT2, for the case of 331-bit, is of 900 seconds. While in the case of 350-bit, the NFS CT2 ran on average 2000 seconds faster. In the case of 370-bit, the NFS CT2 takes much longer than expected. To be precise, the running time in the chart is an approximation due to slow performance. After 5 hours, the factorization was only at 50% of the sieving

[4]For the case of 270-bit, the NFS factorizer was taking around 2 hours, slower than its own performance for 331-bit numbers.

stage.

For the case of the CT1 algorithm, these tests where too slow due to its implementation of the Quadratic Sieve algorithm and for numbers bigger than 250, the algorithm was taking too long. However, we will see that it does not perform as bad with the second group of numbers.

In the second group of numbers we are able to understand the advantage of the different available algorithms. We will also see how higher degrees of parallelization do not guarantee better performance. For instance, in the second number of the group studied in Figure 7.8, CT1 Factorizer tried factorization with all methods, and therefore slowing down the ECM iterations. If we only choose the ECM, the plug-in improves by 13% its running time. The alternative, which is the one used in the GeneralFactorizer, is to run the algorithms linearly. The downside of this is

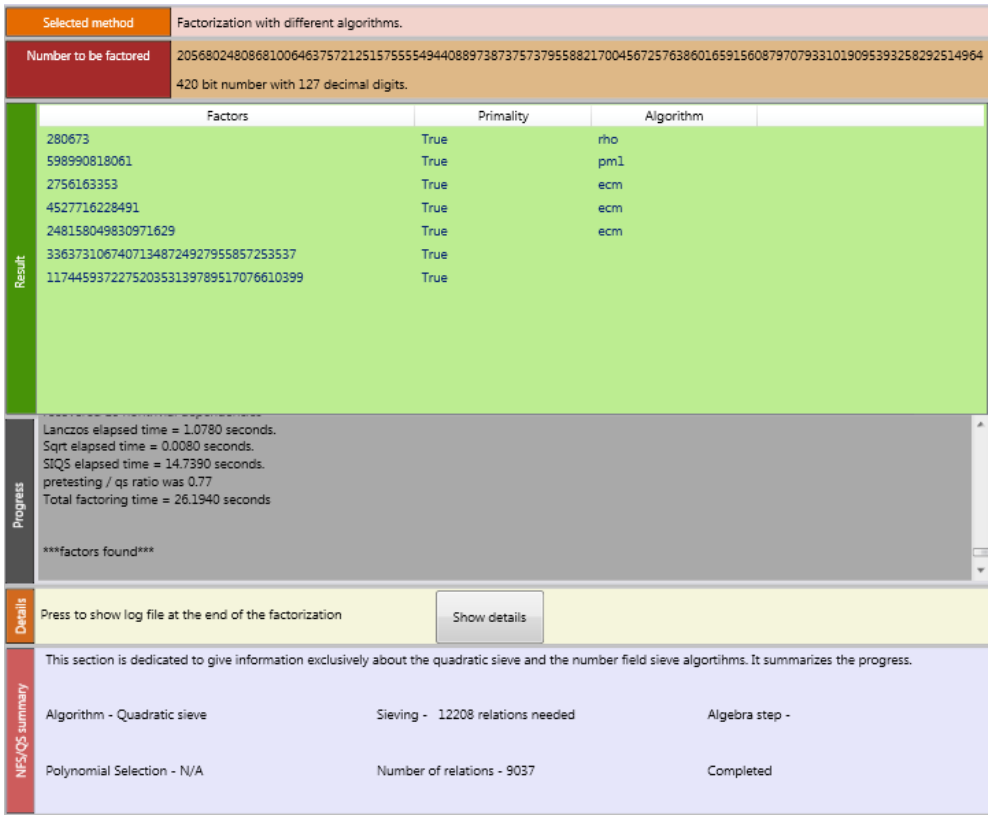

Figure 7.7: Performance of the GeneralFactorizer with number 1 from Table A.2

if, for instance, we want to factor an RSA modulus without the knowledge that the number is of that type, results in running ECM. The ECM will

not find the factors, and therefore it will waste time. As an example, the factorization presented above of a 250-bit number is delayed to a total of 125% its initial running time.
In Figure 7.8 we can see the performance of CT1 Factorizer (CT1), QS CT2 and GeneralFactorizer CT2 (GenFact CT2) with two special numbers where Number 1 is a special 420-bit number and number 2 is a 287-bit number, shown in Table A.2, with factors of different size and therefore suitable for ECM. We can see how the QS factorizer performs terribly, which is due to the fact that after finding some of the small factors with methods as Pollard's ρ or $p-1$, it proceeds directly with the QS for a number of 100 digits. The CT1 Factorizer, even if much slower than the GeneralFactorizer, performs better than the CT2 QS because it also has Lenstra's ECM algorithm implemented in it. In this specific set of numbers we see how the GeneralFactorizer runs much faster than the available algorithms, by taking less than a second for number 2, and therefore not appearing in the logarithmic scale. The reason of this result is, as we explained before, that ECM depends strongly on the size of the smallest factor of a number, while the QS is independent of that and is only dependent on n.

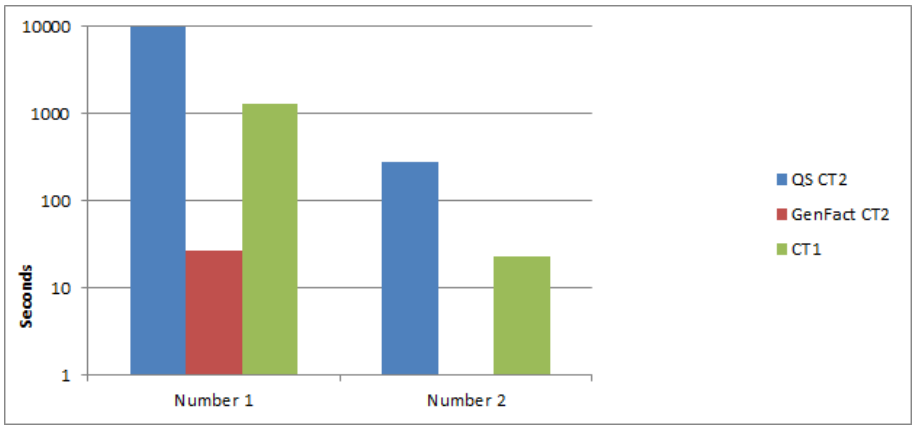

Figure 7.8: Comparison of performance with time in logarithmic scale. Numbers can be found in Table A.2

These results show that the two old plug-ins in CT2 can be replaced with the GeneralFactorizer without any disadvantage.

The plug-in is now functional and factorizations work with the General Number Field Sieve and the 10 other factorization algorithms.

In the following section we will see a few points of further work that can be added to the plug-in. So, despite the fact that we have achieved our purpose of offering a stable plug-in, some points can still be improved.

7.10.2 Further Work for the GeneralFactorizer

Here are some comments regarding further work:

- Algorithms such as Pollard rho, Fermat's method, $p-1$, etc. output always a pair of factors, due to the fact that these algorithms find factors with the difference of squares, $x^2 \equiv y^2 \pmod{n}$. An idea for further work is to automatically continue factoring these numbers if they are composite or to offer the user to factor these without needing to copy and paste them in the input window.
  The reason why this is not implemented is because the main intention of this plug-in in the first place was to offer a plug-in able to factor big RSA numbers, mainly with the GNFS so that the first split gives the full factorization.

- The progress bar is not totally accurate (it will be hard to manage it, due to the complixity of the GNFS) and some modifications might be useful.

- The development of the CT2 CrypCloud is still under progress, but once this is managed, it would be interesting to port the GeneralFactorization plug-in into the CrypCloud. For this it is important to know that YAFU is already implemented to distribute the process among different cores. However this is only performed in the two 'easy' steps to parallelize, mainly the polynomial selection and the sieving, while the parallelization of the linear algebra is not optimal. A good paper explaining how to distribute this process in an effective way is the one of the group of the University of Pennsylvania [62].

- It would also be interesting to check the progress of CADO-NFS and see whether they develop the program for Windows operating systems, and in this case, if there is interest in using the state-of-the-art in factoring, trying to include it. Furthermore, CADO-NFS is updated more often than YAFU (the last update of the latter was in 2013), and therefore is more likely to have the latest advances in the GNFS factorization from within CADO-NFS.

- A final change that could be done to the plug-in would be to add references to this paper once it is online. We could have four main links redirecting to different parts of this paper. First would be *The theory behind the GNFS* redirecting the user to the second chapter (to follow until the end of chapter four). Second link might refer to *The four steps of the GNFS* taking the user to chapter five where explanations are given about the four main steps of the algorithm. Third important section would be *How much is the GNFS a danger for our encryption* which would redirect the user to chapter 6, and

finally a fourth option giving *More details of the plug-in* sending the user to this chapter, chapter 7.

Appendices

Appendix A

Numbers used for performance tests

Some tests were made with more than one number, so the numbers in A.1 are mainly representative. All of the numbers in Table A.1 (as well as the ones used in the performance tests) were generated by YAFU, using the command *"rsa(s)"*, with s the number of bits.

Table A.2 give the numbers with special form to make ECM the most suitable algorithm to use, taken from the README file of YAFU.

Bit Length	Number
200	1420795552156657914899236212440230170883564633098606022036373
220	1577918532112654333223216834840589517321825004569168686462331286249
250	1197143477033289400345490340603978981510549252806031826867156726588301839393
270	1459420954682274614333956623722781584306385064034111509349056056233440002496102477
300	1775493011499636357853216095985966832494033326220981414659996746683023630554396549621114303
320	1499844014444836535174389624822909656678940119918958298898646206230765153756621816927493546374371
350	1313251941000877313402626275192108171439776040631775295017426387595463765266683262103645164214644845373903
370	2365858252265473531686978101079954296393440737593987380165384257507818645474001192704343841118011909692019728541

Table A.1: Hard to factor numbers

Bit Length	Number
287 (#2)	1408702985503599249147041607374199052577475448668926320000628964769686025784829663427 04
400 (#1)	2056802480868100646375721251575555494408897387375737955882170045672576386016591560879707933101909539325829251496440620798637813

Table A.2: Special numbers

Bibliography

[1] *CrypTool 1*. https://www.cryptool.org/en/cryptool1.

[2] *CrypTool 2*. https://www.cryptool.org/en/cryptool2.

[3] *Elliptic Curve Diffie-Hellman*. https://en.wikipedia.org/wiki/Elliptic_curve_Diffie-Hellman.

[4] *Help forum for Telsacrypt Ransomware*. https://community.spiceworks.com/how_to/125475-teslacrypt-2-2-0-removal-and-decryption.

[5] *Help forum for Telsacrypt Ransomware*. http://mersenneforum.org/showthread.php?t=20779.

[6] *LogJam attack*. https://www.mitls.org/pages/attacks/Logjam.

[7] *Post-quantum cryptography*. https://en.wikipedia.org/wiki/Post-quantum_cryptography.

[8] *Primality testing*. https://en.wikipedia.org/wiki/Primality_test.

[9] *RSA Challenge Numbers*. http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm.

[10] *Sieve of Eratosthenes*. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

[11] *United States Cryptography Export/Import Laws*. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/united-states-cryptography-export-import.htm.

[12] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. ACM.

[13] E. B. Barker, W. C. Barker, W. E. Burr, W. T. Polk, and M. E. Smid. *SP 800-57. Recommendation for Key Management, Part 1: General (Revised).* Technical report, Gaithersburg, MD, United States, 2007, National Institute of Standards & Technology.

[14] E. R. Berlekamp. *Factoring Polynomials over Large Finite Fields.* In *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, SYMSAC '71, pages 223–, New York, NY, USA, 1971. ACM.

[15] D. J. Bernstein and A. K. Lenstra. *A general number field sieve implementation*, pages 103–126 in [40]. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[16] M. E. Briggs. *An Introduction to the General Number Field Sieve, Master thesis.* `https://www.math.vt.edu/people/brown/doc/briggs_gnfs_thesis.pdf`, 1998.

[17] J. P. Buhler, H. W. Lenstra, and C. Pomerance. *Factoring integers with the number field sieve*, pages 50–94 of [40]. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[18] B. Buhrow. *Yet Another Factoring Utility (YAFU).* `https://sites.google.com/site/bbuhrow/`.

[19] P. J. Cameron. *Projective and Polar Spaces.* Queen Mary College Dept. of Mathematics, 2nd edition, 2000.

[20] D. G. Cantor and H. Zassenhaus. *A New Algorithm for Factoring Polynomials Over Finite Fields.* Mathematics of Computation, 36(154):587–592, 1981.

[21] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. T. Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. *Factorization of a 512-bit RSA Modulus.* In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 1–18, Berlin, Heidelberg, 2000. Springer-Verlag.

[22] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics.* Springer-Verlag, Berlin, 1993.

[23] S. P. Contini. *Factoring integers with the self-initializing quadratic sieve.* University of Georgia, `http://www.crypto-world.com/documents/contini_siqs.pdf`, 1997.

[24] J.-M. Couveignes. *Computing a square root for the number field sieve*, pages 95–102 of [40]. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[25] R. Crandall, C. Pomerance, R. Crandall, and C. Pomerance. *Prime numbers: a computational perspective. Second Edition*, 2005, Springer-Verlag New York.

[26] J. Daemen and V. Rijmen. *AES Proposal: Rijndael,1999*, 1999.

[27] J. D. Dixon. *Asymptotically fast factorization of integers.* Mathematics of Computation, 36:255–260, 1981.

[28] J. Gilchrist. *factMsieve, http://gilchrist.ca/jeff/factoring/nfs_beginners_guide_perl.html*.

[29] Googulator. *TeslaCrack.* `https://github.com/Googulator/TeslaCrack`.

[30] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[31] N. Heninger and J. A. Halderman. *The legacy of export-grade cryptography in the 21st century.* `http://summerschool-croatia.cs.ru.nl/2016/slides/NadiaHeninger.pdf`, 2016.

[32] A. Hülsing. *Quantum Computing vs. Your Privacy.* `https://huelsing.files.wordpress.com/2013/04/20160526_pqcryptoforprivacy.pdf`.

[33] T. Hungerford. *Algebra*, volume 73 of *Graduate Texts in Mathematics.* Springer New York, 2003.

[34] D. Husemöller. *Elliptic curves*, volume 111 of *Graduate texts in mathematics.* Springer, New York, 2004.

[35] F. Jarvis. *Algebraic Number Theory.* Springer Undergraduate Mathematics Series. Springer International Publishing, 2014.

[36] T. Kleinjung. *On polynomial selection for the general number field sieve.* Mathematics of Computation, 75(256):2037–2047, 2006.

[37] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. *Factorization of a 768-bit RSA Modulus.* In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, volume 6223, Lecture Notes in Computer Science of *CRYPTO'10*, pages 333–350, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] T. Kleinjung, J. W. Bos, and A. K. Lenstra. *Mersenne Factorization Factory.* In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, pages 358–377, 2014.

[39] A. Korobeynikov. *ggnfs,* `https://sourceforge.net/projects/ggnfs/`.

[40] A. K. Lenstra and J. Hendrik W. Lenstra, editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics.* Springer-Verlag, Berlin, 1993.

[41] H. W. Lenstra. *Factoring integers with elliptic curves.* Annals of Mathematics, Second Series, Vol. 126, No. 3, pp. 649-673.

[42] R. A. Mollin. *Algebraic Number Theory, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2011.

[43] P. L. Montgomery. *Square roots of products of algebraic numbers.* [63], pages 567–571.

[44] P. L. Montgomery. *A Block Lanczos Algorithm for Finding Dependencies over GF(2).* In *Advances in Cryptology — EUROCRYPT '95: International Conference on the Theory and Application of Cryptographic Techniques Saint-Malo, France, May 21–25, 1995 Proceedings*, EUROCRYPT '95, 1995.

[45] B. A. Murphy. *Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm.* PhD thesis, Autralian National University, `https://maths-people.anu.edu.au/~brent/pd/Murphy-thesis.pdf`.

[46] J. Papadopoulos. *msieve.* `https://sourceforge.net/projects/msieve/`.

[47] J. M. Pollard. *The lattice sieve*, pages 43–49 in [40]. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[48] C. Pomerance. *Smooth numbers and the Quadratic Sieve.* Algorithmic Number Theory, MSRI Publications, 2008.

[49] C. Pomerance. *The number field sieve.* pages 465–480, in Proceedings of Symposia in Applied Mathematics, Volume 48, 1994.

[50] T. Prest and P. Zimmermann. *Non-linear polynomial selection for the number field sieve.* J. Symb. Comput., 47(4):401–409, 2012.

[51] N. Rezola. *Unique Prime Factorization of Ideals in the Ring of Algebraic Integers of an Imaginary Quadratic Number Field*, 2015, Master thesis, California State University, `http://scholarworks.lib.csusb.edu/cgi/viewcontent.cgi?article=1223&context=etd`.

[52] R. Rivest, A. Shamir, and L. Adleman. *On Digital Signatures and Public-Key Cryptosystems.* Massachusetts Inst Of Tech Cambridge Lab for Computer Science, 1977, Technical Memo.

[53] R. L. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-key Cryptosystems.* Commun. ACM, 21(2):120–126, Feb. 1978.

[54] D. Shanks. *Analysis and Improvement of the Continued Fraction Method of Factorization.* `https://www.usna.edu/Users/math/wdj/_files/documents/mcmath/shanks_analysis.pdf`.

[55] P. W. Shor. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.* SIAM J. Comput., 26(5):1484–1509, Oct. 1997.

[56] R. D. Silverman. *The Multiple Polynomial Quadratic Sieve.* Mathematics of Computation, 48(177):329–339, 1987.

[57] P. Stevenhagen. *The Number Field Sieve*, pages 83–100. Mathematical Sciences Research Institute Publications, 2008.

[58] I. Stewart and D. Tall. *Algebraic Number Theory and Fermat's Last Theorem: Third Edition.* AK Peters Series. Taylor & Francis, 2001.

[59] R. Thangadurai. *Irreducibility of Polynomials Whose Coefficients are Integers.* Mathematics Newsletter, 17:29–61, 2007.

[60] The CADO NFS Development Team. *CADO-NFS, An Implementation of the Number Field Sieve Algorithm*, 2015. Release 2.2.0.

[61] E. Thomé. *Square root algorithms for the number field sieve.* In F. Özbudak and F. Rodríguez-Henríquez, editors, *4th International Workshop on Arithmetic in Finite Fields - WAIFI 2012, Lecture Notes in Computer Science*, volume 7369, pages 208–224, Bochum, Germany, July 2012. Springer.

[62] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger. *Factoring as a Service.* Financial Cryptography and Data Security, 2015. `http://eprint.iacr.org/2015/1000`.

[63] B. Vancouver. *Mathematics of computation 1943-1993: A half-century of computational mathematics, January 1995.* American Mathematical Soc.

[64] S. S. Wagstaff. *The Joy of Factoring*. Student mathematical library. American Mathematical Society, 2013.

[65] S. H. Weintraub. *Factorization: Unique and Otherwise*. AK Peters Series. Taylor & Francis, 2008.

[66] D. H. Wiedemann. *Solving sparse linear equations over finite fields*. IEEE Transactions on Information Theory, 32:54 – 62, 1986.

[67] H. C. Williams. *A $p + 1$ method of factoring*. Mathematics of Computation, Vol. 39:pp. 225–234, Jul., 1982.

[68] R. Williams. *Cubic Polynomials in the Number Field Sieve*. MSc Thesis, Texas Technical University, `http://www.math.ttu.edu/~cmonico/research/Williams_Ronnie_Thesis.pdf`.

[69] M. Yang, Q. Meng, Z. yi Wang, L. Wang, and H. Zhang. *Polynomial selection for the number field sieve in geometric view*. IACR Cryptology ePrint Archive, 2013:583, 2013.