Integer Factorization

Per Leslie Jensen <pleslie@diku.dk>

Master Thesis

Department of Computer Science
University of Copenhagen
Fall 2005

This document is typeset using LaTeX 2_ε.

Abstract

Many public key cryptosystems depend on the difficulty of factoring large integers.

This thesis serves as a source for the history and development of integer factorization algorithms through time from trial division to the number field sieve. It is the first description of the number field sieve from an algorithmic point of view making it available to computer scientists for implementation. I have implemented the general number field sieve from this description and it is made publicly available from the Internet.

This means that a reference implementation is made available for future developers which also can be used as a framework where some of the sub algorithms can be replaced with other implementations.

Contents

CHAPTER 1

Preface

"Unfortunately what is little recognized is that the most worthwhile scientific books are those in which the author clearly indicates what he does not know; for an author most hurts his readers by concealing difficulties."

- Evariste Galois(1811-1832)

In 1801 *Gauss* identified *primality testing* and *integer factorization* as the two most fundamental problems in his *"Disquisitiones Arithmeticae"*[29] and they have been the prime subject of many mathematicians work ever since and with the introduction of public key cryptography in the late 1970's it is more important than ever.

This thesis is a guide to integer factorization and especially the fastest general purpose algorithm known: *the number field sieve*. I will describe the number field sieve algorithm so it can be implemented from this description. To convince the reader I have implemented the algorithm from my description and have made the implementation public available on the Internet[1] so it can be used as a reference and maybe be improved further by anyone interested.

The target reader is a computer scientist interested in public key cryptography and/or integer factorization. This thesis is not a complete mathematical breakdown of the problems and methods involved.

Even before Gauss, the problem of factoring integers and verifying primes were a hot topic, but it seemed intractable and no one could find a feasible way to solve these problems and to this day it still takes a large amount of computational power.

Till this day the problem of factoring integers is still not solved, which means there is no deterministic polynomial time algorithm for factoring integers. The primality testing problem has just recently been solved by Agrawal, Kayal and Saxena[4].

Integer factorization has since the introduction of public key cryptosystems in 1977 been even more important because many cryptosystems rely on the difficulty of factoring integers (large integers). This means that a very

[1] Implementation available from http://www.pgnfs.org

fast factoring algorithm would make for example our home banking and encrypted communication over the Internet insecure.

The fastest known algorithm for the type of integers which cryptosystems rely upon is the *general number field sieve*, it is an extremely complex algorithm and to understand why and how it works one needs to be very knowledgeable of the mathematical subjects of algebra and number theory.

This means that the algorithm is not accessible to a large audience and to this point is has only been adopted by a small group of people who have been involved in the development of the algorithm, that is: until now...

This thesis has the first algorithmic description of all the steps of the number field sieve and it has resulted in a working implementation of the algorithm made publicly available.

The reader should know that it is a large task to implement all the steps from scratch, and here you should benefit from my work by having a reference implementation and maybe start by trying to implement a part of the algorithm and put it into my framework. My implementation is probably not as fast and optimal as it could be but it is readable and modular so it is not to complex to interchange some of the steps with other implementations.

The implementation here has been developed over several months but it helped the understanding of the algorithm and have given me the knowledge to write the algorithms in a way so a computer scientist can implement it without being a math wizard.

The thesis consists of the following Chapters:

Chapter 2 gives the fundamental algebra for the rest of the paper, this Chapter can be skipped if you know your algebra or it can be used as a dictionary for the algebraic theory used.

In Chapter 3 I will take a trip trough the history of integer factorization and pinpoint the periods where the development took drastic leaps, to understand the ideas behind todays algorithms you need to know the paths the ideas has travelled on.

The motivation for this paper is the link to cryptography and in Chapter 4 I will give an introduction to public-key cryptography and take a more detailed look on the RSA scheme, which is based on the difficulty of factoring integers and it is still the most used public-key cryptosystem today.

In Chapter 5 I will take a closer look at some of the algorithms mentioned in Chapter 3 and describe the algorithms leading to the number field sieve.

The main contribution in this paper is Chapter 6 and 7. In Chapter 6 I will describe the number field sieve algorithm step by step, and write it on algorithmic form usable for implementation. In Chapter 7 I will describe my implementation of the number field sieve, and give some pointers on implementation specific issues of the algorithm.

Acknowledgments

I would like to thank *Stig Skelboe* and *Jens Damgaard Andersen* for taking the chance and letting me write this paper.

I am grateful for valuable help from *Prof. Daniel J. Bernstein* with the square root step.

Thanks to my parents and friends for supporting me through the long period I have worked on this, and for understanding my strange work process.

Thanks to *Bue Petersen* for helping with the printing and binding of this thesis.

Special thanks to *Jacob Grue Simonsen* for valuable discussions on- and off-topic and for proofreading the paper.

And a very special thanks to *Katrine Hommelhoff Jensen* for keeping me company many a Wednesday and for keeping my thoughts from wandering to far from the work at hand.

Per Leslie Jensen
October 2005

CHAPTER 2

Mathematical Prerequisites

"God made the integers, all else is the work of man."

- Leopold Kronecker(1823-1891)

The main purpose of this chapter is to present the algebra and number theory used in this paper.

The theorems are presented without proofs. For proofs see [5] and [69].

The reader is expected to have some insight into the world of algebra, concepts like groups and fields are refreshed here, but the theory behind should be well known to the reader. For a complete understanding of the inner workings of the algebra, a somewhat deeper mathematical insight is needed.

If you have a mathematical background this chapter can be skipped.

2.1 The Theory of Numbers

In all branches of mathematics the building blocks are numbers and some basic operations on these. In mathematics the numbers are grouped into sets. The number sets used in algebra are, for the biggest part, the set of integers, denoted $\mathbb{Z}$, the set of rational numbers, denoted $\mathbb{Q}$ and the set of complex numbers, denoted $\mathbb{C}$.

Other sets of great interest includes the set of polynomials with coefficients in the three above mentioned sets.

We will now take a look at some of the properties of these sets.

2.1.1 The Set $\mathbb{Z}$

The set of integers $(\ldots, -2, -1, 0, 1, 2, \ldots)$, denoted $\mathbb{Z}$, has always been the subject for wide and thorough studies by many of the giants in the history of mathematics.

Many of the properties of $\mathbb{Z}$ are well known and used by people in everyday situations, without thinking about the underlying theory.

I will now take a look at the more intricate properties of $\mathbb{Z}$ that consitute the foundation for most of the theory used in this thesis.

Prime numbers are the building blocks of the set $\mathbb{Z}$, this is due to the unique factorization property which leads to *the fundamental theorem of arithmetic*.

Definition 2.1 **Greatest Common Divisor (GCD)**
Greatest Common Divisor of $a, b \in \mathbb{Z}$, denoted $\gcd(a, b)$, is the largest positive number, c, so that $c|a$ and $c|b$

Definition 2.2 **Prime number**
An integer $a > 1$ is called a prime number if $\forall\ b \in \mathbb{Z},\ b < a\ :\ \gcd(a, b) = 1$

Definition 2.3 **Relative prime**
Two integers a, b are said to be relatively prime if $\gcd(a, b) = 1$

Theorem 2.1 **Fundamental Theorem of Arithmetic**
Let $a \in \mathbb{Z}$ and $a > 1$ then a is a prime or can be expressed uniquely as a product of finitely many primes.

The next definition with the accompanying lemmas are the work of *Euler*, and they help us get to *Euler*'s theorem and *Fermat*'s little theorem.

Leonhard Euler (1707-1783) is without comparison the most productive mathematician ever. He worked in every field of mathematics and in the words of **Laplace**: *"Read Euler, Read Euler, he is the master of us all´´.* For a view of the man and his mathematics, see [24].

Definition 2.4 **The Euler phi function**
Let n be a positive integer. $\phi(n)$ equals the number of positive integers less than or equal to n, that are relative prime to n.

Lemma 2.1 If p is a positive prime number, then $\phi(p) = p - 1$

Lemma 2.2 For p and q distinct primes, $\phi(pq) = (p - 1)(q - 1)$

Lemma 2.3 If p is a prime and k is a positive integer, then $\phi(p^k) = p^k - p^{k-1}$

Pierre de Fermat (1601-1665) is considered to be the biggest "hobby" mathematicians of all times, he was a lawyer of education but used his spare time on mathematics. *Fermat's last theorem* is one of the most famous theorems ever, it remained unproven for more than 300 years. The story of *Fermat's last theorem* is told beautifully in [70].

Lemma 2.4 If $n > 1$ has prime factorization $n = p_1^{k_1} \cdot p_2^{k_2} \cdots p_r^{k_r}$, then $\phi(n) = (p_1^{k_1} - p_1^{k_1-1})(p_2^{k_2} - p_2^{k_2-1}) \cdots (p_r^{k_r} - p_r^{k_r-1})$

Lemma 2.5 a is an integer and $p \neq q$ are primes and $\gcd(a, p) = \gcd(a, q) = 1$ then $a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$

Theorem 2.2 B**-smooth**
If n is a positive integer and all its prime factors are smaller than B then n is called B*-smooth*.

By now we have the theory behind *Euler*'s Theorem and *Fermat*'s little theorem, and these can now be stated.

Theorem 2.3 ***Euler*'s Theorem**
If n is a positive integer with $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$

Theorem 2.4 ***Fermat*'s little theorem**
If p is a prime number and a is a integer and $p \nmid a$, then $a^{p-1} \equiv 1 \pmod{p}$

2.1.2 Polynomials in $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{C}$

In the previous section we saw some of the properties of the set $\mathbb{Z}$, especially the unique factorization property and the theory of irreducibility and prime numbers.

Now we turn our attention to polynomials, recall that a polynomial is an expression of the form $a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$ for some $m \in \mathbb{Z}$ and the coefficients $a_i \in \mathbb{Z}, \mathbb{Q}$ or $\mathbb{C}$. And a lot of the properties of $\mathbb{Z}$ is applicable to polynomials with coefficients in $\mathbb{Q}$ or $\mathbb{C}$.

Let us take a look at irreducibility and primes in regards to polynomials.

Definition 2.5 **Divisibility in** $\mathbb{Q}[x]$
Let $f, g \in \mathbb{Q}[x]$. f is a **divisor** of g, denoted $f \mid g$, if there exists $h \in \mathbb{Q}[x]$, such that $g = fh$. Otherwise we say that f is not a **divisor** of g and write $f \nmid g$.

Definition 2.6 **Irreducibility and primes in** $\mathbb{Q}[x]$ We give a classification of the elements in $\mathbb{Q}[x]$. An element in $f \in \mathbb{Q}[x]$ is one of the following

- $f = 0$ and is called the zero polynomial
- $f|1$ and is called a **unit**
- f is called **irreducible** if $f = gh, g, h \in \mathbb{Q}[x] \Rightarrow g$ or h is a **unit**
- f is called **prime** if $f = gh, g, h \in \mathbb{Q}[x]$ and $f|gh \Rightarrow f|g$ or $f|h$ (or both)
- otherwise f is called **reducible**

$f, g \in \mathbb{Q}[x]$ is called **associates** if $f = gu$ where u is a unit.

2.2 Groups

Until now we have looked at elements from sets, and then applied operations to them, both addition and multiplication. But when we look at elements from a set and only are interested in one operation, then we're actually looking at a *group*.

A *group* G is a set with an operation $\circ$ which satisfies

- The **associative** law:

$$\forall\, a, b, c \in G \;:\; (a \circ b) \circ c = a \circ (b \circ c)$$

- Existence of **neutral** or **identity** element:

$$\exists\, e \in G\, \forall a \in G \;:\; e \circ a = a \circ e = a$$

- Existence of **inverses**:

$$\forall\, a \in G\, \exists\, a^{-1} \;:\; a \circ a^{-1} = a^{-1} \circ a = e$$

Definition 2.7 **Abelian group**
A group G is called *Abelian* if

$$\forall\, a, b \in G \;:\; a \circ b = b \circ a$$

Definition 2.8 **Order of group**
The order of group G, denoted $\|G\|$, is the number of elements in G

A finite group is a group with a finite order. Unless otherwise stated the groups we are working with are finite.

Definition 2.9 **Order of group element**
The order of an element $g \in G$, denoted $\|g\|$, is the smallest number a such that $g^a = e$, if a doesn't exist then g has **infinite** order

Definition 2.10 **Cyclic group**
If there exists an element $g \in G$, where G is a group, and G can be written as $G =< g >= (g, g^1, g^2 \ldots)$, then we say that G is a cyclic group and g is a generator for G

Corollary 2.1 Any group G with prime order is cyclic

An example of a group are $\mathbb{Z}$ with $+$ as operator, this group is also Abelian. An example of a cyclic group is the multiplicative subgroup $\mathbb{Z}_n^*$ of $\mathbb{Z}_n$, which contains elements that are relative prime with n, $\mathbb{Z}_n^*$ is also finite.

Definition 2.11 **Subgroup**
H is a subgroup of G, if it contains a subset of the elements of G and satisfies the 4 requirements of a group. It must at least contain the neutral element from G. The order of the subgroup H has to be a divisor of the order of G.

Definition 2.12 **Normal subgroup**
Let H be a subgroup of a group G. Then H is a normal subgroup of G, denoted $H \lhd G$, if for all $x \in G$

$$xHx^{-1} = H$$

2.3 Mappings

A mapping $\sigma : G \mapsto H$ takes an element $g \in G$ and maps it to $\sigma(g) \in H$.
Mappings can be defined for groups, fields and rings.

Definition 2.13 **Homomorphism**
A mapping $\sigma : G \mapsto H$ is called a **homomorphism**, if it maps G to H and preserves the group operation, i.e. for all $g_1, g_2 \in G$ we have $\sigma(g_1 g_2) = \sigma(g_1)\sigma(g_2)$

Definition 2.14 **Isomorphism**
A mapping σ is called an **isomorphism**, if it is a homomorphism and is bijective.

If an isomorphism exists between two groups they are called **isomorphic**.

Definition 2.15 **Automorphism**
A mapping σ is called an **automorphism**, if it is an isomorphism and maps G to G.

A useful theorem, using isomorphism is **The Chinese Remainder Theorem**, which is very helpful when working with finite groups of large order.

Theorem 2.5 **The Chinese Remainder Theorem**
If $n = n_1 n_2 \cdots n_m$ where all n_i's are pairwise relatively prime, that is $\forall 0 < i < j \leq m : \gcd(n_i, n_j) = 1$ then

$$\mathbb{Z}_n \cong \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_1} \cdots \times \mathbb{Z}_{n_m}$$

2.4 Fields and Rings

One of the most important building blocks in abstract algebra is a *field*. A field is a set $\mathbb{F}$ with the following properties

- $(\mathbb{F}, \odot)$ is a multiplicative group
- $(\mathbb{F}, \oplus)$ is an Abelian group

When we look upon the properties for a field, we can see that the set of rational numbers $\mathbb{Q}$ is a field, and so is $\mathbb{C}$. One of the fields of great interest for cryptography over the years is the field $\mathbb{Z}/\mathbb{Z}p$ of integers modulo a prime number p, which is denoted $\mathbb{F}_p$.

Another interesting property of a field is that we can use it to create a finite group, if we have the field $\mathbb{F}_p = \mathbb{Z}/\mathbb{Z}p$, we can create the group $G = \mathbb{F}_p^* = (\mathbb{Z}/\mathbb{Z}p)^*$, which is the group of integers relatively prime with p (the set $1, \ldots, p-1$) and with multiplication modulo p as operator.

Often these properties are not fulfilled and therefore there is another strong object in abstract algebra, and that is a **ring**.

A **ring** $\mathbb{F}$ is a set with all the same properties as a field, except one or more of the following properties are not fulfilled:

- multiplication satisfies commutative law
- existence of multiplicative identity 1
- existence of multiplicative inverses for all elements except 0
- for all $a, b \in \mathbb{F}$ if $a \cdot b = 0$ then $a = 0$ or $b = 0$

Definition 2.16 **Ring homomorphism**
Let R be a ring with operations $+$ and $\cdot$ and let S be a ring with operations $\oplus$ and $\odot$. The map $\theta : R \to S$ is a **ring homomorphism** from the ring $R, +, \cdot$ to the ring $S, \oplus, \odot$ if the following conditions are true for all $a, b \in R$:

1. $\theta(a + b) = \theta(a) \oplus \theta(b)$
2. $\theta(a \cdot b) = \theta(a) \odot \theta(b)$

Definition 2.17 **Characteristic of field**
The characteristic p of a field $\mathbb{F}$, is an integer showing how many times we have to add the multiplicative identity 1 to get 0, if this never happens we say that $\mathbb{F}$ has characteristic 0.

Definition 2.18 **Order of a field**
The order n of a field is the number of elements it contains.

Definition 2.19 **Finite Field**
A field is called *finite* if it has a finite number of elements, i.e. a finite order.

Definition 2.20 A field $\mathbb{F}$ is *algebraically closed* if every polynomial $p \in \mathbb{F}[x]$ has roots in $\mathbb{F}$.

Carl Friedrich Gauss (1777-1855) is considered to be one of the greatest mathematicians ever, he worked on algebra and number theory. And a major part of his work was a large publication on number theory in 1801[29].

Definition 2.21 **Field Extension**
E is called an extension field of a field F if F is a subfield of E.

A field can be extended with one or more elements which results in an extension field. This is normally done with roots of a polynomial. $\mathbb{Q}$ can for example be extended with the roots of the polynomial $t^2 - 5$, this would be denoted $\mathbb{Q}(\sqrt{5})$. $\mathbb{Q}(\sqrt{5})$ is the smallest field that contains $\mathbb{Q}$ and $\sqrt{5}$ hence must contain all the roots of $t^2 - 5$.

An extension with a root from a given polynomial is called a *radical extension*.

A polynomial $f(t)$ is said to split over the field $\mathbb{F}$, if it can be written as a product of linear factors all from $\mathbb{F}$. In other words: all of $f(t)$'s roots are in $\mathbb{F}$.

Definition 2.22 **Splitting field**
$\mathbb{F}$ is a splitting field for the polynomial f over the field $\mathbb{G}$ if $\mathbb{G} \subseteq \mathbb{F}$, and $\mathbb{F}$ is the smallest field with these properties.

From the definition above it is clear that the splitting field $\mathbb{G}$ of f over the field $\mathbb{F}$ equals $\mathbb{F}(r_1\ r_2\ \ldots r_n)$, where $r_1 \ldots r_n$ are the roots of f in $\mathbb{F}$.

The next few theorems appear to belong in Section but they are used on elements from fields, so it is justified to have them here instead. The reciprocity laws are considered among the greatest laws in number theory and *Gauss* even called them the golden theorems. For a very thorough look at the history and theory of reciprocity laws see [38].

Definition 2.23 **Quadratic Residue**
A integer n is said to be quadratic residue modulo p if n is a square in $\mathbb{F}_p$, $p > 2$, otherwise it is called a non-residue

Definition 2.24 **The Legendre symbol**
Let a be an integer and $p > 2$ a prime. Then the *Legendre* symbol $(\frac{a}{p})$ is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{if } p \mid a \\ 1, & \text{if } a \text{ is a quadratic residue} \mod p \\ -1, & \text{if } a \text{ is a non-residue} \mod p \end{cases}$$

Definition 2.25 **The Jacobi symbol**
Let a be an integer and n an odd composite number with prime factorization $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_m^{\alpha_m}$. Then the *Jacobi* symbol $(\frac{a}{n})$ is defined as the product of the *Legendre* symbols for the prime factors of n

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdot \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_m}\right)^{\alpha_m}$$

A note on the last two definitions. In typographic terms the *Jacobi* and *Legendre* symbols are the same, but remember that they are not the same. *Legendre* is modulo a prime and *Jacobi* is modulo a composite.

The reader have probably noticed that none of these symbols works for quadratic residues modulo 2. This is because 2 is a special case and all numbers on the form $8n + 1$ and $8n - 1$ for $n \in \mathbb{Z}$ are quadratic residues modulo 2.

2.5 Galois Theory

In the nineteenth century, the great *Galois* brought many interesting ideas to the algebraic society. Although many of the ideas were not accepted right away, his genius was recognized years later and today he is considered to be one of the biggest algebraists ever.

Going through all the ideas and theory developed by the young *Galois* deserves a couple of papers by itself and is out of the scope of this paper. For a complete look into the work of *Galois* the reader should turn to [6] and [73].

We will only take a look on some of the main ideas of *Galois* and the theory needed later in this paper. This is especially the concept of *Galois Groups* and *Galois fields*.

In the last sections we have seen how a field can be extended by using elements like roots in a polynomial. *Galois* took the concept of field extensions and cooked it down to the core elements.

Evariste Galois (1811-1832) had ideas which expanded algebra to what it is today. His life is one of the biggest tragedies in all of mathematical history. For a look into the life of **Galois** take a look at the excellent book [63].

Definition 2.26 **Galois group**
Let K be a subfield of F, the set of all automorphisms τ in F, which are such that $\tau(k) = k$ for all $k \in K$, forms a group, called the Galois group of F over K and denoted Gal(F/K).

As a consequence of *Galois*' work, it became clear that all finite fields contains p^n elements, where p is a prime and n some integer. These fields can all be constructed as a splitting field for $t^n - t$ over $\mathbb{Z}_p$. This discovery is the reason for naming finite fields after the great *Evariste Galois*.

Definition 2.27 **Galois Field**
The finite field $\mathbb{F}_n$ with n elements, where $n = p^m$ for some prime p, is written **GF**(n).

The definition of a Galois Field adds another name for a finite field, so we actually have three names for a finite field: $\mathbb{Z}_p, \mathbb{F}_p$ and **GF**(p) in this thesis a finite field is denoted **GF**(p).

Definition 2.28 **Characteristic of GF**(p^n)
The characteristic of a Galois Field **GF**(p^n), is p

Although the most known terms are Galois group and Galois Field, *Galois*' main work was on the theory of radical extensions and solvable groups.

Definition 2.29 **Solvable group**
A group is called solvable if it has a finite sequence of subgroups

$$1 = G_0 \subseteq G_1 \subseteq \cdots \subseteq G_n = G$$

such that $G_i \lhd G_{i+1}$ and G_{i+1}/G_i is Abelian for $i = 0, \ldots, n$

Theorem 2.6 Let $\mathbb{F}$ be a field with characteristic 0. The equation $f = 0$ is solvable in radicals if and only if the Galois group of f is solvable.

Although some of these definitions looks simple, they are very hard to work with. One of the computational difficult subjects in mathematics is solving the *Inverse Galois Problem*, that is verifying that a given group is a Galois group for some polynomial in a field extension. This is not relevant to this thesis, but the interested reader should take a look at [34].

2.6 Elliptic Curves

Elliptic curves is a branch of algebra which has received a lot of attention over the last couple of decades. The reason for algebraists' interest in elliptic curves is that the set of points on an elliptic curve along with a special operation can be made into a group. The general theory of elliptic curves is beyond the scope of this paper, but we will take a look at some of the theory of elliptic curves, so we can illustrate how to create groups based on elliptic curve.

For a more comprehensive treatment of the theory of elliptic curves, the reader should turn to [69], [8],[42] or [27] and the more recent publication [31], which is the most comprehensive guide I have ever read, to the implementation issues of elliptic curves.

2.6.1 Introduction

Elliptic curves should not be confused with ellipses, even though they share the same name. Elliptic curves are the result of a special kind of functions, namely the elliptic functions.

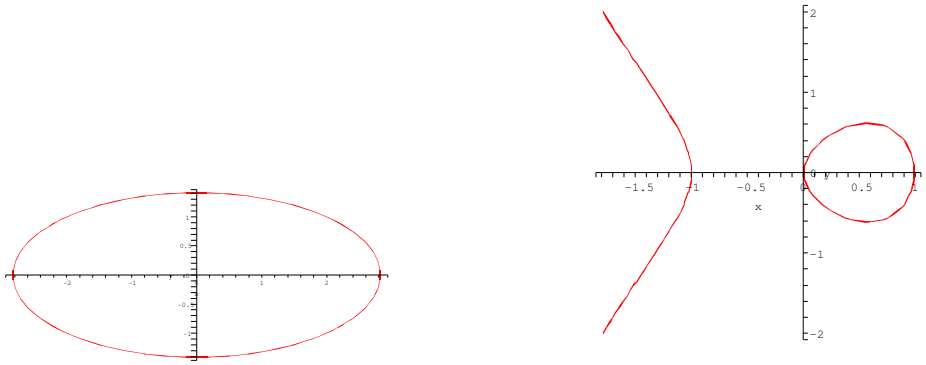

Figure 2.1: An ellipse is shown on the left and a projection of the elliptic curve: $y^2 + x^3 - x$ on the right

Elliptic functions have been studied by many of the great mathematicians through history, especially the work of *Galois*[28] and *Abel*[2] in this area is notable.

For mathematicians, an elliptic function is a meromorphic function[1] defined in the complex plane and is periodic in two directions, analogous to a geometric function which is only periodic in one direction. You can just consider elliptic functions as being polynomials in 2 variables.

2.6.2 The Weierstraß Form

Karl Theodor Wilhelm Weierstraß (1815-1897) One of the pioneers on hyperelliptic integrals. He taught prominent mathematicians like *Cantor, Klein* and *Frobenius* to mention a few.

Elliptic curves are often written on the *Weierstraß form* although there exists other equations which can be used like the *Legendre equation* and *quartic equation*. In this paper we will use the Weierstraß equation to describe elliptic curves, because the theory we need are best described in the Weierstraß form.

Definition 2.30 **Generalized Weierstraß equation**
Let K be a field. An equation on the form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad ,$$

with $a_i \in K$, is called the generalized *Weierstraß equation* over K in the affine plane[2].

[1] A *meromorphic function* is a single-valued function that is analytic in all but possibly a finite subset of its domain, and at those singularities it must go to infinity like a polynomial, i.e. the singularities must be poles and not essential singularities.

[2] For a field $\mathbb{F}_n$, the *affine plane* consists of the set of points which are ordered pairs of elements in $\mathbb{F}$ and a set of lines which are themselves a set of points.

In the Weierstraß form, the x and y cover a plane, and they can belong to any kind of algebraic closure.

Interesting properties of a Weierstraß equation is its discriminant Δ and its j-invariant j. In most literature on the subject, it is common practice to use a few intermediate results to express Δ and j, these are

$$\begin{aligned} b_2 &= a_1^2 + 4a_2 \\ b_4 &= 2a_4 + a_1a_3 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \\ c_4 &= b_2^2 - 24b_4 \end{aligned}$$

The discriminant and j-invariant can then be written as

$$\begin{aligned} \Delta &= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \\ j &= \frac{c_4^3}{\Delta} \end{aligned}$$

Definition 2.31 **Singularity**
A point p is a called a *singular point*, if the partial derivatives of the curve are all zero at the point p. Singular points could for example be

- a *cusp* which is a point where the tangent vector changes direction
- a cross intersection point where the curve "crosses" itself

Definition 2.32 A curve is called *singular* if it has one or more singular points, otherwise it is called *non-singular*.

Theorem 2.7 If a Weierstraß equation has discriminant $\Delta = 0$ then it is singular.

These theorems makes it possible to define what an elliptic curve is: A non-singular Weierstraß equation defines an elliptic curve. When using the term: "elliptic curve E over a field K" I mean the solutions in K to the curve's Weierstraß equation the set of which is denoted E/K.

Definition 2.33 Two elliptic curves defined over the same base field are *isomorphic* if their j-invariant are the same.

When using an elliptic curve over a Galois Field, we can estimate the number of points by using *Hasse's theorem*, which was originally conjectured by Artin in his thesis for general curves and proved later by Hasse for elliptic curves. A proof can be found in [69].

Theorem 2.8 **Hasse's Theorem**
Given a Galois Field $\mathbf{GF}(q)$, the number of rational points on the elliptic curve $E/\mathbf{GF}(q)$ is denoted $|E/\mathbf{GF}(q)|$:

$$q + 1 - 2\sqrt{q} \leq |E/\mathbf{GF}(q)| \leq q + 1 + 2\sqrt{q}$$

Definition 2.34 **Trace of Frobenius**
For an elliptic curve E over $\mathbf{GF}(q)$, then t defined as

$$t = |E/\mathbf{GF}(q)| - q - 1$$

is called the **Trace of Frobenius**

René Schoof created an algorithm for computing the points on elliptic curves[67], which uses Hasse's theorem along with the Trace of Frobenius. The Schoof algorithm have been improved later by Atkin.

Describing the algorithm is out of the scope of this paper, but the fact that there exists algorithms for determining the number of points on elliptic curves is important.

Theorem 2.9 If the **Trace of Frobenius** of an elliptic curve E over $\mathbf{GF}(p^n)$, is divisible by p then the elliptic curve E is called *supersingular*.

This definition gives a total classification of supersingular elliptic curves:

Corollary 2.2 Let E be an elliptic curve defined over $\mathbf{GF}(q)$, then E is supersingular if and only if

$$t^2 = 0 \vee t^2 = q \vee t^2 = 2q \vee t^2 = 3q \vee t^2 = 4q$$

where t is the trace of Frobenius

Elliptic curves over a Galois Field work differently depending on the characteristic of the field. The case of characteristic 2,3 and $p > 3$ have to be treated separately.

2.6.2.1 Elliptic Curves Over $\mathbf{GF}(2^n)$

All elliptic curves over $\mathbf{GF}(2^n)$ are isomorphic to one of the two following curves:

$$y^2 + y = x^3 + a_4x + a_6$$

or

$$y^2 + xy = x^3 + a_2x^2 + a_6$$

These curves leads to simplification of Δ and the j-invariant

$$\begin{aligned} E/\mathbf{GF}(2^n): \quad & y^2 + xy = x^3 + a_2x^2 + a_6 \\ & \Delta = a_6 \\ & j = \frac{1}{a_6} \\ E/\mathbf{GF}(2^n): \quad & y^2 + a_3y = x^3 + a_4x + a_6 \\ & \Delta = a_3^4 \\ & j = 0 \end{aligned}$$

The latter curve is supersingular due to the well known result[27], that a curve over $\mathbf{GF}(2^n)$ is supersingular when the j-invariant is 0.

2.6.2.2 Elliptic Curves Over GF(3^n)

All elliptic curves over $\mathbf{GF}(3^n)$ are isomorphic to one of the two following curves:

$$y^2 = x^3 + a_2x^2 + a_6$$

or

$$y^2 = x^3 + a_4x + a_6$$

These curves lead to simplification of Δ and the j-invariant

$$\begin{aligned} E/\mathbf{GF}(3^n): \quad & y^2 = x^3 + a_2x^2 + a_6 \\ & \Delta = -a_2^3a_6 \\ & j = -\frac{a_3^2}{a_6} \\ E/\mathbf{GF}(3^n): \quad & y^2 = x^3 + a_4x + a_6 \\ & \Delta = -a_4^3 \\ & j = 0 \end{aligned}$$

The latter curve is supersingular due to the well known result[27], that a curve over $\mathbf{GF}(3^n)$ is supersingular when the j-invariant is 0.

2.6.2.3 Elliptic Curves Over GF(p^n)

When looking at elliptic curves over prime fields, the equation can be reduced even further. All elliptic curves over $\mathbf{GF}(p^n)$ for $p > 3$ a prime are isomorphic to the following curve:

$$y^2 = x^3 + a_4x + a_6$$

This curve leads to simplification of the discriminant Δ and the j-invariant

$$\begin{aligned} E/\mathbf{GF}(p^n): \quad & y^2 = x^3 + a_4x + a_6 \\ & \Delta = -16\left(4a_4^3 + 27a_6^2\right) \\ & j = 1728\frac{4a_4^3}{4a_4^3 + 27a_6^2} \end{aligned}$$

2.6.2.4 Group Law

Now I will show how elliptic curves can be used to construct algebraic groups, by using points on the curve.

Theorem 2.10 Points on an elliptic curve, E defined over $\mathbf{GF}(p^n)$, along with a special addition, $\oplus$, forms an Abelian group $G = (E, \oplus)$, with $\mathcal{O}$ as identity element.

This special addition is defined as follows. Let $P = (x_1, y_1)$,$Q = (x_2, y_2) \in E$, where E is defined as in definition 2.30 and let $\mathcal{O}$ be the point at infinity, then

- $\mathcal{O} \oplus P = P \oplus \mathcal{O} = P$
- $\ominus\mathcal{O} = \mathcal{O}$
- $\ominus P = (x_1, -y_1 - a_1x_1 - a_3)$
- if $Q = \ominus P$ then $P \oplus Q = \mathcal{O}$
- if $Q = P = (x, y)$ then $Q \oplus P = (x_3, y_3)$, where
  $x_3 = \left(\frac{3x^2+2a_2x+a_4-a_1y}{2y+a_1x+a_3}\right)^2 + a_1\left(\frac{3x^2+2a_2x+a_4-a_1y}{2y+a_1x+a_3}\right) - a_2 - 2x$
  $y_3 = \left(\frac{3x^2+2a_2x+a_4-a_1y}{2y+a_1x+a_3}\right)(x - x_3) - y - (a_1x_3 + a_3)$
- if $Q \neq P$ then $Q \oplus P = (x_3, y_3)$, where
  $x_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)^2 + a_1\left(\frac{y_2-y_1}{x_2-x_1}\right) - a_2 - x_1 - x_2$
  $y_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1 - x_3) - y_1 - (a_1x_3 + a_3)$

2.6.2.5 Elliptic Curves and Rings

It is also possible to define an elliptic curve over the ring $\mathbb{Z}_n$ where n is composite, this is interesting because they are used in various factoring and primality testing algorithms like the *Atkin-Goldwasser-Killian* primality test[47].

An elliptic curve over $\mathbb{Z}_n$, where $gcd(n, 6) = 1$, is isomorphic to the following curve[42]:

$$\begin{aligned} y^2 &= x^3 + a_4x + a_6 \\ \Delta &= 4a_4^3 + 27a_6^2 \end{aligned} \tag{2.1}$$

where $a_4, a_6 \in \mathbb{Z}_n$ and $gcd(\Delta, n) = 1$.

An elliptic curve $y^2 = x^3 + ax + b$ over the ring $\mathbb{Z}_n$ is denoted $E_n(a, b)$.

The points on an elliptic curve defined over a ring is not a group under the addition defined in the previous section. This is due to the fact that the addition is not defined for all points. The cases where it is not defined are the ones which would involve a division by a non-invertible element in $\mathbb{Z}_n$.

The cases where the $\oplus$ addition of two points, $P = (x_1, y_1), Q = (x_2, y_2)$ is not defined are

- if $P \neq Q$ the case where $gcd(x_2 - x_1, n) > 1$
- if $P = Q$ the case where $gcd(2y_1, n) > n$

Both of these cases would yield a non-trivial factor of n.

We define an operation $\oplus$, often denoted as "pseudo addition" in the literature, on elements in $E/\mathbb{Z}_n$. This special addition is defined as follows.

Let $P = (x_1, y_1), Q = (x_2, y_2) \in E$, where E is defined as in equation 2.1 and let $\mathcal{O}$ be the point at infinity, then

- $\mathcal{O} \oplus P = P \oplus \mathcal{O} = P$
- $\ominus\mathcal{O} = \mathcal{O}$
- $\ominus P = (x_1, -y_1 - a_1x_1 - a_3)$
- if $Q = \ominus P$ then $P \oplus Q = \mathcal{O}$
- let $p = gcd(x_2 - x_1, n)$ if $1 < p < n$ then p is a non-trivial divisor of n, and the addition fails
- let $p = gcd(y_1 + y_2, n)$ if $1 < p < n$ then p is a non-trivial divisor of n, and the addition fails
- if $gcd(y_1 + y_2, n) = n$ then $P \oplus Q = \mathcal{O}$
- if $Q = P = (x, y)$ then $Q \oplus P = (x_3, y_3)$, where
  $x_3 = \left(\frac{3x^2+a_4}{2y}\right)^2 - 2x$
  $y_3 = \left(\frac{3x^2+a_4}{2y}\right)(x - x_3) - y$
- if $Q \neq P$ then $Q \oplus P = (x_3, y_3)$, where
  $x_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)^2 - x_1 - x_2$
  $y_3 = \left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1 - x_3) - y_1$

Points on elliptic curves over finite rings behave similarly to elliptic curves over fields, except for the finite number of points which lead to failure of the addition operation and actually reveals a non trivial factor of n. This is the reason elliptic curves over rings are used in primality proving algorithms.

When $n = pq$ where p and q are primes of approximately the same sizes, then the "pseudo addition" described above fails only if we actually have factored n which we find as an unlikely situation. Therefore the points on an elliptic curve defined over the ring $\mathbb{Z}_{pq}$, along with $\oplus$ will define a "pseudo" Abelian group.

The set of points on an elliptic curve defined over the ring $\mathbb{Z}_{pq}$ will constitute a group because it will be the direct product of two groups, and from *The Chinese Remainder Theorem* we get:

$$G_{E/\mathbb{Z}_{pq}} \cong G_{E/\mathbb{Z}_p} \times G_{E/\mathbb{Z}_q} = G_{E/GF(p)} \times G_{E/GF(q)}$$

CHAPTER 3

The History Of Integer Factorization

"It appears to me that if one wishes to make progress in mathematics, one should study the masters and not the pupils."

- Niels Henrik Abel (1802-1829)

I believe in the importance of studying the origins of the theory one uses. I will give an overview of the turning points in the history of integer factorization which eventually lead to the number field sieve. For a complete reference on the history of primality testing and integer factorization I refer to [77] and [62] where references to the original articles can be found.

The history reflects that integer factorization requires great computational powers compared to primality testing which does not require the same computations, and the history of primality testing starts earlier and in 1876 Lucas was able to verify the primality of a 39 digit number in comparison it was not until 1970 and by using computers that Morrison and Brillhart were able to factor a 39 digit composite integer.

Before Fermat

The concept of factoring numbers into primes has been around since Euclid defined what primes are and the idea of unique factorization with the fundamental theorem of arithmetic ~ 300 B.C.

Early development of mathematics was mainly driven by its use in business or in general life and factoring of integers do not have a use in neither business nor everyday life so its development have always been driven by theoretical interest but since the late 1970's it is not driven by theoretical interest but by security interest.

There is no indications that any methods other than trial division existed before the time of Fermat and so my recap of the history of factoring integers starts with Pierre de Fermat.

Fermat (~ 1640)

In 1643 Fermat showed some interesting ideas for factoring integers, ideas that can be traced all the way to the fastest methods today. Fermat's idea was to write an integer as the difference between two square numbers, i.e. he tried to find integers x and y such that the composite integer n could be written as $n = x^2 - y^2$ thereby revealing the factors $(x+y)$ and $(x-y)$. Fermat's method is very fast if the factors of n is close otherwise it is very inefficient.

Euler (~ 1750)

As the most productive mathematician ever Euler of course also had some thoughts on integer factorization. He only looked at integers on special forms. One of his methods is only applicable to integers that can be written on the form $n = a^2 + Db^2$ in two different ways with the same D.

The method proceeds as Fermat's method for finding the numbers that are representable in the way described. He used the method to successfully factor some large numbers for that time.

And ironically he also used his method to disprove one of Fermat's theorems.

Legendre (~ 1798)

Legendre presented the idea that eventually would evolutionize factoring algorithms.

Legendre's theory of congruent squares which I will describe in details in Chapter 5 is the core of all modern general factorization algorithms. At the time of Legendre the lack of computational power limited the size of the numbers that could be factored but Legendre's idea would prove itself to be the best method when computational power would be made available by various machines and at the end by computers.

Gauss (~ 1800)

At the same time that Legendre published his ideas was Gauss working on the most important work on number theory. And in 1801 was *Disquisitiones Arithmeticae* published which had several idea and methods for factoring integers.

Gauss' method is complicated but can be compared to the sieve of Erastothenes which means that it works by finding more and more quadratic residues mod n and thereby excluding more and more primes for being possible factors, when there is a few elements left they are trial divided.

The idea of sieving elements is an important element of all modern factorization method and I will show this in Chapter 5.

In the years following Legendre and Gauss no new ideas emerged. Even

with the methods of Gauss and Legendre it was still hard to factor integers with more than 10-15 digits and even that took days and a lot of paper and ink, so mathematicians of the time did not "waste" their time doing manual computations. It was not until someone decided to build a machine to do the sieving that the problem of factoring was revisited.

Enter the machines...

In the end of the 19th century several people independent of each other built various types of machines to do some of the tedious computations of sieving based factoring algorithms.

In 1896 Lawrence described a machine which used movable paper strips going trough movable gears with teeth where the number of teeth represent an exclusion modulus and the places on the paper that is penetrated represents acceptable residues. Although it seems like the machine was never built it inspired other to actually built similar machines.

In 1910 a French translation of Lawrence's paper was published and shortly thereafter Maurice Kraitchik built a machine with the ideas of Lawrence. At the same time G´erardin and the Carissan brothers build similar machines, but it was not before after the first world war that the Carissan brothers built the first workable sieving machine showing good results (the machine was hand driven).

The most successful machine builder was Lehmer who was seemingly unaware of the work of the Carissans and Kraitchik for many years and he built numerous sieving devices some of them pushing contemporary technology of his time.

The machines by Kraitchik and Lehmer and the algorithms they use were very similar to the later developed quadratic sieve algorithm.

Late 20th Century

As far up as to 1970 the sieving machines were the fastest factoring methods, but at the time computers could provide vastly more computational power all they needed was a usable algorithm.

The 1970's provided many factoring algorithms, starting with Daniel Shank using quadratic forms (like Gauss) in the SQUFOF algorithm which is quite complicated and not trivial to implement. It has been used successfully over the years in different versions.

John Pollard was very active and developed the $p - 1$ method in 1974 and the year after the ρ method they were both targeted at a special class of integers. Morrison and Brillhart presented the first fast general purpose algorithm CFRAC in 1975, based on continued fractions like used originally by Legendre.

Brent optimized the ρ method in 1980 and around 1982 Carl Pomerance invented the quadratic sieve algorithm that added some digits to the numbers that could be factored resulting in a factorization of a 71 digit number in 1983.

Schnorr and Lenstra took the ideas of Shanks and improved vastly on the

original method in 1984, and in 1987 Lenstra took a completely different road and developed a new method by using elliptic curves in his ECM algorithm, which is specialized for composites with small factors.

31. August 1988 John Pollard sent a letter to A. M. Odlyzko with copies to Richard P. Brent, J. Brillhart, H. W. Lenstra, C. P. Schnorr and H. Suyama, outlining an idea of factoring a special class of numbers by using algebraic number fields. Shortly after the number field sieve was implemented and was generalized to be a general purpose algorithm.

The number field sieve is the most complex factoring algorithm but it is also the fastest and has successfully factored a 512 bit composite.

Since the number field sieve arrived around 1990 there has not been any big new ideas, only optimizations to the number field sieve.

CHAPTER 4

Public Key Cryptography

"When you have eliminated the impossible, what ever remains, however improbable must be the truth."

- Sir Arthur Conan Doyle(1859-1930)

Integer factorization is interesting in itself as one of number theory's greatest problem but it is even more interesting it is because of its use in cryptography.

In this Chapter I will give an introduction to public key cryptography and especially the RSA scheme as it is built on the integer factorization problem.

4.1 Introduction

The word *cipher* has been used in many different contexts with many different meanings. In this paper it means a function describing the encryption and decryption process. A cryptosystem consists of a cipher along with one or two keys.

Cipher: any method of transforming a message to conceal its meaning. The term is also used synonymously with cipher text or cryptogram in reference to the encrypted form of the message.
Encyklopædia Britannica

Cryptosystems can be categorized into *symmetric* and *asymmetric*:

- *symmetric* cryptosystems use a shared secret in the encryption process as well as in the decryption process. Cryptosystems in this category includes **DES**[74], **AES**[22], and **Blowfish**[66].
- *asymmetric* cryptosystems uses a publicly available key for encryption and a private key for decryption, this means that there is no need for a shared secret. Asymmetric cryptosystems are also known as public-key cryptosystems. Cryptosystems in this category includes **RSA**[64], **ElGamal**[25], and **Diffie-Hellman**[23].

In this paper I will only look at public-key cryptography because they are prone to number theoretic attacks like factoring, (for an introduction to all kinds of cryptography see [74] or [43]).

4.1.1 How It All Started

Throughout history a lot of importance has been given to cryptography and the breaking of these systems, a lot of historic researchers give cryptanalysts a great deal of credit for the development of the second world war[81].

The history of public-key cryptosystems has been driven by the need for distribution of common secrets, like keys for symmetric cryptosystems. The increasing computational power has also had its share in the progress of these systems. The history of public-key cryptography began in 1976, or so it was believed until 1997 where a number of classified papers were made public which showed that different people have had the same ideas.

One of the classified papers was [26], where *James Ellis* had stumbled upon an old *Bell Labs* paper from 1944, which described a method for securing telephone communication without no prior exchanging of keys.

James Ellis wrote in the paper of the possibility of finding a mathematical formula which could be used. Three years later *Clifford Cocks* discovered the first workable mathematical formula in [18], which was a special case of the later discovered **RSA**, but like James Ellis' paper, it was also classified.

A few months after Cocks' paper, *Malcolm Williamson* made a report [80] where he discovered a mathematical expression and described its usage as very similar to the later discovered key exchange method by *Diffie* and *Hellman*.

Ellis, Cocks and Williamson didn't get the honor of being the inventors of public-key cryptography because of the secrecy of their work, and who knows if there was someone before them ?

It was from the academic world the first publicly available ideas came. *Ralph Merkle* from Berkeley defined the possibility of public-key encryption and the concepts were refined by Whitfield Diffie and Martin Hellman from Stanford University in 1976 with their ground-breaking paper [23].

The paper revealed ideas like key exchanging which is still widely used today along with some mathematical ideas which could lead to usable functions.

Three researchers at MIT read the paper by Diffie and Hellman and decided to find a practical mathematical formula to use in public-key cryptosystems and they found it. The researchers where *Ron Rivest*, *Idi Shamir* and *Len Adleman* and they invented the **RSA** cryptosystem with [64]. At the same time, *Robert James McEliece* invented another public-key cryptosystem based on algebraic error codes equivalent to the *subset sum problem* in the paper [41].

The two papers ([64],[41]) started a wave of public-key cryptosystems. In 1985 *ElGamal* defined , in the paper [25], a public-key cryptosystem based on the *discrete logarithm problem*.

Since the first paper on public-key cryptography a lot of different systems have been proposed, a lot of the proposed systems are similar but some of them present really unique ideas. The only thing they all have in common is their algebraic structure.

For a complete, and quite entertaining, view at the birth of public-key cryptography, the interested reader should turn to the book [40], which takes the reader into the minds of the creators of the systems that changed private communication forever.

4.1.2 The Theory

We use the names ***Alice*** and ***Bob*** to describe two persons communicating, and ***Eve*** to describe an eavesdropper trying to intercept the communication between ***Alice*** and ***Bob***.

A plain text is denoted $\mathcal{M}$.

A cipher text is denoted $\mathcal{C}$.

Bob's private key is denoted d_B and ***Alice***'s private key is denoted d_A (d as $d$ecryption).

Bob's public key is denoted e_B and ***Alice***'s public key is denoted e_A (e as $e$ncryption).

A cipher, $\mathcal{K}$ are the needed functions and parameters, like public and private key.

Encryption of plain text $\mathcal{M}$ with the cipher $\mathcal{K}_B$ is denoted $E_{\mathcal{K}_B}(\mathcal{M})$.

Decryption of cipher text $\mathcal{C}$ with the cipher $\mathcal{K}_B$ is denoted $D_{\mathcal{K}_B}(\mathcal{C})$.

Although public-key cryptosystems comes from a wide variety of mathematical theory, they all share the same overall structure which can be formalized.

In this Section we will take a look at the, somewhat simple, theory that makes it possible to create public-key cryptosystems.

The term *intractable* is used throughout this thesis and it means that the computational time used for solving a problem, is larger than current computational powers permit to solve in reasonable time. For example a problem which takes one billion years on all computers in the world to solve is considered to be intractable.

Some complexity classes contain only intractable problems, but one should be aware that not all instances of a hard problem are intractable. It is possible to find instances of problems in the complexity classes **NP** and **co-NP** that are far from intractable.

Definition 4.1 **One-way function**
A function f is a *one-way function* if it is easy to compute f, but difficult to compute f^{-1}. We say that f is not computationally invertible when computing f^{-1} is *intractable*.

Definition 4.2 **One-way trapdoor function**
A one-way function $f(x)$ is called a *one-way trapdoor function*, if it is easy to compute $f'(x)$ with some additional information, but intractable without this information.

Definition 4.2 is the core of public-key cryptography, and it can be used to define a public-key cryptosystem.

A public-key cryptosystem consists of the following:

$E_v(M)$	:	a one-way trapdoor function with some parameter v on M
$D_v(M)$	:	the inverse function to $E_v(M)$
x	:	a chosen value for the parameter v of $E_v(M)$
y	:	the trapdoor used to calculate $D_x(C)$

The one-way trapdoor function $E_v(M)$ is the actual encryption function, x is the public key used along with $E_v(M)$ to encrypt the input M into the cipher text C.

The private key y is the trapdoor used to construct the decryption function, $D_x(C)$, which is the inverse to the encryption function with parameter x.

The original input M can be recovered by $D_x\left(E_x\left(M\right)\right)$.

The private and public keys, x, y are dependent, which means that separately they are of no use.

RSA is the most widespread public-key cryptosystem. It was invented and patented in 1977. The patent was owned by *RSA Laboratories* and the use was therefore limited to the companies which purchased a license.

It was the strongest public-key cryptosystem of its time, but the high royalties and the big pressure from especially the *NSA* prevented the system to be widely adopted as the standard for public-key encryption and signing. NSA prevented that the algorithm was shipped outside the US.

The patent expired on midnight at the 20. September 2000, and could from this date be used freely in non commercial as well as commercial products. This is what makes RSA still interesting so many years after its invention.

In this Chapter we will take a look at the mother of all public-key systems, and formalize it in such a way that the key elements are obvious.

4.2 RSA

The RSA paper in 1978[64] proposed a working system displaying the ideas of *Whit Diffie* and *Martin Hellman*.

RSA is based on a very simple number theoretic problem and so are a lot of the public-key cryptosystems known today.

In general the public-key cryptosystems known today based on number theoretic problems can be categorized into three groups:

- Intractability is based on the extraction of roots over finite Abelian groups, in this category is **RSA**, **Rabin**[61], **LUC**[72]...
- Intractability is based on the discrete logarithm problem, in this category is the **Diffie-Hellman scheme**[23], **ElGamal**[25]...
- Intractability is based on the extraction of residuosity classes over certain groups, in this category we have **Goldwasser-Micali**[74], **Okamoto-Uchiyama**[53]...

We are mainly interested in the ones based on factoring, but in Section 4.3 I will give a short introduction to discrete logarithm based schemes because the problem can be solved by factoring aswell.

I will use RSA as an example of a factoring based scheme afterwards I will briefly discuss other factoring based algorithms by formalizing the RSA scheme.

To use RSA encryption, one has to have a public and a private key, these are created before using the scheme.

example: **The original RSA scheme**

Setting up RSA:

1. choose two primes p and q
2. calculate $n = pq$
3. calculate $\phi(n) = (p-1) \cdot (q-1)$
4. choose d such that $\gcd(d, \phi(n)) = 1$
5. choose e as the multiplicative inverse of d, ie. $ed \equiv 1 \pmod{\phi(n)}$

The pair (e, n) is the public key and (d, n) is the private key

To encrypt a plain text $\mathcal{M}$ into the cipher text $\mathcal{C}$:

1. represent $\mathcal{M}$ as a series of numbers $\mathcal{M}_0, \ldots, \mathcal{M}_j$ in the range $0, \ldots, n-1$
2. for all $\mathcal{M}_i \in \mathcal{M}_0, \ldots, \mathcal{M}_j$ calculate $\mathcal{C}_i = \mathcal{M}^e \pmod{n}$
3. the resulting cipher text $\mathcal{C}$ is $\mathcal{C}_0, \ldots, \mathcal{C}_j$

To decrypt a cipher text $\mathcal{C}$ into the plain text $\mathcal{M}$:

1. represent $\mathcal{C}$ as a series of numbers $\mathcal{C}_0, \ldots, \mathcal{C}_j$ in the range $0, \ldots, n-1$ (this should be identical to the resulting series in the encryption phase)
2. for all $\mathcal{C}_i \in \mathcal{C}_0, \ldots, \mathcal{C}_j$ calculate $\mathcal{M}_i = \mathcal{C}^d \pmod{n}$
3. the resulting plain text $\mathcal{M}$ is $\mathcal{M}_0, \ldots, \mathcal{M}_j$

To make this scheme more clear let us look at a short example: ***Alice*** wants to talk secretly with ***Bob***, she obtains ***Bob***'s public key (e_B, n_B) from some authority. She then calculates $\mathcal{C} = \mathcal{M}^{e_B} \pmod{n_B}$, where $\mathcal{M}$ is the message she wants to give him. She sends the encrypted message $\mathcal{C}$ to ***Bob***.

Bob then decrypts the message from ***Alice*** using his private key (d_B, n_B) in $\mathcal{M} = \mathcal{C}^{d_B} \pmod{n_B}$ and can now read the secret message from ***Alice***.

4.2.1 Correctness

The reader can surely see the beauty in this simple system. The RSA scheme consists of a *permutation polynomial* and its inverse. We have just seen how these polynomials look like, now we will verify that they work as expected.

The selection of e as the multiplicative inverse of d and $\gcd(d\phi(n)) = 1$ ensure us that

$$\begin{aligned} ed &\equiv 1 \pmod{\phi(n)} \\ ed &= k\phi(n) + 1 \end{aligned}$$

for some positive integer k.

The encryption process is:

$$M^e \pmod{n}$$

Using the decryption process on this gives

$$\begin{aligned} (M^e)^d \pmod{n} &= M^{ed} \pmod{n} \\ &= M^{k\phi(n)+1} \pmod{n} \end{aligned}$$

Fermat's Little Theorem (Theorem 2.4) implies that for any prime p and for all M where $p \nmid M$

$$M^{k'(p-1)} \equiv 1 \pmod{p} \quad \text{so} \quad M^{k'(p-1)+1} \equiv M \pmod{p}$$

for any positive integer k', the last one even works for all M.

using both p and q we get

$$\begin{aligned} M^{k'(p-1)+1} &\equiv M \pmod{p} \\ M^{k'(q-1)+1} &\equiv M \pmod{q} \end{aligned}$$

since p and q are different primes we have

$$\begin{aligned} M^{k'(p-1)(q-1)+1} &\equiv M \pmod{pq} \\ M^{k'\phi(n)+1} &\equiv M \pmod{n} \\ M^{ed} &\equiv M \pmod{n} \end{aligned}$$

4.2.2 Formalizing RSA

The RSA scheme as described in the previous Section, is well defined and there is no degree of freedom to the implementer. This is mostly because of the numbers it works on and the permutation polynomials used.

In this Section we will try to dismantle the RSA scheme down to its core algebraic elements, and thereby free some choices to the implementer. This broken down scheme will be called the *general RSA scheme*.

When breaking down an original scheme, one has to be aware of the ideas behind the original scheme, this means that the original scheme should be an instance of the general scheme.

Why it is interesting to formalize the scheme ?

By formalizing the scheme one can identify the algebraic properties and extend the scheme to work in other fields/groups/rings. This means that the underlying problem the security depends on is transformed into a new realm where it can be even harder to solve.

RSA has two main ingredients: the ring it works in and the permutation polynomials.

Now we will take a look at these two core elements of the RSA scheme and generalize them in such a way, so that the *RSA-like* schemes can be identified.

4.2.2.1 The ring $\mathbb{F}_n$

The original scheme works in the ring $\mathbb{Z}_{pq}$ which has all the properties of a field, except for the existence of multiplicative inverses to the elements divisible by p or q. It is somewhat more correct to say that RSA works in a subgroup of this ring namely the group $\mathbb{Z}/\mathbb{Z}pq$.

The general RSA scheme works on a ring $\mathbb{F}_n$, which is not necessarily isomorphic to $\mathbb{Z}_{pq}$. This ring can be chosen freely as long as it has a multiplicative identity 1, and if the scheme should be used as a signature scheme, multiplication has to be Abelian.

4.2.2.2 Permutation polynomials

The RSA scheme consists of two permutation polynomials which are inverses of each other and work on elements from the ring $\mathbb{Z}_{pq}$.

A permutation polynomial is a bijective mapping from G to G, also known as an automorphism.

The permutation polynomials used in the original scheme are:

$$\begin{aligned} E(x) &= x^e \pmod n \\ D(x) &= x^d \pmod n \end{aligned}$$

It is important that these functions work as one-way trapdoor function, which implies that calculating the inverse $E'(x)$ of $E(x)$, is intractable, but utilizing the knowledge of the private key (d, n) and its structure which implies that $D(x) = E'(x)$.

The trapdoor used is the knowledge of the private key (d, n), and the fact that it along with the polynomial $D(x)$ constitutes the inverse mapping of $E(x)$.

In the general scheme any permutation polynomial can be used, but this is probably the biggest deviation from the original scheme, because the permutation polynomials used in the original scheme are the "heart and soul" of RSA.

4.2.3 RSA Variants

In the previous Section we described the two building blocks of the RSA scheme and it is these two parameters that can be altered to create a RSA-like scheme.

From the description of the two building blocks, it should be quite intuitive which schemes can be called RSA-like.

We will take a look at some of the schemes proposed since the original paper. We will not cover all RSA-like systems but look at some representatives of these.

The RSA-like cryptosystems known today can be categorized in the obvious two categories:

- Schemes using other groups
- Schemes using other permutation polynomials

Since the original paper in 1978, every public-key cryptosystem whose security relies on the intractability in factoring large integers is called RSA-like.

This Section will use the properties of RSA as described in Section 4.2.2 to classify different schemes as being RSA-like or not.

4.2.3.1 Rabin and Williams

Martin O. Rabin presented a "RSA-like" scheme which was provably as hard to solve as integer factorization[61]. The scheme has a major drawback. The decryption function gives four plain texts from which the receiver has to decide the correct one.

Hugh C. Williams presented a modified Rabin scheme which has a mechanism for deciding which of the resulting plain texts are the right one[78].

From the definitions of Section 4.2.2, the Rabin and Williams Schemes can not be classified as RSA-like, because the decryption function is not bijective and thereby not a permutation polynomial, although Williams added a way to choose the correct plain text, the system lacks the strict structure of the general RSA scheme.

The reason for calling the Rabin and Williams schemes "RSA-like" throughout history, is probably the similarity in their security considerations, they all seemingly rely on the intractability in factoring large integers.

The reason for mentioning these schemes is because they have the interesting feature, that it is provable that breaking the schemes is equivalent to factoring the modulus n of the public key.

4.2.3.2 KMOV and Demytko

In the late 1980's much research was done on elliptic curves and their application to cryptography. Using elliptic curves in cryptography was first proposed by Neal Koblitz, but many different schemes followed.

In 1991 the authors of the paper [37] presented a RSA-like scheme using elliptic curves over the ring $\mathbb{Z}_n$. They presented three new trapdoor functions, but only two of them can be used for encryption, and only one of these is similar to RSA.

In Chapter 2 we saw how elliptic curves over a ring looked like and how operations with the points on the curve worked. Now we will take a look at how the KMOV scheme works.

The KMOV scheme works on the points of an elliptic curve $E_n(0, b)$ over the ring $\mathbb{Z}_n$ with some fixed b.

These points form an Abelian group with operations as defined in Section 2.6.2.5 in Chapter 2.

As permutation polynomials KMOV uses

$$\begin{aligned} E(x) &= e \cdot x \\ D(x) &= d \cdot x \end{aligned}$$

Where $x \in E_n(0, b)$ and multiplying this with an integer i means using the $\oplus$ operation i times on x.

The KMOV scheme can certainly be called RSA-like, it works with the group of points on the elliptic curve along with a point at infinity.

The encryption and decryption functions are indeed permutation polynomials as they are bijective. The KMOV scheme is a good example of variation of the RSA scheme, and using elliptic curves in RSA-like encryption.

In 1993 *Nicholas Demytko* presented another RSA variant using elliptic curves, it extended the ideas of the KMOV people and the KMOV scheme can be obtained as a special case of the Demytko scheme.

Demytko generalized the KMOV scheme so that different types of elliptic curves can be used. The scheme is more complex than the KMOV scheme, and it doesn't have the property of defining a bijective encryption and decryption function.

The Demytko scheme is more similar to the Williams scheme in the sense that the decryption key is dependent of the message. It does contain a method for selecting the correct decryption key. And the general Demytko scheme can therefore not be called RSA-like.

It is possible to construct a message independent decryption in the Demytko scheme, but this would be identical to the KMOV scheme.

There have been more attempts to use elliptic curves in RSA-like schemes but many of these have been reduced to the KMOV or Demytko scheme. In Section 4.2.4 we will take a look at the advantages/disadvantages in using elliptic curves in this setting.

4.2.3.3 Dickson polynomials and Lucas sequences

In 1981 *Nöbauer* and *Müller* suggested using Dickson polynomials in public-key cryptography[50]. In 1986 they proposed a scheme based on Dickson polynomials called the Dickson scheme[51].

In 1993 *Peter J. Smith* and *Michael J. J. Lennon* designed a new public-key scheme, named LUC, based on Lucas sequences[72].

Apparently it was not until late in the development they saw the work of *Nöbauer* and *Müller*. The Lucas functions used by *Smith* and *Lennon* in LUC, are identical to the Dickson polynomials used by *Nöbauer* and *Müller* in the Dickson scheme.

A Dickson polynomial is a polynomial over $\mathbb{Z}$ on the form:

$$g_k(a, x) = \sum_{i=0}^{\lfloor k/2 \rfloor} \frac{k}{k-i} \binom{k-i}{i} (-a)^i x^{k-2i} \tag{4.1}$$

Nöbauer and *Müller* showed that Dickson polynomials with certain parameters are permutation polynomials. The LUC scheme utilizes this by using polynomials with $a = 1$.

At first sight LUC has message dependent decryption keys, but it is possible to construct a message independent key at the cost of longer key length.

The LUC scheme works on elements from the ring $\mathbb{Z}_n$. The scheme seems somewhat more computational expensive, but in an implementation a lot of tricks can be applied so the overhead is minimal compared to the original RSA scheme.

4.2.4 Security

A cryptosystem is obviously unusable if it does not provide some kind of security. The concept of security is not easy to define and depends on the usage and environment.

The security of a public-key cryptosystem lies in the "impossibility" for a person to decrypt a message not intended for him/her. In other words there should be no way to discover the plain text without the private key.

In public-key cryptography this is assured by using some well known "hard to solve´´ problem as a base. Some cryptosystems are provably as hard to break as some known problem, and others are supposedly as hard to break as some other known problem. RSA is in the last category because it is believed that breaking RSA is as hard as factoring large integers, but not proved.

Like RSA not being proved to be as hard as factoring integers, neither is factoring integers proved to lie in some hard complexity class like **NP**, which leads to the question:

How secure is RSA then really?

One important test that RSA has passed with flying colors is the test of time. It was the first usable public-key cryptosystem implemented and has for many years been the de facto standard of public-key cryptography. Because it is so widespread it has been the subject of many studies.

RSA has not been broken or compromised in any way, research has found properties that users and implementers should be aware of but no way to recover the plain text of a correctly RSA encrypted cipher text.

Now we will take a look at some of the attacks and weaknesses discovered through the last 25 years of cryptanalysis on the RSA and RSA-like schemes.

The attacks/weaknesses can be categorized into four groups

- *implementation*
  implementations can have security holes for a possible attacker!
- *homomorphic*
  utilizing the homomorphic structure of the scheme!
- *parameters*
  some bad choices of parameters can compromise the cryptosystem!

- *factoring*
  factoring the modulus and thereby breaking the system!

Let's take a look at some of the attacks in these groups.

4.2.4.1 Implementation attacks

When implementing an algorithm on small and simple devices there are some attacks to be aware of.

Implementation attacks use knowledge of the implementation to find weaknesses. In this category are timing attacks and power analysis attacks.

Many of the timing attacks proposed work on the signature scheme of RSA, and since we are not investigating the signature scheme in this paper we will not go into details of these attacks, but give an overview of how they work.

Timing attacks

Timing attacks are attacks based on measuring the time a given operation takes. Most of the attacks proposed on RSA use signature generation to attack the implementation. As we are not concerned with signatures in this paper we will not go into details with these attack but give an overview of them.

Kocher[36] showed that by measuring the time it takes a smart card to perform a RSA operation (decryption or signature), an attacker can discover the private key.

The attack works by the attacker choosing some plain texts which is then signed or encrypted. The corresponding time for a signing/decryption is measured. *Kocher* showed that an attacker can use these pairs to find some of the bits in the decryption key (the private key), starting from the least significant bit.

The obvious way to prevent against a timing attack is making all operations take the same time, this is done by inserting delays assuring that all operations take the same time, but making software running in fixed time is difficult because of the different key sizes.

Defending against timing attacks in this way is far from optimal, especially because of the difficulty in assuring that operations takes fixed time. And attempts of implementing a fixed time implementation often results in very slow software and will also be open for power attacks.

There is another method for defending against timing attacks and that is *blinding*.

Blinding is a technique used for blinding signatures proposed by *Chaum* in his '82 paper [17].

The method works on a smart card doing decryption of the cipher text $\mathcal{C}$ with public key (e, n) and private key (d, n) obtaining the plain text $\mathcal{M}$ in the following way:

1. the smart card chooses some random $r \in \mathbb{Z}_n^*$
2. the smart card then computes $\mathcal{C}' = \mathcal{C} \cdot r^e \mod n$
3. the smart card then computes $\mathcal{M}' = (\mathcal{C}')^d \mod n$

4. the plain text $\mathcal{M}$ is recovered by $\mathcal{M} = \mathcal{M}'/r \mod n$

In this way the attack is not possible because the decryption works on a text unknown to the attacker because of the transformation $\mathcal{C}' = \mathcal{C} \cdot r^e \mod n$.

Power analysis

Instead of measuring the time a given operation takes, one can measure the power used. This is called *power analysis*. Like timing attacks one can figure out the bits of the private key when measuring the power consumption under some given operation.

Power analysis can only be used on units where it is possible to measure power consumption like smart cards and other small units which do not do any other computations at the same time, it is not possible to do power analysis on a PC with a modern operating system at it runs many processes simultaneously. See [35] for details on power analysis and especially *differential power analysis*.

An obvious prevention against power analysis of all sorts is to prevent an attacker in gaining close enough access to the device, this may not be possible to assure for example in smart cards it would also involve an increase in costs.

Another way to protect against an attacker is to add signal noise to the power channel so that an attacker would need infeasibly many samples and not be able to make an attempt to break the code.

To really defend against this attack the algorithms have to be constructed with the target architecture in mind.

Other

There are other measuring techniques, like electromagnetic radiation and super conducting quantum imaging devices. They can all threaten an implementation if the implementer is not aware of the possible "holes" an attacker can use.

All of the measuring attacks can only be applied when the attacker has physical access to the working devices. The easiest way to prevent those attacks is to make sure that it is not possible to gain physical access to the operating machine.

4.2.4.2 Attacking the homomorphic structure of RSA

The permutation polynomial used in the RSA scheme has the following property

$$(m_1 m_2)^e \equiv m_1^e m_2^e \equiv c_1 c_2 \pmod{n}$$

which is called the homomorphic structure of RSA.

This property can be used to mount an *adaptive chosen cipher text attack* which has a lot in common with the blinding technique described previously.

The *adaptive chosen cipher text attack* utilizes the homomorphic property by blinding a cipher text and getting the intended user to decrypt it.

The attacker ***Marvin*** wants to decrypt the cipher text $\mathcal{C}$ intended for ***Alice***. ***Alice*** will not decrypt $\mathcal{C}$ but is willing to decrypt other messages(for example for using as a signature). ***Marvin*** then computes $\mathcal{C}' = \mathcal{C} \cdot x^e \mod n$ where

$x \in \mathbb{Z}_n^*$ is randomly chosen and (e, n) is ***Alice'***s public key. ***Marvin*** then gets ***Alice*** to decrypt $\mathcal{C}'$ returning $\mathcal{M}' = (\mathcal{C}')^d \mod n$. ***Marvin*** can then get the plain text of $\mathcal{C}$ by computing $\mathcal{M} = \mathcal{C}'/x \mod n$, since

$$\mathcal{M}' \equiv (\mathcal{C}')^d \equiv \mathcal{C}^d (x^e)^d \equiv \mathcal{M}x \pmod{n}$$

This attack can be prevented by being more restrictive with what is decrypted and returned, in practice this attack is only possible on very naive users and a badly implemented RSA algorithm.

4.2.4.3 Choosing Parameters

Some of the first attacks on RSA were on systems based on the wrong selection of parameters. There is some properties of the parameters that an user of RSA has to be aware of.

In the key generation phase the user has to make sure that the keys generated satisfy some certain properties. This is because a series of attacks over the years have revealed that certain keys are easier to break than others.

We will now give an overview of attacks on the key structure and give a guide of how to generate keys for RSA that are not vulnerable for these attacks.

Small decryption exponent d

In 1989 *Martin Wiener* showed that using a small decryption exponent was not wise[76]. *Wiener* showed that using continued fractions it was possible to reveal a small decryption exponent. He showed that decryption exponents smaller than $n^{1/4}$, where n is the modulus was unsafe, he also proposed that this bound was not minimal.

In 2000 *Boneh* and *Durfee* changed the bound to $n^{0.292}$ in [9]. They used a similar approach. In the paper they suggested that the precise bound could be upwards $n^{1/2}$.

Because of the *Wiener* attack one should choose $d > n^{1/2}$, where n is the modulus.

The *Wiener* attach has been extended to the elliptic variants of RSA as well as the variants utilizing Dickson polynomials, like LUC, in [54].

Small encryption exponent e

When it is possible to attack the use of a small decryption exponent, it seems obvious that an attack on a small encryption exponent should be possible.

Attacks on a small encryption exponent is not as forwarded as the use of continued fractions in the attack on a small decryption exponent.

An attack on a small encryption exponent seems only to be possible when the attacker can obtain some messages that are linear dependent.

Håstad suggested an attack on a RSA protocol using random data padding[33]. Many RSA protocol implementations use some padding of the data with random data, but when padding some messages in the same way, ie. with the same amount of randomness in the same position, it results in a series of polynomial related messages which can be put through standard linear algebra giving the decryption key.

Coppersmith laid the ground for attacking small encryption exponent by showing that "small" solutions of polynomial congruence can be found in reasonable time[21]. The key elements of the paper can be summarized in the following theorem.

Theorem 4.1 **(Coppersmith)**
Let n be a positive integer, $f(x)$ a monic polynomial of degree d and $X = n^{1/d-\epsilon}$ for some small $\epsilon > 0$. Given n and f then all integers $x_i < X$ satisfying $f(x_i) \equiv 0 \pmod{n}$ can be found in polynomial time.

This theorem can, for example, be used on a series of messages, where only a part of the messages change, like messages starting with the text: "The tasks of today is:...". Then the attacker solves the polynomial congruences of the known text T and some unknown text x in the following way: $(T+x)^e - \mathcal{C} \equiv 0 \pmod{n}$, where (e, n) is the public key and $\mathcal{C}$ is the cipher text.

As seen is attacks on small encryption exponent dependent on protocol implementation, and can be avoided by avoiding the protocol properties mentioned, but for greater security one should use a not to small encryption exponent.

Common parameters

One may never use the same parameters for cryptosystems. The use of the same modulus for two different people with two different encryption exponent enables an attacker to recover the plain text of a message sent to the two people with the same modulus.

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the two cipher texts of the plain text $\mathcal{M}$. The two encryption exponents e_1 and e_2 will be relative prime and the attacker can then find u, v such that $ue_1 + ve_2 \equiv 1 \pmod{n}$, with u, v the attacker can then recover $\mathcal{M}$:

$$\mathcal{C}_1^u \mathcal{C}_2^v \equiv \mathcal{M}^{ue_1+ve_2} \equiv \mathcal{M} \pmod{n}$$

Analogous to this the use of a common encryption exponent will give the attacker the possibility to recover the plain text by using the *The Chinese Remainder Theorem*.

The important thing to notice is that none of these attacks enables the attacker to discover the decryption exponent, he can only discover the original plain text.

Strong Primes

There has been many discussions of using so called "strong primes" in the key generation phase.

"strong prime" is widely accepted as being defined as follows:

Definition 4.3 A prime number p where $p - 1$ and $p + 1$ has a large prime factor are called a *strong prime*.

The argumentation for using strong primes in the key generation phase is to prevent factoring of the modulus with specialized algorithms like *Pollard*'s

$p-1$ algorithm. *Rivest* and *Silverman* discussed the use of "strong primes" is RSA schemes in [65].

They argumented that although non-"strong primes" give rise to more vulnerable composites, the specialized factoring algorithms are inferior to the far more complex and general algorithms like the GNFS.

By choosing large prime numbers for the composite modulus one can avoid the threat imposed by specialized algorithms, there is no reasons not to use strong primes, because they can be constructed without great overhead.

4.2.4.4 Factoring integers

RSA can be "broken" once and for all by the discovery of a fast integer factorization algorithm. If the modulus can be factored then the secret key could easily be computed.

This is the subject of this paper and in Section 5.4 I will discuss the current security status of RSA in regards to integer factorization.

4.3 Discrete Logarithms

In designing public-key cryptosystems two problems dominates the designs, these are *the integer factorization problem* and *the discrete logarithm problem*.

In the previous Section we took an in-depth look at the RSA scheme because it is directly related to integer factorization but solving the discrete logarithm problem is closely related to factoring.

One of the reasons for many newer schemes being based on discrete logarithms rather than factoring is that integer factorization has not proved itself to be a good base for elliptic curve based cryptosystems although useful systems exists like the **KMOV**[37] and **LUC**[72] cryptosystems.

Discrete logarithms have a natural extension into the realm of elliptic curves and hyperelliptic curves, and the **DSA** and **Elliptic ElGamal** have proved to be strong cryptosystems using elliptic curves and discrete logarithms.

4.3.1 Introduction

Originally, the discrete logarithm problem (DLP) was defined for a cyclic subgroup $\mathbb{Z}_p^*$ to the finite field Z_p. We will define it for an arbitrary group G, and afterwards look at the special cases for the elliptic and hyperelliptic variation.

Definition 4.4 **Discrete Logarithm Problem (DLP)**
The *discrete logarithm problem* in the group G is, given some generator α of a cyclic subgroup G^* of G and an element $\beta \in G^*$, find the element $x, 0 \leq x \leq p-2$, such that $\alpha^x = \beta$ ($\alpha x = \beta$ if the groups is written additively).

The most used cryptosystem utilizing the DLP is **ElGamal**, which can be seen in the example box on the next page.

Let us now look at the elliptic curve variant.

Definition 4.5 **Elliptic Curve Discrete Logarithm Problem (ECDLP)**
Let m be the order of $(E/\mathbf{GF}(p^n))$, P an element from $(E/\mathbf{GF}(p^n))$. The *elliptic curve discrete logarithm problem* on the elliptic curve group $(E/\mathbf{GF}(p^n))$ is,

given some element Q in the cyclic subgroup generated by P, find the integer x, such that $xP = Q$.

Definition 4.5 is the base for algorithms like **ECDSA** and **Elliptic ElGamal**.

example: **ElGamal Cryptosystem**

Alice wants to talk secretly with *Bob*.
Setting up: Sometime in the past, Bob has created his keys in the following way:

1. Bob chooses a random large prime p and a generator α of the multiplicative group $\mathbb{Z}_p^*$
2. Bob chooses a random integer a where $1 \leq a \leq p-2$
3. Bob computes $\alpha^a \mod p$
4. The triple $e_B = (p, \alpha, \alpha^a)$ is the public key and $d_B(p, \alpha, a)$ is the private key

Alice obtains Bob's public key from some public key server.
Encryption: Alice wants to encrypt a plain text $\mathcal{M}$ with the cipher $e_B = (e, d, n)$. She starts by choosing a random integer k where $1 \leq k \leq p-2$ And then encrypting the plain text $\mathcal{M}$ into the cipher text $\mathcal{C}$:

$$E_{K_B}(\mathcal{M}) = \mathcal{C} = (\gamma, \delta) = \left(\alpha^k \quad , \quad \mathcal{M} \cdot (\alpha^a)^k \mod p\right)$$

Alice then sends Bob the encrypted message $\mathcal{C}$.
Decryption: Bob then decrypts the cipher text $\mathcal{C} = (\gamma, \delta)$ with the cipher $d_B = (e, d, n)$ in the following manner:

$$D_{K_B}(\mathcal{C}) = \mathcal{M} = \left(\gamma^{-a}\right) \cdot \delta \mod p$$

The reason for choosing an elliptic version rather than the original $\mathbb{Z}_p$ version, is that the ECDLP is harder than DLP, and thereby being able to provide the same security with smaller keys.

The difficulty of solving the discrete logarithm problem in some group, depends on the underlying group.

General Attacks on the DLP

Because the Diffie-Hellman key exchange and other cryptographic schemes based on the difficulty of the discrete logarithm problem is used in many applications, there has always been a lot of research and development in algorithms for solving the discrete logarithm problem.

This has resulted in subexponential algorithms for solving the DLP in cyclic subgroups of finite fields.

The introduction of groups based on points on elliptic curves, resulted in

the elliptic curve discrete logarithm problem which has proven to be harder to solve than its equivalent in cyclic groups of finite fields.

Algorithms for solving the discrete logarithm over an arbitrary group G of order $|G|$ are:

- *Daniel Shanks* developed an algorithm known as the *Baby step - Giant Step* method[68]. The algorithm requires $\mathcal{O}(\sqrt{|G|})$ steps and $\mathcal{O}(\sqrt{|G|})$ workspace. The algorithm works without knowledge of the group order. It can be improved slightly when working over elliptic curves, but the algorithm is far to slow and requires a lot of storage when used on problems over group with the size of order that is used in cryptography.

- *John M. Pollard* is one of the most productive people in the area of factorization and solving the discrete logarithm problem. He has developed many algorithms for factoring and solving discrete logarithms. He is the developer of the *Pollard's ρ method* of solving the discrete logarithm problem and the generalized *Pollard's λ method* both presented in [58]. Both algorithms are *Monte-Carlo* methods, meaning that they use some random walk which can yield a solution, in case it does not the algorithm takes another walk. The algorithms uses roughly the same time as *Shanks*'s algorithm but they do not use as much space. This is one of the methods used to solve instances of ECDLP, and of 2003 the largest ECDLP instance solved with Pollard's ρ algorithm is for an elliptic curve over a 109 bit field. The Pollard ρ method for factoring is described in the next Chapter.

- In 1978 *Pohlig* and *Hellman* proposed an algorithm[55] after the success and acceptance of the Diffie-Hellman key exchange protocol. The algorithm is fast but only when the group order has small prime divisors. The algorithm only works when the group order is known in advance. To avoid an attack with the Pohlig-Hellman algorithm one should make sure that the group order has large prime factors or is of prime order. Pohlig and Hellman has an analogues method for factoring.

Solving DLP over a Multiplicative Group

Solving the discrete logarithm problem of a multiplicative group of a finite field, can be done in subexponential time by the algorithm called *index calculus*.

The index calculus is a method based upon the ideas of *Western* and *Miller* [75]. The algorithm was developed independently by *Adleman*[3], *Merkle*[44] and *Pollard*[58]. The algorithm exists in many versions, some generalized to special fields like the very fast version by *Coppersmith*[19, 20] that is specialized to the fields $\mathbf{GF}(2^n)$.

The index calculus algorithms are identically to the modern general purpose factoring algorithms which is presented in the next Chapter. It consists of the same 3 basic steps.

The development of algorithms for solving discrete logarithms follows the development of factoring algorithms and a fast method for factoring integers

would likewise result in a fast way to compute discrete logarithms.

Attacking the ECDLP

The index calculus was one of the concerns which made Koblitz and Miller propose to use groups based on elliptic curves, because the index calculus algorithms do not work on these groups. Although there exist index calculus algorithms adapted for elliptic curve groups, they do not have the same subexponential running time.

The general algorithms presented above can of course also be used on elliptic curve based groups.

The only weaknesses in groups based on elliptic curves comes from badly chosen curves.

In 1993 *Menezes*, *Okamoto* and *Vanstone*[1] presented an attack on a special class of curves. The attack, named MOV after the authors, reduces the discrete logarithm over an elliptic curve to the discrete logarithm over a finite field. The MOV attack only works on supersingular curves; the attack can therefore be avoided by not using supersingular curves.

Another class of elliptic curves have also proved to be weak for cryptographic purposes, namely the class of anomalous elliptic curves. A curve E is called anomalous if $|E/\mathbf{GF}(q)| = q$. The first attack on anomalous curves was proposed by *Nigel Smart*[71], the attack is quite complex and uses p-adic numbers lifted to another group. We do not go into further details with the attacks on anomalous curves or the MOV attack because they are easy to circumvent by carefully choosing the underlying curve[31].

CHAPTER 5

Factorization Methods

"The obvious mathematical breakthrough, would be development of an easy way to factor large prime numbers."

- Bill Gates(1955 -)

In the previous Chapter we got a view of the history and development of integer factorization and various algorithms were mentioned. It is some of those I will take a closer look at now and describe the algorithms leading to the number field sieve.

Algorithms for integer factorization can be split into two groups, the *special* and the *general* algorithms. The special algorithms are those targeted at a special class of numbers, for example if the number has one or more small factors. The general algorithms are those not targeted at a special class of numbers, i.e. it takes the same time to split a 100 bit number into a 1 and a 99 bit factor as it takes to split it into two 50 bit factors.

All algorithms described depends on that the number n to be factored is a composite, so it should be tested for primality before attempting to factor it.

Some of the algorithms have *subexponential* running time and this is often defined with the $L_n[\alpha, c]$ expression.

Definition 5.1 **Subexponential time algorithm**
Let A be an algorithm whose input is an integer n or a small set of integers modulo n, that is the input size is $\mathcal{O}(\log_2 n)$. If A has the running time:

$$L_n[\alpha, c] = \mathcal{O}\left(e^{(c+\mathcal{O}(1))(\log n)^{\alpha}(\log\log n)^{1-\alpha}}\right) \tag{5.1}$$

where c is a constant and α is a constant satisfying $0 < \alpha < 1$. Then A is a *subexponential algorithm*.

5.1 Special Algorithms

The special algorithms are not interesting for RSA type numbers, because they are constructed to be hard integers and do not have any easy structure, so special algorithms would either run "forever" or fail. This means that they

are useless for our purpose, but for completeness I will give an idea of how the most known special methods works some of them uses ideas that are in use in general algorithms as well.

5.1.1 Trial division

Trial division is a fast method for small composites, and as its name indicates it trial divides possible factors to see if the remainder is zero.

It does not fail for hard composites, it just takes a long time.

There is one major decision to make before implementing a trial division algorithm, should we use primes or not ?

This means if we should use a list of primes for the divisions or we should generate pseudo-primes on-the-fly. It takes a lot of time and storage to create and store a list of primes, instead we can use the following numbers instead

$$2, 3, 6k \pm 1 \qquad k \in \mathbb{Z}, k > 0$$

This sequence covers all the primes[62], but also includes a few composites which means that we may do a few more divisions than needed, but that is a fair price for avoiding to create and store a list of primes.

There is different methods to generate the sequence described above, we can start by 5 and then alternately add 2 and 4. I will use a variation of this in the algorithm below. The algorithm described returns the first factor p found, for a complete factorization it should be run again on n/p.

algorithm: **Trial Division**

input: composite integer n.
output: a nontrivial factor p of n.

1. **if** $n \equiv 0 \mod 2$ **return** $p = 2$
2. **if** $n \equiv 0 \mod 3$ **return** $p = 3$
3. **set** $p = 3$
   set $b = 2$
4. **while** $p < \sqrt{n}$
   (a) **set** $p = p + b$
   (b) **if** $n \mod p = 0$ **return** p
   (c) **set** $b = 6 - b$

Trial division is useless for composites with only large factors, but for smaller composites it is definitely a usable algorithm and it is a good exercise to implement it.

The time complexity of trial division is $\mathcal{O}(\sqrt{n})$, where n is the number to be factored.

5.1.2 Pollard's $p-1$ method

In 1975 *John M. Pollard* proposed a new algorithm for factoring integers[56]. Which is based on Fermat's Little Theorem (Theorem 2.4). It is a specialized method for integers having some factor p where $p-1$ is B-smooth for some relatively small B.

An attempt to use it on non-B-*smooth* integers will result in failure.

From Fermat we have

$$a^{p-1} \equiv 1 \pmod{p} \quad \text{, so} \quad a^{k(p-1)} \equiv 1 \pmod{p}$$

and therefore

$$a^{k(p-1)} - 1 \equiv 0 \pmod{p}$$

for any integer a and prime p, where $p \nmid a$.

The algorithm can be described as

algorithm: **Pollard's** $p-1$

input: composite integer n.
output: a nontrivial factor p of n.

1. **select** smoothness bound B
2. **select** integer a
3. **for each** prime q below the smoothness bound B
   (a) **compute** $s = \lfloor \frac{\ln(n)}{\ln(q)} \rfloor$
   (b) **set** $a = a^{q^s} \mod n$
4. **set** $p = \gcd(a-1, n)$
5. **if** $1 < p < n$ then p is a nontrivial factor of n
6. **else select** new a and **goto** 3

The *Pollard* $p-1$ method is not that effective unless one uses some smart strategy for choosing a's.

Conjecture 5.1 **Complexity of the** $(p-1)$**-method**[43]
Let p be a factor of n and p is B-smooth, then the $(p-1)$-algorithm has expected running time

$$\mathcal{O}\left(B\frac{\ln n}{\ln B}\right)$$

Williams developed the method into a $p+1$ factoring algorithm utilizing Lucas sequences[79], which works well if the composite has a factor p where $p+1$ is smooth.

5.1.3 Pollard's ρ method

In 1975 *Pollard* proposed a Monte Carlo method for factoring integers[57].

The *Pollard* ρ method[1] of factoring showed some very interesting ideas usable for future factoring algorithms. Although it shares some similarities with the $p-1$ method, it contains a more constructive way of choosing the trial divisors. It is specialized for composite integers with small factors.

The idea of the ρ-method can be described in these three steps:

1. Compute a sequence of integers $a_0, \ldots, a_m$ that are periodically recurrent modulo p.
2. Find the period of the sequence, i.e. find i and j such that $a_i \equiv a_j \mod p$
3. Identify the factor p

Here is the algorithm

algorithm: **Pollard's** ρ

input: composite integer n.
output: a nontrivial factor p of n.

1. **set** $a = b = 2$,$c = 1$
2. **set** $f_c(x) = x^2 + c \pmod n$
3. **set** $a = f_c(a)$ and $b = f_c(f_c(b))$
4. **set** $d = \gcd(a - b, n)$
5. **if** $1 < d < n$ **return** the nontrivial factor $p = a - b$
6. **if** $d = 1$ **goto** 3
7. **if** $d = n$ **set** $c = c + 1$ and **goto** 2

In 1980 Richard P. Brent refined the algorithm[11] and it is his version that is commonly used. It is about 25% faster than the original Pollard version, and it has been applied to some large composites like the 8th Fermat number[2] in 1980, it took 2 hours on a UNIVAC.

The difference from the original version is in the way it detects cycles in the sequence.

[1] It gets the ρ name because the iteration of the elements looks like a circle with a tail when sketched.

[2] The nth Fermat number is defined as $F_n = 2^{2^n} + 1$, and is named after Pierre de Fermat that postulated that F_n is prime for all n which it is not. So far only the first 5 Fermat numbers $(F_0, F_1, F_2, F_3, F_4)$ are primes but it is not proved that they are the only Fermat primes. The Fermat numbers $F_7, F_8, F_9, F_{10}, F_{11}$ have shown to be hard composites and a worthy opponent for modern day factoring algorithms.

algorithm: **Brent-Pollard's** ρ

input: composite integer n.
output: a nontrivial factor p of n.

1. **set** $r = q = p = 1$,$c = 1$
2. **choose** random integer y
3. **choose** $f_c(x) = x^2 + c \pmod n$
4. **while** $p = 1$
   (a) **set** $x = y$
   (b) **for** $i = 1$ to r
      i. **set** $y = f(y)$
   (c) **set** $k = 0$
   (d) **while** $k \neq r$ **and** $p = 1$
      i. **set** $y_s = y$
      ii. **for** $i = 1$ **to** $\min(m, r - k)$
         A. **set** $y = f(y)$
            set $q = (q|x - y|) \mod n$
      iii. **set** $p = \gcd(q, n)$
         set $k = k + m$
   (e) **set** $r = 2r$
5. **if** $g = n$
   (a) **while** $p = 1$
      i. **set** $y_s = f(y_s)$
         set $p = \gcd(x - y_s, n)$
6. **if** $p = n$ **set** $c = c + 1$ and **goto** 1
   else return p

Conjecture 5.2 **Complexity of the ρ-method**[43]
Let p be a factor of n and $p = \mathcal{O}(\sqrt{n})$, then the ρ-algorithm has expected running time

$$\mathcal{O}(\sqrt{p}) = \mathcal{O}(n^{1/4})$$

5.1.4 Elliptic Curve Method (ECM)

In 1985 *Lenstra* suggested using elliptic curves for factoring composites[30], shortly thereafter Brent refined the ideas of Lenstra in [10]. It is the fastest of the special algorithms.

The ECM method is a lot like *Pollard*'s $p-1$ method, actually it is the same, but Lenstra generalized it and applied it to the group of points on an elliptic curve.

Pollard worked in the multiplicative group of the finite field K, and Lenstra uses a group based on the points of an elliptic curve, as described in Section 2.6.

The group operations of an elliptic curve group is more expensive, but it gives the opportunity to use several groups at the same time. ECM have been used in some big factorizations, see [12] for some results of the ECM.

I will give the algorithm here; for details of arithmetic with elliptic curves and notation used see Section 2.6.

algorithm: **Lenstra's ECM**

input: composite integer n with $\gcd(n, 6) = 1$.
output: a nontrivial factor p of n.

1. **select** smoothness bound B
2. **choose random** pair (E, P), where E is an elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{Z}/n\mathbb{Z}$ and $P(x, y) \in E(\mathbb{Z}/n\mathbb{Z})$ is a point on E (see Section 2.6)
3. **select** integer a to be a product of some or all of the primes below B, e.g. $a = B!$
4. **compute** the point $c = aP \in E(\mathbb{Z}/\mathbb{Z})$, which is done by repeated additions as described here:

   To compute $P_3(x_3, y_3) = P_1(x_1, y_1) + P_2(x_2, y_2) \mod n$

   $$(x_3, y_3) = (\lambda^2 - x_1 - x_2 \mod n, \lambda(x_1 - x_3) - y_1 \mod n)$$

   where

   $$\lambda = \begin{cases} \frac{d_1}{d_2} = \frac{3x_1^2+a}{2y_1} \mod n & \text{if } P_1 = P_2 \\ \frac{d_1}{d_2} = \frac{y_1-y_2}{x_1-x_2} \mod n & \text{otherwise} \end{cases}$$

5. **if** $c = \mathcal{O}$ **set** $p = \gcd(d_2, n)$
   else goto 2
6. **if** $1 < p < n$ then p is a nontrivial factor of n
   else goto 2

As with all other of the special algorithms one can add another phase based on some continuation principle, i.e. what to do if it does not work on the first run, and the ECM have been developed and different continuation strategies have been proposed, for a summary of continuation strategies see [12].

The time complexity of the ECM algorithm is $L_n[\frac{1}{2}, 1]$ based on heuristic estimates see [43].

5.2 General Algorithms

In the Section above I described some of the special methods for factoring, but because they are "special" they are not usable for the purpose of factoring RSA composites, because the primes used are chosen to be hard as described in Section 4.2.4.3.

The only algorithms usable for RSA composites are the general ones, and for the size of composites used in RSA today it is only the number field sieve that is applicable.

All of the modern general purpose factoring algorithms are based on the same principle and idea, namely congruent squares.

5.2.1 Congruent Squares

Congruent squares is a pseudonym for Legendre's congruence

Definition 5.2 **Legendre's Congruence**

$$x^2 \equiv y^2 \mod n \quad , 0 \leq x \leq y \leq n, x \neq y, x + y \neq n$$

If we have integers x and y which satisfy Legendre's Congruence then $\gcd(n, x - y)$ and $\gcd(n, x + y)$ are possibly nontrivial factors of n.

If Legendre's Congruence is satisfied we have

$$x^2 \equiv y^2 \mod n$$

This implies

$$\begin{aligned} x^2 \equiv y^2 \mod n &\Leftrightarrow n|x^2 - y^2 \\ &\Leftrightarrow n|(x - y)(x + y) \end{aligned}$$

If n has the prime factorization $n = pq$, classical number theory gives

$$\begin{aligned} x^2 \equiv y^2 \mod pq &\Leftrightarrow pq|x^2 - y^2 \\ &\Leftrightarrow pq|(x - y)(x + y) \\ &\Leftrightarrow p|(x - y) \text{ or } p|(x + y) \\ &\Leftrightarrow q|(x - y) \text{ or } q|(x + y) \end{aligned}$$

This implies that we can find p or/and q by computing $\gcd(n, x \pm y)$. We do not always end up with the nontrivial divisors, below is a table of the different results from congruent squares with $n = pq$.

$p\|(x-y)$	$p\|(x+y)$	$q\|(x-y)$	$q\|(x+y)$	$\gcd(n, x-y)$	$\gcd(n, x+y)$	*nontrivial factor?*
yes	no	yes	no	n	0	no
yes	no	no	yes	p	q	yes
yes	no	yes	yes	n	q	yes
no	yes	yes	no	q	p	yes
no	yes	no	yes	0	n	no
no	yes	yes	yes	q	n	yes
yes	yes	yes	no	n	p	yes
yes	yes	no	yes	p	n	yes
yes	yes	yes	yes	n	n	no

Assuming that the combinations in the table are equally likely to happen, we have a probability of 2/3 of getting a nontrivial factor of n. This means that we probably only need to have a small number of pairs (x, y) that satisfy Legendre's Congruence in order to factor n.

Modern factoring methods all consists of the 3 basic steps

1. Identify a set of relations that are smooth over some factor base.

2. Solve the system of linear equations and find the relations that yields squares.

3. Compute the $\gcd$ of the composite and the squares found.

These are the steps that the modern factoring methods uses, the difference lies only in how they find the pairs of integers that satisfy Legendre's Congruence.

5.2.2 Continued Fractions (CFRAC)

Legendre himself had a method for factoring using the idea of congruent squares, but it was Morrison and Brillhart that gave us the first modern method[48]. For many years it was the fastest algorithm for large composites, it was first surpassed when the quadratic sieve emerged and later the number field sieve.

The continued fractions method (CFRAC) is interesting because it uses a completely different method for finding the desired squares than any of the later methods.

CFRAC as it names indicates use continued fractions. A continued fraction is a number representation and is used to represent real numbers with integers.

Any real number x can be represented by a general **continued fraction** on the form

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \dots}}}$$

where $a_i, b_i \in \mathbb{Z}$. If $b_i = 1$ it is called a *simple continued fraction*.

A simple continued fraction is often written on a special form

$$\begin{aligned} x &= a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \ldots}}} \\ x &= [a_0, a_1, a_2, a_3, \ldots] \end{aligned}$$

Here is an algorithm for finding the continued fraction of a real number.

algorithm: **Continued Fraction representation**

input: real number x.
output: simple continued fraction representation $[q_0, q_1, q_2, \ldots]$ of x.

$$\begin{array}{rclrcl} & & & x_0 & = & x \\ q_0 & = & \lfloor x_0 \rfloor & x_1 & = & \frac{1}{x_0 - q_0} \\ q_1 & = & \lfloor x_1 \rfloor & x_2 & = & \frac{1}{x_1 - q_1} \\ & \vdots & & & \vdots & \\ q_i & = & \lfloor x_i \rfloor & x_{i+1} & = & \frac{1}{x_i - q_i} \\ & \vdots & & & \vdots & \end{array}$$

Theorem 5.1 The square root $\sqrt{n}$ of a square free integer n has a periodic continued fraction of the form

$$\sqrt{n} = [a_0, \overline{a_1, \ldots, a_n, 2a_0}]$$

Furthermore we have $0 \leq a_i \leq 2\sqrt{n}$

I need one more tool before I can describe the CFRAC algorithm.

Definition 5.3 The convergents P_n/Q_n of a simple continued fraction

$$[q_0, q_1, q_2, q_3, \ldots]$$

are defined as

$$\begin{aligned} \frac{P_0}{Q_0} &= \frac{q_0}{1} \\ \frac{P_1}{Q_1} &= \frac{q_0 q_1 + 1}{q_1} \\ &\vdots \\ \frac{P_i}{Q_i} &= \frac{q_i P_{i-1} + P_{i-2}}{q_i Q_{i-1} + Q_{i-2}} \quad , i \geq 2 \end{aligned}$$

CFRAC uses the convergents of the continued fraction of $\sqrt{n}$ and the ones that are smooth over the factor base are kept as relations. The relations are put through a linear algebra step and the ones that yield a square are then used for the $\gcd$ step.

Here is an outline of the CFRAC algorithm, for further details consult [48].

algorithm: **CFRAC**

input: composite integer n.
output: a nontrivial factor p of n.

Step 1: *(Setting up factor base)*
Construct the *factor base* fb with all primes below some chosen limit.

Step 2: *(Finding relations)*
Compute the continued fraction expansion $[q_0, \overline{q_1, q_2, \ldots, q_r}]$ of $\sqrt{n}$
For each congruent P_i/Q_i compute the corresponding integer $W = P_i^2 - Q_i^2 n$, and if it is fb-smooth, store a vector of its prime factorization as a relation.
When we have obtained more relations than elements in fb we can proceed to the next step.

Step 3: *(finding squares)*
From the relations we can use Gaussian Elimination over $\mathbf{GF}(2)$ and find x and y that satisfy Legendre's Congruence. (I will go into details with this in the next Chapter)

Step 4: *(gcd)*
p can then be found (with a probability of 2/3 as described in Section 5.2.1) by $p = \gcd(n, x \pm y)$.

The period of the continued fraction of $\sqrt{n}$ can be so short that enough relations cannot be found. A way around this is to use $\sqrt{kn}$ instead, see more in [60].

Conjecture 5.3 **Complexity of the CFRAC-method**[62]
CFRAC has heuristic running time

$$\mathcal{O}\left(n^{\sqrt{1.5\frac{\ln\ln n}{\ln n}}}\right)$$

Where n is the integer to be factored.

5.2.3 Quadratic Sieve

One of the shortcomings of the CFRAC algorithm is that a lot of useless divisions are performed. CFRAC does not have a way of assuring that the divisions that are done are on numbers that are smooth and end up as a relation.

Carl Pomerance presented in [60] a new factoring algorithm called the *Quadratic Sieve* (QS) which avoided a large part of the useless divisions done in CFRAC.

Like CFRAC, QS is based on the idea of congruent squares, but it uses a complete different approach to finding the squares.

Like its name indicates it uses a quadratic polynomial and quadratic residues. We start by defining the polynomial

$$Q(x) = \left(x + \lfloor\sqrt{n}\rfloor\right)^2 = \tilde{x}^2 - n \tag{5.2}$$

We then compute $Q(x_1), Q(x_2), \ldots, Q(x_k)$ for some x_i defined later. From the evaluations of $Q(x_i)$ we want a subset $Q(x_{i_1})Q(x_{i_2})\cdots Q(x_{i_r})$ which is a square $y^2 \in \mathbb{Z}$.

From 5.2 we have $Q(x) \equiv \tilde{x}^2 \pmod{n}$ so we get the wanted congruence

$$Q(x_{i_1})Q(x_{i_2})\cdots Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}\cdots x_{i_r})^2 \mod n \tag{5.3}$$

And a nontrivial factor can (probably) be found as described in Section 5.2.1.

What remains unanswered is how we find the $Q(x_i)$'s that gives a square. Like CFRAC it gather relations that are smooth over a factor base and then uses linear algebra to find the relations that gives a square.

The factor base in QS consists the primes p_i below some bound B and where n is a quadratic residue $\pmod{p_i}$, i.e. the Legendre symbol

$$\left(\frac{n}{p_i}\right) = 1$$

The relations are then found by sieving over some interval of x_i's and the x_i that have a $Q(x_i)$ which is smooth over the factor base are kept as a relation.

Here is an overview of the algorithm, for further details consult [60].

algorithm: **QS**

---

input: composite integer n.
output: a nontrivial factor p of n.

Step 1: *(Setting up factor base)*
Construct the *factor base* fb with all primes p_i below some chosen limit B, and which has Legendre symbol $(\frac{n}{p_i}) = 1$.

Step 2: *(Finding relations)*
Compute the polynomial $Q(x)$ and define a sieving interval $[-M; M]$.
Compute $Q(x_i)$ for $x_i \in [-M; M]$
If $Q(x_i]$ is smooth over the factor base, store x_i as a relation.
Go on to the next step when there are more relations stored than the number of elements in the factor base.

Step 3: *(finding squares)*
From the relations we can use Gaussian Elimination over $\mathbf{GF}(2)$ and find x and y that satisfy Legendre's Congruence as in (5.3). I will go into details with this in the next Chapter.

Step 4: *(gcd)*
p can then be found (with a probability of 2/3 as described previously) by $p = \gcd(n, x \pm y)$.

The quadratic sieve was for many years the fastest factoring algorithm for large composites but has been superceeded by the number field sieve, but for integers in the 60-110 digit range it still seems to be the fastest method.

There has been various improvements to the original method, including using multiple polynomials in the *multi polynomial quadratic sieve* (MPQS) and different sieving methods which we will see more on in the next Chapter.

The time complexity of the QS algorithm is the same as that of the ECM: $(L_n[\frac{1}{2}, 1])$, but the operations in the ECM are more expensive so for composites with only large prime factors is the QS method faster than ECM[43].

5.2.4 Number Field Sieve

In 1988 John Pollard circulated the famous letter that presented the idea of the *number field sieve* (NFS).

In this thesis when I say "number field sieve" I am actually talking about the *general number field sieve*. The original number field sieve is denoted *special number field sieve* (SNFS) today.

The number field sieve is the fastest general algorithm known today, but because of its complexity and overhead it is only faster than the QS for number larger than 110-120 digits.

The number field sieve took the quadratic sieve to another level by using algebraic number fields. Going in deep with the theory behind the number field sieve deserves several papers on its own and for the "dirty" details, see for example [13], [39] and [32].

I will describe the fundamental idea behind the algorithm and try to convince you that it works. The next Chapter describes the algorithm in details.

The number field sieve is a lot like the quadratic sieve described in 5.2.3, but it differs on one major point: the field it works in.

A number field is a common word for all subfields of $\mathbb{C}$ and one way to construct a number field K is to take an irreducible polynomial f of degree d with a root $\alpha \in \mathbb{C}$ and $K = \mathbb{Q}[\alpha]$ is a degree d extension field.

A number ring is a subring of a number field; the number field sieve uses the number ring $\mathbb{Z}[\alpha] = \mathbb{Z}[x]/f\mathbb{Z}[x]$. A familiar example of a number ring is the subring $\mathbb{Z}[i]$ of *Gaussian integers* which is a subring of the number field $\mathbb{Q}[i]$ derived from the polynomial $f(x) = x^2 + 1$ with $\alpha = i$.

The number field $\mathbb{Q}[\alpha]$ consists of elements on the form $\sum_{j=0}^{d-1} q_j \alpha^j$ with $q_j \in \mathbb{Q}$, the number ring $\mathbb{Z}[\alpha] = \mathbb{Z}[x]/f\mathbb{Z}[x]$ contains elements on the form $\sum_{j=0}^{d-1} s_j \alpha^j$, with $s_j \in \mathbb{Z}$. We need an embedding into $\mathbb{Z}$ for the number ring $\mathbb{Z}[\alpha]$ before it is usable for factoring, but fortunately we have the ring homomorphism ϕ

Theorem 5.2 Given a polynomial $f(x) \in \mathbb{Z}[x]$, a root $\alpha \in \mathbb{C}$ and $m \in \mathbb{Z}/n\mathbb{Z}$ such that $f(m) \cong 0 \pmod{n}$, there exists a unique mapping

$$\phi : \mathbb{Z}[\alpha] \mapsto \mathbb{Z}/n\mathbb{Z}$$

satisfying

$$\begin{aligned} \phi(1) &\equiv 1 \mod n \\ \phi(\alpha) &\equiv m \mod n \\ \phi(ab) &= \phi(a)\phi(b) \\ \phi(a+b) &= \phi(a) + \phi(b) \end{aligned}$$

for all $a, b \in \mathbb{Z}[\alpha]$

The ring homomorphism ϕ leads to the desired congruent squares. If we can find a non-empty set S with the following properties

$$\begin{aligned} y^2 &= \prod_{(a,b)\in S} (a + bm) \quad : \quad y^2 \in \mathbb{Z} \\ \beta^2 &= \prod_{(a,b)\in S} (a + b\alpha) \quad : \quad \beta^2 \in \mathbb{Z}[\alpha] \end{aligned}$$

we get the congruence

$$\begin{aligned} \phi(\beta)^2 &= \phi(\beta)\phi(\beta) \\ &= \phi(\beta^2) \\ &= \phi\left(\prod_{(a,b)\in S} (a + b\alpha)\right) \\ &= \prod_{(a,b)\in S} \phi((a + b\alpha)) \\ &= \prod_{(a,b)\in S} (a + bm) \\ &= y^2 \end{aligned}$$

As I showed in Section 5.2.1, there is a probability of $2/3$ that we get non-trivial divisors by computing $\gcd(n, \beta \pm y)$.

One of the major questions remaining is: how do we find the set S ?

Definition 5.4 **RFB-Smooth**
The pair (a, b) is said to be **RFB**-smooth if its rational norm

$$N(a, b) = a + bm$$

factors completely into elements from **RFB**.

Definition 5.5 **AFB-Smooth**
The pair (a, b) is said to be **AFB**-smooth if its algebraic norm

$$\mathcal{N}(a, b) = (-b)^{\deg(f)} f\left(-\frac{a}{b}\right)$$

factors completely into elements from **AFB**.

S is the set of (a, b)'s that are both **RFB**-smooth and **AFB**-smooth and is found by sieving as described in Section 6.2.3.

This only works if $\mathbb{Z}[\alpha]$ is a *unique factorization domain*(UFD), but this is not always guaranteed.

5.2.4.1 Working Around Wrong Assumptions

The assumption that $\mathbb{Z}[\alpha]$ is a UFD makes the theory work, but in practice it is not always a UFD, and this leads to errors, but there is a way around these.

What can happen is that we can have an extension field $\mathbb{Q}[\alpha]$ which contains an algebraic integer that is not in $\mathbb{Z}[\alpha]$. The problem that arises is that we can end up with a product of elements from S that is not in $\mathbb{Z}[\alpha]$ and this means that we cannot use the homomorphism ϕ.

It is easier to allow the product $S(x)$ to be a product in $\mathbb{Q}[\alpha]$, because by multiplying it with $f'(x)^2$ we are guaranteed that it is in $\mathbb{Z}[\alpha]$, and we can multiply the rational product with $f'(m)^2$ to obtain the wanted congruent squares

$$\begin{aligned}
\phi(\beta)^2 &= \phi(\beta)\phi(\beta) \\
&= \phi(\beta^2) \\
&= \phi\left(f'(\alpha)^2 \prod_{(a,b)\in S}(a+b\alpha)\right) \\
&= \phi(f'(\alpha)^2) \prod_{(a,b)\in S}\phi((a+b\alpha)) \\
&= f'(m)^2 \prod_{(a,b)\in S}(a+bm) \\
&= y^2
\end{aligned}$$

see [14] for further details.

One other consequence of sieving the way explained and assuming that $\mathbb{Z}[\alpha]$ is a UFD is that we are not guaranteed that S actually is a square in $\mathbb{Z}[\alpha]$.

The **AFB** contains first degree prime ideals of $\mathbb{Z}[\alpha]$ and although the sieving assures us that the resulting elements are **AFB**-smooth, the product is not necessarily a perfect square in $\mathbb{Z}[\alpha]$. To get past that obstacle the **QCB** is used.

Quadratic characters are introduced in the linear algebra step and they give a higher probability of a product of elements from S being a perfect square in $\mathbb{Z}[\alpha]$. The higher the number of quadratic characters used the more likely we are to arrive at a perfect square in $\mathbb{Z}[\alpha]$. See [7] for a discussion of the usage of quadratic characters where it is implied that 50-100 quadratic chracters should suffice for any factorization.

Actually, there are a few more theoretic obstacles in assuming that $\mathbb{Z}[\alpha]$ is UFD and the use of first degree prime ideals, but they are all solved by the use of quadratic characters.

The reader should at this point be aware that one do not necessarily get the wanted result from using the number field sieve, but using it correctly with the right parameters has a very high probability of succeeding as described in 5.2.1.

The next Chapter will take you through the algorithm step by step and will also answer some of the unanswered questions from the Sections on CFRAC and QS.

Conjecture 5.4 **Complexity of NFS**[62]
Under some reasonable heuristic assumptions, the NFS method factors n in

$$L_n[\frac{1}{3}, c] = \mathcal{O}\left(e^{c+\mathcal{O}(1))\sqrt[3]{\log n}\sqrt[3]{(\log\log n)^2}}\right)$$

where $c = \sqrt[3]{\frac{32}{9}}$ for the SNFS and $c = \sqrt[3]{\frac{64}{9}}$ for the GNFS.

5.3 Factoring Strategy

If we were to factor a random integer we would not start by applying a general purpose algorithm. We should start by applying the ECM to reveal small factors and if it is not capable to factor it in reasonable time one should roll in the heavy artillery and apply the QS (for $n < 120$ digits) or else the number field sieve.

If we know that n is a RSA modulus then there is no reason to use anything other than the number field sieve.

5.4 The Future

I have discussed various factoring methods in this Chapter including the fastest general purpose factoring algorithm known - which I will describe in details in the next 2 Chapters. I have also discussed various attacks on RSA besides factoring in Chapter 4.

But where does that leave the security of public key cryptography today ?

5.4.1 Security of Public Key Cryptosystems

Current implementations of GNFS has successfully factored a 512 bit RSA modulus, and recommendations for key sizes has been 1024 bit or more for some years, but a 1024 bit modulus is factorable with high probability in the near future.

Does this mean that individuals that use home banking from their PC need to worry ?

It is correct that it is possible to factor a 512 bit RSA modulus, but the computations involved are extensive and it takes half a year on 1000+ machines. Common people should not worry about their secure communications, but I think that the we all need to put some serious thinking into the usage of digital signatures.

It is the idea that digital signatures should be used instead of the handwritten signature in almost any situation in the future. This means that a digital signature should be unforgeable for many years, ideally for the duration of a human lifetime. One thing we can say for sure is that we do not know the status of factoring after a human lifetime - even public key cryptography has only been in the public domain for about 30 years.

All known attacks against the RSA cryptosystem can be circumvented by using the system correctly and carefully choosing the underlying primes. Resulting in factoring of the modulus as the only true weakness of the scheme.

This means that the security of RSA relies solely on the ability of factoring large integers.

We can not say with certainty that choosing a 4096 bit key for signatures would give enough security for x years. The past has shown a slow increase in the bits of factorable numbers, so choosing 2048+ bit key sizes should give some margin of security.

5.4.2 Factoring in The Future

What will the future of integer factorization bring ? This is impossible to predict but if we look at current development there is some research in *Factorization Circuits*. Research has been done for many years on designing circuits for factoring and the idea seems to be a hot topic. The main problem is the price of such a circuit compared to mainstream PC architectures, D. J. Bernstein has hinted in a recent conference that he is going to present a new circuit design with ECM as smoothness test on sieving elements in a number field sieve implementation, so it is still a hot topic.

Computational power is increasing day for day and although there is far to a PC being able to factor a 512 bit modulus in reasonable time there is still the threat that an organization can build a large farm of cheap PC's and start a code breaking service for the highest bidder. This would probably not be used to invade the privacy of an individual but to obtain financial gains or in a war or terror situation.

The last 10+ years, advances in factorization algorithms have all been based on using smoothness and factor bases and it is very likely that one have to choose another strategy to find even faster algorithms, hopefully a polynomial time algorithm.

The polynomial time algorithm for testing primality[4] used some elementary number theory and is quite different from other primality testing algorithms known. This could be what is needed for integer factorization unless it is found to be NP-complete.

CHAPTER 6

The Number Field Sieve

"God does arithmetic."

- Carl Friedrich Gauss(1777-1855)

The reader is now aware of the various types of algorithms for factoring, and know that the fastest algorithms are based on sieving over a well-defined domain; actually one can refine it to say that the fastest algorithms calculates congruences and solve systems of linear equations.

The number field sieve variants have the smallest time-complexity compared to other factoring algorithms, and have also proved themselves to be the best algorithms to use in real life for large composites.

The number field sieve shares many similarities with the quadratic sieve described in Section 5.2.3, and can be seen as an expansion of the QS to utilize polynomials of degree > 2.

The two main variants are the *special number field sieve* (SNFS) and the *general number field sieve* (GNFS). The SNFS works on a special type of composites, namely integers on the form: $r^e - s$, for small integers r, s and integer e and the GNFS works on all types of composites. The difference between SNFS and GNFS is in the polynomial selection part of the algorithm, where the special numbers which SNFS can be applied to, make a special class of polynomials especially attractive and the work in the square root step is also more complex for the GNFS.

6.1 Overview of the GNFS Algorithm

The algorithm is complex, and to understand it in full details it will help to start with a rough outline. In this Section I will give you an overview of the GNFS algorithm, in the next Section I will take a look at the implementation issues by writing algorithms for the different steps and then I will describe the properties that make the algorithm work.

The algorithm shares a lot of simularities with other sieving based algorithms mentioned in Chapter 5. The algorithm consist of 4 main steps which are mutual dependent and can not be made in parallel, but some of the steps can be parallel internally. I have chosen to describe it in 5 steps to make the

flow of the algorithm more digestable, but behind each step is a lot of calculations and different algorithms in interaction.

algorithm: **GNFS (rough outline)**

input: composite integer n.
output: a nontrivial factor p of n.

Step 1: *(Polynomial selection)*
Find an irreducible polynomial $f(x)$ with root m, ie. $f(m) \equiv 0 \mod n$, $f(x) \in \mathbb{Z}[x]$.
Step 2: *(Factor bases)*
Choose the size for the *factor bases* and set up the *rational factor base*, *algebraic factor base* and the *quadratic character base*.
Step 3: *(Sieving)*
Find pairs of integers (a, b) with the following properties:

- $\gcd(a, b) = 1$
- $a + bm$ is smooth over the rational factor base
- $b^{\deg(f)} f(a/b)$ is smooth over the algebraic factor base

A pair (a, b) with these properties is called a *relation*. The purpose of the sieving stage is to collect as many relations as possible (at least one larger than the elements in all of the bases combined). The sieving step results in a set $\mathcal{S}$ of relations.
Step 4: *(Linear algebra)*
Filter the results from the sieving by removing duplicates and the relations containing a prime ideal not present in any of the other relations.
The relations are put into relation-sets and a very large sparse matrix over $\mathbf{GF}(2)$ is constructed.
The matrix is reduced resulting in some dependencies, ie. elements which lead to a square modulo n.
Step 5: *(Square root)*
Calculate the rational square root, ie. y with

$$y^2 = \prod_{(a,b)\in\mathcal{S}} (a - bm)$$

Calculate the algebraic square root, ie. x with

$$x^2 = \prod_{(a,b)\in\mathcal{S}} (a - b\alpha)$$

where α is a root of $f(x)$.
p can then be found by $\gcd(n, x - y)$ and $\gcd(n, x + y)$.

The outline above gives an idea of the flow in the algorithm. The steps described above are nowhere near a description of the algorithm that can be used for implementation. For implementation purposes the steps have to be unfolded to a number of algorithms without any steps concealed in mathematical properties and assumptions.

The steps described above are the ones I will refer to in the rest of this chapter.

6.2 Implementing the GNFS Algorithm

In this Section I will describe the GNFS algorithm to such an extent, that it is possible to make a working implementation of the algorithm. In Section 5.2.4 I described why and how the algorithm works from a mathematical point of view, and summarized the main theory behind the algorithm.

6.2.1 Polynomial Selection

Selecting a usable polynomial is not difficult, but a good polynomial is hard to find, because the concept of "a good polynomial" is not that clear.

There is some research in this area, but it is one of the least understood elements of the number field sieve. The best reference on the subject is Murphy's thesis[49].

The *yield* of a polynomial $f(x)$ refers to the number of smooth values it produces in its sieve region. There are two main factors which influence the yield of a polynomial. These are *size* and *root properties*. A polynomial $f(x)$ is said to be good if it has a good yield, ie. has a good combination of size and root properties.

Definition 6.1 A polynomial $f(x)$ has a good *size property* if the values taken by $f(x)$ are small. This can be achieved by selecting a polynomial with small coefficients. The smaller the size the better the size property.

Definition 6.2 A polynomial $f(x)$ is said to have good *root properties* if $f(x)$ has many roots modulo small primes.

There is no direct way to choose a good polynomial. The best method is to assure some level of size and root properties and then create candidates which are then tested and the one with the best yield is chosen.

The problem lies in finding good polynomials among the usable candidates. *Brian Antony Murphy* has in his thesis[49] done a lot of work in polynomial yield.

The polynomial $f(x)$ has to have the following properties

1. is irreducible over $\mathbb{Z}[x]$
2. has a root m modulo n

It is easy to create a polynomial with some desired root m in $\mathbb{Z}/\mathbb{Z}n[x]$ by using a *base-m representation*. This will also (in most cases) result in an irreducible polynomial, otherwise we have obtained the factorization of n, since

we would have $f(x) = g(x)h(x)$ and $f(m) = g(m)h(m) = n$. This should of course be checked before proceeding.

The results from constructing a polynomial $f(x)$ by using the base-m representation of n can be modified freely as long as $f(m) \equiv 0 \mod n$. That is why the polynomial does not have to be a true base-m expansion but an expansion of kn for some integer k.

The base-m representation of n is

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

with $0 < a_i < m$. d is determined in advance and is normally in the range 3-6. A degree d polynomial $f(x)$ can be constructed with the a_i's as coefficients. This polynomial will have m as a root, ie.

$$\begin{aligned} f(x) &= a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0 \\ f(m) &\equiv 0 \mod n \end{aligned}$$

The size property can be obtained by making the coefficients smaller, for example by making the replacements

$$\begin{aligned} a_i &= a_i - m \\ a_{i+1} &= a_{i+1} + 1 \end{aligned}$$

which results in smaller a_i's whilst preserving $f(m) \equiv 0 \mod n$. It is especially important to have a_d and a_{d-1} small.

The best polynomial yield seems to be obtained using skewed polynomials, but these are a lot more complex to create, I will use non-skewed polynomials.

Murphy have made a usable heuristic for measuring polynomial yield. He defines the $\alpha(F)$ function, which gives a very good estimate, whereby the obviously bad polynomials can be removed, and the remaining polynomials can then be put through a small sieving test and thereby identifying the one with the best yield.

The α-function is defined as

$$\alpha(F) = \sum_{p \le B} \left(1 - q_p \frac{p}{p+1}\right) \frac{\log p}{p-1}$$

where B is some smoothness bound, and q_p is the number of distinct roots of $f \mod p$.

The polynomial selection consists of the following steps

1. identify a large set of usable polynomials.
2. using heuristics remove obviously bad polynomials from the set.
3. do small sieving experiments on the remaining polynomials and choose the one with the best yield.

I have presented enough details to describe an algorithm for creating polynomials suitable for the general number field sieve.

algorithm: **Polynomial Selection (non-skewed)**

input: composite integer n, integer d.
output: polynomial $f(x)$

1. **choose** m_0 such that $\lfloor n^{\frac{1}{d+1}} \rfloor \leq m_0 \leq \lceil n^{\frac{1}{d}} \rceil$
2. **choose** interval of suitable a_d's such that $\chi_1 < \frac{|a_d|}{m_0} < \chi_2$, where $0 < \chi_1 < \chi_2 < 0.5$.
3. **define** the interval of m's such that $\mid a_{d-1} \mid$ are smallest. This will be the interval from m_0 to where a_d change, which is $m_{\text{change}} = \left\lfloor \left(\frac{n}{a_d}\right)^{\frac{1}{d}} \right\rfloor$.
4. **for** all usable combinations of a_d and $m \in [m_0 : m']$ with a_d having a large b-smooth cofactor **do**
   (a) **write** n in base-m representation, such that $n = a_d m^d + a_{d-1} m^{d-1} + \cdots + a_0$.
   (b) **write** the polynomial $f_m(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0$
   (c) **check** yield, by calculating $\alpha(f_m)$.
      i. **if** yield is satisfying **add** $f_m(x)$ to the set F'.
      ii. **else** discard polynomial.
5. **choose** $f(x)$ among the polynomials in F', by doing small sieving experiments.
6. **return** $f(x)$.

6.2.2 Factor Bases

The algorithm needs a well defined domain to work in, and this is specified with the factor bases.

The sizes of the factor bases have to be determined empirically, and are dependent on the precision of the sieving code, if all smooth elements are found or if one skips some by using special-q methods.

The *rational factor base* **RFB** consists of all the primes p_i up to some bound and we also store $p_i \mod m$, so the rational factor base consists of pairs $(p, p \mod m)a$, ie.

$$\textbf{RFB} = (p_0, p_0 \mod m), (p_1, p_1 \mod m), \ldots$$

The *algebraic factor base* **AFB** consists of pairs (p, r) such that $f(r) \equiv 0 \mod p$, ie.

$$\mathbf{AFB} = (p_0, r_0), (p_1, r_1), \ldots$$

The size of the algebraic factor base should be 2-3 times the size of the rational factor base.

The *quadratic character base* **QCB** contains pairs (r, p) with the same properties as the elements of the algebraic factor base, but the p's are larger than the largest in the algebraic factor base, ie.

$$\mathbf{QCB} = (p_0, r_0), (p_1, r_1), \ldots$$

The number of elements in **QCB** are usually relatively small compared to the number of elements in **RFB** and **AFB**. In recordbreaking factorizations the number is < 100 see [7].

6.2.3 Sieving

The sieving step is not the theoretically most complex part of the algorithm, but it is the most time consuming part beacuse it iterates over a large domain with some expensive calculations like division and modulo, although some of these can be avoided by using logarithms.

In general optimization of the sieving step will give the biggest reduction in actual running time of the algorithm.

It is easy to use a large amount of memory in this step, and one should be aware of this and try to reuse arrays and use the smallest possible data types. The factor bases can for record factorizations contain millions of elements, so one should try to obtain the best on-disk/in-memory tradeoff.

The purpose of the sieving step is to find usable relations, i.e. elements (a, b) with the following properties

- $\gcd(a, b) = 1$
- $a + bm$ is smooth over the rational factor base
- $b^{\deg(f)} f(a/b)$ is smooth over the algebraic factor base

Finding elements with these properties can be done by various sieving methods like the classical *line sieving* or the faster *lattice sieving*.

Line sieving is done by fixing b and sieving over a range of (a, b) for $a \in [-C : C]$. This means that the rational norm is calculated for all values of (a, b) and then it is divided with elements from the **RFB** and the entries with a resulting 1 are smooth over the **RFB** and the same procedure is done for the algebraic norm over the **AFB**.

It is not necessary to test whether each element on a sieving line factors completely over the factor bases by trial dividion with each element from the **RFB** and **AFB**.

The elements on the sieving line which have a given element from the factor base as a factor are distributed along the sieving line by a simple pattern.

- The elements (a, b) with rational norm divisible by element (p, r) from **RFB** are those with a on the form $a = -bm + kp$ for $k \in \mathbb{Z}$.

- The elements (a, b) with algebraic norm divisible by element (p, r) from **AFB** are those with a on the form $a = -br + kp$ for $k \in \mathbb{Z}$.

With these properties in mind, we can take each element from the factor base and remove its contribution from the elements that have it as a factor, and this is the idea of the line sieving algorithm given here.

algorithm: **Line sieving**

input: **RFB**,**AFB**,**QCB**,polynomial $f(x)$, root m of $f(x) \mod n$, integer C
output: list of pairs $rels = \{(a_0, b_0), \ldots, (a_t, b_t)\}$

1. $b = 0$
2. $rels = []$
3. **while** $\#rels < \#\textbf{RFB} + \#\textbf{AFB} + \#\textbf{QCB} + 1$
   (a) **set** $b = b + 1$
   (b) **set** $a[i] = i + bm$ for $i \in [-C; C]$
   (c) **foreach** $(p, r) \in$ **RFB**
      i. Divide out largest power of p from $a[j]$ where $j = -bm + kp$ for $k \in \mathbb{Z}$ so that $-C \leq j \leq C$.
   (d) **set** $e[i] = b^{\deg(f)} f(i/b)$ for $i \in [-C; C]$
   (e) **foreach** $(p, r) \in$ **AFB**
      i. Divide out largest power of p from $e[j]$ where $j = -br + kp$ for $k \in \mathbb{Z}$ so that $-C \leq j \leq C$.
   (f) **for** $i \in [-C; C]$
      i. **if** $a[i] = e[i] = 1$ **and** $\gcd(i, b) = 1$ add (i, b) to $rels$
   (g) $b = b + 1$
4. **return** $rels$.

6.2.3.1 Speeding up the sieve

Another sieving algorithm often used is *lattice sieving* proposed by John Pollard in [59]. It is similar to the *special-q* method for the Quadratic sieve algorithm. The factor bases are split into smaller sets and then the elements which are divisible by a large prime q are sieved. This speeds up the sieve at the expense of missing some smooth elements.

One obvious optimization that can be done for both the line- and the lattice sieve is to use logarithms to avoid divisions.

This means that one should store the logarithm of the norms and then subtract $\log(p)$ instead of dividing it, due to one of the laws of logarithms

$$\log\left(\frac{n}{p}\right) \quad = \quad \log(n) - \log(p)$$

So we avoid the many divisions of large integers, but there are some issues one should be aware of under this optimization

1. There is no way of detecting powers of primes p^j in the norms, so if 3^7 is a factor in the norm only $\log(3)$ will be subtracted.
2. Because of the floating point operations, correct elements can be bypassed and wrong elements can be included in the result.

The first problem can to some extent be handled by adding a *fuzz* factor to the logarithms. If a norm is not **RFB**-smooth then it must have a factor which is greater than the largest prime in **RFB**, so if we subtract $\log(\max(\textbf{RFB}))$ from all the entries, we should catch more of the elements which are divisible by prime powers without approving more wrong ones.

Both of the issues mentioned above imply that the elements found should be trial divided to assure that they all are **RFB**- and **AFB**-smooth.

6.2.4 Linear Algebra

From the sieving step we have a list of (a, b)'s which are **RFB**- and **AFB**-smooth and we want to find a subset of these which yields a square, ie. we have to find a combination of elements from the relation set which has a product that is a square.

For a number to be a true square, the elements in its unique factorization must have an even power, and this property is what we use to find elements that are squares.

To clarify we can simply say that we have a list of numbers:

$$\{34, 89, 46, 32, 56, 8, 51, 43, 69\}$$

We want to find a subset of these numbers which forms a product that is a square, one solution is:

$$\{34, 46, 51, 69\}$$

With the product

$$34 \cdot 46 \cdot 51 \cdot 69 = 5503716 = 2^2 \cdot 3^2 \cdot 17^2 \cdot 23^2 = (2 \cdot 3 \cdot 17 \cdot 23)^2$$

This is equivalent to solving a system of linear equations and can be done by building a matrix and eliminate it. The bottleneck in this operation is the dimensions of the linear system - which can result in a matrix with dimensions larger than $10^9 \times 10^9$, which can be hard to represent in memory on a modern computer.

The matrix consists of the factorization over the rational- and algebraic bases and some information derived from the quadratic character base along

with the sign of the norm. Since we are only interested in the parity of the power of the elements in the factorization (even/odd), we put a 1 if the element appears as an odd power in the factorization and a 0 if its even or zero. This should be more clear from the algorithm below and the extended example in 6.3.

This results in a matrix $\mathcal{M}$ over $\mathbf{GF}(2)$ with dimensions $(\#relations) \times (\#\mathbf{RFB} + \#\mathbf{AFB} + \#\mathbf{QCB} + 1)$.

algorithm: **Building the matrix**

input: **RFB**,**AFB**,**QCB**,polynomial $f(x)$, root m of $f(x)$, list $rels = \{(a_0, b_0), \ldots, (a_t, b_t)\}$ of smooth pairs
output: Binary matrix $\mathcal{M}$ of dimensions $(\#rels) \times (\#\mathbf{RFB} + \#\mathbf{AFB} + \#\mathbf{QCB} + 1)$

1. **set** all entries in $\mathcal{M}[i, j] = 0$
2. **foreach** $(a_i, b_i) \in rels$
   (a) **if** $a_i + b_i m < 0$ **set** $\mathcal{M}[i, 0] = 1$
   (b) **foreach** $(p_k, r_k) \in \mathbf{RFB}$
      i. **let** l be the biggest power of p_k that divides $a_i + b_i m$
      ii. **if** l is odd **set** $\mathcal{M}[i, 1 + k] = 1$
   (c) **foreach** $(p_k, r_k) \in \mathbf{AFB}$
      i. **let** l be the biggest power of p_k that divides $(-b_i)^d f(-\frac{a_i}{b_i})$
      ii. **if** l is odd **set** $\mathcal{M}[i, 1 + \#\mathbf{RFB} + k] = 1$
   (d) **foreach** $(p_k, r_k) \in \mathbf{QCB}$
      i. **if** the Legendre symbol $\left(\frac{a_i + b_i p_k}{r_k}\right) \neq 1$ **set** $\mathcal{M}[i, 1 + \#\mathbf{RFB} + \#\mathbf{AFB} + k] = 1$
3. **return** $\mathcal{M}$.

This matrix can then be put on reduced echelon form and from this we can derive solutions which yield a square.

The process of solving linear systems by reducing a matrix of the system has been thoroughly studied since the birth of algebra and the most commonly used method is *Gaussian Elimination*. It is fast and easy to implement but has one drawback in this case; as the matrix is quite large, memory-usage becomes a problem.

There exist variations of Gaussian Elimination which utilize block partioning of the matrix, but often a completely different method is used, namely the *Block Lanczos* method specialized for the $\mathbf{GF}(2)$ case by Peter Montgomery in [46]. It is probabilistic but in both theory and practice the algorithm is

faster for large instances of the problem. It utilizes an iterative approach by dividing the problem into orthogonal subspaces.

I will only describe Gaussian Elimination which will work even for large problems if sufficient memory is available, but the interested reader should turn to [46] for more details on the Block Lanczos method.

algorithm: **Gaussian Elimination**

input: $n \times m$ matrix $\mathcal{M}$
output: $\mathcal{M}$ on reduced echelon form

1. **set** $i = 1$
2. **while** $i \leq n$
   (a) **find** first row j from row i to n where $\mathcal{M}[j, i] = 1$, if none exists try with $i = i + 1$ until one is found.
   (b) **if** $j \neq i$ **swap** row i with row j
   (c) **for** $i < j < n$
      i. **if** $\mathcal{M}[j, i] = 1$ subtract row i from rwo j
   (d) **set** $i = i + 1$
3. **for** each row $0 < j \leq n$ with a leading 1 (in column k)
   (a) **for** $0 < i < j$
      i. **if** $\mathcal{M}[i, k] = 1$ subtract row j from row i
4. **return** $\mathcal{M}$.

6.2.5 Square Root

From the linear algebra step we get one or more solutions, i.e. products which are squares and thereby can lead to a trivial or non-trivial factor of n.

We need the square root of the solution, a rational square root $\mathcal{Y}$ and an algebraic square root $\mathcal{X}$. The rational squareroot is trivial, although the product of the solution from the linear algebra step is large, it is still an integer and a wide variety of methods for deriving the square root is known.

algorithm: **Rational square root**

input: n,polynomial $f(x)$, root m of $f(x)$, list of smooth elements the product of which is a square $deps = \{(a_0, b_0), \ldots, (a_t, b_t)\}$
output: integer $\mathcal{Y}$

1. **compute** the product $S(x)$ in $\mathbb{Z}[x]/f(x)$ of the elements in $deps$
2. **return** $\mathcal{Y} = \sqrt{S(m) \cdot f'(m)^2} \mod n$

The reason for multiplying by $f'(x)^2$ is given in 5.2.4.1.

Finding the square root of an algebraic integer is far from trivial, and is the most complex part of the GNFS algorithm but not the most time consuming one, and optimization of this step is therefore not important.

Finding the square root of an algebraic integer is equivalent to finding the square root of a polynomial over an extension field. There exist algorithms for factoring polynomials in various fields but not many for factoring over a number field.

Montgomery has once again developed a method for this in [45] and further refined in [52], but it is complex and uses lattice reduction which depends on a lot more theory than covered in this thesis. I will instead describe a method based on [14] and used in [7].

It does an approximation and uses a Newton iteration to find the true square root.

algorithm: **Algebraic square root**

input: n,polynomial $f(x)$, root m of $f(x)$, list of smooth elements the product of which is a square $deps = \{(a_0, b_0), \ldots, (a_t, b_t)\}$
output: integer $\mathcal{X}$

1. **compute** the product $S(x)$ in $\mathbb{Z}[x]/f(x)$ of the elements in $deps$
2. **choose** a large prime p (e.g. 2-5 times larger than n)
3. **choose** random $r(x) \in \mathbb{Z}_p[x]/f(x)$ with $\deg(r) = \deg(f) - 1$
4. **compute** $R_0 + R_1 y = (r(x) - y)^{\frac{p^d-1}{2}} \in (\mathbb{Z}_p[x]/f(x))[y]/(y^2 - S)$, i.e. compute the $\frac{p^d-1}{2}$ power of $r(x)$ modulo $y^2 - S$.
5. **if** $SR_1^2 \neq 1$ **goto** 2 and choose other p and/or $r(x)$
6. **set** $k = 0$
7. **set** $k = k + 1$
8. **compute** $R_{2k} = \frac{R_k(3 - SR_k^2)}{2} \mod p^{2k}$
9. **if** $(R_k S)^2 \neq S$ **goto** 7
10. **compute** $s(x) = \pm SR_k$.
11. **return** $\mathcal{X} = s(m) \cdot f'(m) \mod n$

Once again I multiply with $f'(x)^2$ as described in Section 5.2.4.1.

6.3 An extended example

To clarify the different steps of the GNFS algorithm and to convince the reader, I will now go through the algorithm by providing an extended example.

I want to factor the number $n = 3218147$, which should be tested for primality before proceding with the factorization.

6.3.1 Setting up factor bases

The next step is to set up the factor bases. The sizes are chosen empirically and the main measure is to find the size which gives the fastest sieving as well as being small, because a smaller size in factor bases would give a smaller number of relations to work within the linear algebra stage.

I have chosen the primes below 60, and as described in Section 6.2.2 we store the prime $\bmod m$ as well, so the rational factor base consists of the 16 elements

RFB	=	(2,1)	(3,0)	(5,2)	(7,5)
		(11,7)	(13,0)	(17,15)	(19,3)
		(23,2)	(29,1)	(31,24)	(37,6)
		(41,35)	(43,31)	(47,23)	(53,11)

Now I have to construct the algebraic factor base and I choose it to be approximately 3 times the size of the **RFB**. The **AFB** consists of elements (p, r) where $f(r) \equiv 0 \mod p$, e.g. $f(7) \equiv 0 \mod 17$ so $(7, 17)$ is in the **AFB**, which consists of the following 47 elements

AFB	=	(2,0)	(2,1)	(3,2)	(3,1)
		(5,1)	(5,3)	(13,10)	(17,2)
		(17,16)	(17,7)	(19,9)	(29,10)
		(31,0)	(31,12)	(31,3)	(41,4)
		(41,8)	(43,36)	(43,23)	(43,5)
		(53,40)	(59,6)	(61,12)	(61,38)
		(61,41)	(71,66)	(79,28)	(83,44)
		(89,9)	(101,65)	(103,87)	(109,64)
		(109,85)	(109,14)	(127,95)	(131,57)
		(131,108)	(131,31)	(149,49)	(157,63)
		(163,155)	(163,126)	(179,165)	(193,105)
		(197,93)	(197,11)	(197,191)	

Even though I do not need the quadratic character base for the sieving step, I construct it here. It consists of the same type of elements as the **AFB**, but the elements are larger than the largest in the **AFB**. I choose to have 6 elements in the **QCB**

$$\textbf{QCB} = (233, 166) \quad (233, 205) \quad (233, 211) \quad (281, 19) \quad (281, 272) \quad (281, 130)$$

After setting up the bases I have the following:

- $n = 3218147$
- $m = 117$
- $f(x) = 2x^3 + x^2 + 10x + 62$
- Factor bases: **RFB,AFB,QCB**

6.3.2 Sieving

Now I want to find elements that are **RFB**-smooth and **AFB**-smooth. This is done by sieving, as described in 6.2.3. I start by selecting a linewidth for the sieving, I choose to sieve with $a \in [-200; 200]$, then I follow the algorithm from 6.2.3, which gives me the number of smooth elements wanted, e.g. the pair $(13, 7)$ has rational norm

$$\begin{aligned} \mathcal{N}_{\text{rational}}(a, b) &= a + bm \\ \mathcal{N}_{\text{rational}}(13, 7) &= 13 + 7 \cdot 117 \\ &= 832 \\ &= 2^6 \cdot 13 \end{aligned}$$

i.e. it is smooth over the **RFB**. It has algebraic norm

$$\begin{aligned} \mathcal{N}_{\text{algebraic}}(a, b) &= (-b)^d f(-\frac{a}{b}) \\ \mathcal{N}_{\text{algebraic}}(13, 7) &= (-7)^3 f(-\frac{13}{7}) \\ &= -11685 \\ &= (-1) \cdot 3 \cdot 5 \cdot 19 \cdot 41 \end{aligned}$$

i.e. it is also smooth over the **AFB**, so $(13, 7)$ is one of the desired elements. In total I need at least 70 elements (#**RFB**+#**AFB**+#**QCB**+1). I find the following 72 elements

rels =	(-186,1)	(-155,1)	(-127,1)	(-126,1)	(-123,1)
	(-101,1)	(-93,1)	(-68,1)	(-66,1)	(-65,1)
	(-49,1)	(-41,1)	(-36,1)	(-31,1)	(-23,1)
	(-12,1)	(-9,1)	(-7,1)	(-6,1)	(-5,1)
	(-3,1)	(-2,1)	(-1,1)	(0,1)	(2,1)
	(3,1)	(4,1)	(6,1)	(7,1)	(8,1)
	(13,1)	(16,1)	(19,1)	(23,1)	(24,1)
	(37,1)	(81,1)	(100,1)	(171,1)	(-65,3)
	(-62,3)	(-31,3)	(-8,3)	(-1,3)	(17,3)
	(26,3)	(86,3)	(181,3)	(200,3)	(-137,5)
	(-102,5)	(-68,5)	(-46,5)	(-24,5)	(-7,5)
	(7,5)	(31,5)	(39,5)	(-152,7)	(-83,7)
	(-44,7)	(9,7)	(13,7)	(17,7)	(18,7)
	(26,7)	(41,7)	(-64,9)	(-17,9)	(5,9)
	(11,9)	(20,9)			

After sieving I have the following:

- $n = 3218147$
- $m = 117$
- $f(x) = 2x^3 + x^2 + 10x + 62$
- Factor bases: **RFB,AFB,QCB**
- List of smooth elements $rels$

6.3.3 Linear Algebra

After the sieving step, I have a list of 72 elements which are smooth over the factor bases. Now I need to find one or more subset(s) of these which yield a product that is a square. This is done by solving a system of linear equations. The matrix I use is built as described in Section 6.2.4, e.g. the element $(13, 7)$ is represented in the following way ($(13, 7)$ is the 63th element in $rels$).

The first entry is the sign of $\mathcal{N}_{\text{rational}}(13, 7)$ which is positive and therefore we have

$$\mathcal{M}[63] = 0 \ldots$$

The next 16 entries are the factorization of $\mathcal{N}_{\text{rational}}(13, 7)$ over **RFB**, and we only store the parity of the power, so I get

$$\mathcal{M}[63] = 0000000010000000000 \ldots$$

The next 47 entries are the factorization of $\mathcal{N}_{\text{algebraic}}(13, 7)$ over **AFB**, and I store only the parity of the power, so I get

$$\mathcal{M}[63] = 0000000010000000000001010000010000100000000000000000000000000000000\ldots$$

The last 6 entries are for the **QCB**. For each element (p_i, r_i) in the **QCB** I store a 1 if the Legendre symbol $\left(\frac{13+7\cdot p_k}{r_k}\right) \neq 1$ else a 0, so I get

$$\mathcal{M}[63] = 0000000010000000000001010000010000100000000000000000000000000000000100100$$

Doing this for all the relations gives the 72×70 matrix $\mathcal{M}$

$$\mathcal{M} = \begin{bmatrix}
1010000001000000010001000100000000000000000100000000000000000000100101 \\
1100000010000000000100001000110000000000000000000000000000100000100111 \\
1101000000000000000010010000000110000000000010000000000000000000111101 \\
1000000000000000010000000010100000000000000000000000000000100000110110 \\
1110000000000000000000100001000000000000000000000000100000000000111100 \\
0000000000000000000101010100000000000000000000000000000000000000100110 \\
0110000000000000000000100000010000000100000000000000000000000100101111 \\
0000000000000000010100100000100000000000000000000000000000000000011110 \\
0010000100000000010001000001000000010000000100000000000000000000011110 \\
\vdots \\
\vdots \\
0000001000000000000101000001000010000000000000000000000000000000100100 \\
0000010010000000000010000000100000000010000000000000000000000000011111 \\
0010000000010000010000000000000000000000010000000000000000000000001100 \\
0001000000000000010010000000000010000000000100000000000000000000000111 \\
0001000000000010000010000100000100000000000100000000000000000000010111 \\
0000000001000010010001010000000000000000010010000000000000000000010100 \\
0000100000001000000000100000000000000000000000000100010000000000100100 \\
0100000000000000000000000010000010000010000000000000000000000000001111 \\
0100100010000000000001000000000000000100000000000000100000000000000100 \\
0000000000000000000000000000000000000000000000000000000000000000000000
\end{bmatrix}$$

The matrix is then transposed, an additional column is appended and I apply the Gauss-Jordan algorithm to it, giving the result on reduced echelon form

$$\mathcal{M}_e = \begin{bmatrix}
1000000000000000000000000000000000000000000000000000000000000000000000100000 \\
0100000000000000000000000000000000000000000000000000000000000000000000100000 \\
0010000000000000000000000000000000000000000000000000000000010000101001001000 \\
0001000000000000000000000000000000000000000000000000000000010001011011101000 \\
0000100000000000000000000000000000000000000000000000000000000111001010000000 \\
0000010000000000000000000000000000000000000000000000000000001010010110000000 \\
0000001000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000100000000000000000000000000000000000000000000000000000111000010100000 \\
0000000010000000000000000000000000000000000000000000000000001111110000100000 \\
0000000001000000000000000000000000000000000000000000000000001111011011000000 \\
0000000000100000000000000000000000000000000000000000000000001110101000000000 \\
\vdots \\
\vdots \\
0000000000000000000000000000000000000000000000000000000000010000011101011110000 \\
0000000000000000000000000000000000000000000000000000000000010000111110101000000 \\
0000000000000000000000000000000000000000000000000000000000010000000111001000000 \\
0000000000000000000000000000000000000000000000000000000000001001110010110000000 \\
0000000000000000000000000000000000000000000000000000000000001111110111011000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000010 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000000000000000000000000000000
\end{bmatrix}$$

From this I get a vector of free variables, i.e. elements which lead to a solution by setting one or more of them to 1.

$$\mathcal{V}_{\text{free}} = [00000000000000000000000000000000000000000000000000000000000010011111111111101]$$

After linear algebra I have the following:

- $n = 3218147$
- $m = 117$
- $f(x) = 2x^3 + x^2 + 10x + 62$
- Factor bases: **RFB,AFB,QCB**
- List of smooth elements $rels$
- Matrix $\mathcal{M}_e$
- Vector of free relations $\mathcal{V}_{\text{free}}$

6.3.4 Square roots

I now have different solutions to choose from by selecting one or more of the free variables from $\mathcal{V}_{\text{free}}$, i.e. I can choose to add one or more of the free variables to the solution. I take the first free variable and get the solution

$$\mathcal{V}_{\text{sol}} = [00110000000000010001010110000000111110011000001101011000100000000000000]$$

Which can easily be verified by $\mathcal{M}_e \cdot \mathcal{V}_{\text{sol}} = 0$. So the following 20 elements are to be used

deps =	(-127,1)	(-126,1)	(-12,1)	(-5,1)
	(-2,1)	(0,1)	(2,1)	(19,1)
	(23,1)	(24,1)	(37,1)	(81,1)
	(-65,3)	(-62,3)	(86,3)	(181,3)
	(-137,5)	(-68,5)	(-46,5)	(31,5)

Now I apply the algorithm from 6.2.5, Ind I take it step by step.
I start by computing the product $S(x)$ in $\mathbb{Z}[x]$

$$S(x) = (-127+x)\cdot(-126+x)\cdot\ldots\cdot(-46+5x)\cdot(31+5x)$$

The rational square root $\mathcal{Y}$ is found by

$$\begin{aligned}\sqrt{S(m)\cdot f'(m)^2} &= \sqrt{204996970067909038034618021120435457884160000\cdot 6786134884}\\ &= \sqrt{139113708969214137194560415274043582601821100703744 0000}\\ &= 11794647471171579581478912 00\end{aligned}$$

$$\begin{aligned}\mathcal{Y} &= \sqrt{S(m)\cdot f'(m)^2} \mod n\\ &= 11794647471171579581478912 00 \mod 3218147\\ &= 2860383\end{aligned}$$

From now we need the product $S(x)$ in $\mathbb{Z}[x]/f(x)$ instead of $\mathbb{Z}[x]$

$$\begin{aligned}S(x) &= (-127+x)\cdot(-126+x)\cdot\ldots\cdot(-46+5x)\cdot(31+5x) \mod f(x)\\ &= \frac{-1747051404798697196153516520153907 5}{262144}\cdot x^2 +\\ &\quad \frac{3221408885540911801419827086788375}{131072}\cdot x +\\ &\quad \frac{1164401337297998456998315348104339 75}{131072}\end{aligned}$$

For computing the algebraic square root $\mathcal{X}$ I need to do some more work. I start by choosing a random 128 bit prime p

$$p = 42 \cdot 10^{42} + 43$$

I then choose a random $r(x) \in \mathbb{Z}_p[x]/f(x)$

$$\begin{aligned} r(x) &= 331433955179089017205088896783327744131509 64 \cdot x^2 + \\ &\quad 993511604982289903454724652876234630482885 6 \cdot x + \\ &\quad 375233784773654088638092652579164104220841 20 \end{aligned}$$

As described in 6.2.5 I then calculate $R_0 + R_1 y \in (\mathbb{Z}_p[x]/f(x))[y]/(y^2 - S)$ and get

$$\begin{aligned} R_1(x) &= 404384437426466821629760580101071467678509 28 \cdot x^2 + \\ &\quad 109758739765744771607769176310600913432507 04 \cdot x + \\ &\quad 256024528177751590591946876445648810105936 83 \end{aligned}$$

This is usable, i.e. $SR_1^2 = 1$ so I compute R_2 and get

$$\begin{aligned} R_2(x) = {} & 148428045253485193219118873225285686003130691005890705213794607300261722136536060942545 3x^2 + \\ & 101243878338502139540877286172500592345110294552034268028685817452056177808996535271217 1x + \\ & 337076433860489670648869780713225956803031046704516055895536152085177422391455832741 37 \end{aligned}$$

So I get

$$\begin{aligned} S_{\text{alg}}(x) &= \pm S(x) R_2(x) \mod p^2 \\ &= \pm \left(\frac{-17805551987379270x^2 + 460901612569132380 + 14073072352402140x}{262144} \right) \end{aligned}$$

And then I get the algebraic square root

$$\begin{aligned} \mathcal{X} &= S_{\text{alg}}(m) \cdot f'(m) \mod n \\ &= 484534 \end{aligned}$$

So now I have $\mathcal{Y}$ and $\mathcal{X}$ and I can find the divisors of n

$$\begin{aligned} n_{\text{divisor 1}} &= \gcd(n, \mathcal{X} - \mathcal{Y}) \\ &= \gcd(3218147, 484534 - 2860383) \\ &= 1777^1 \\ n_{\text{divisor 2}} &= \gcd(n, \mathcal{Y} - \mathcal{X}) \\ &= \gcd(3218147, 2860383 - 484534) \\ &= 1811^2 \end{aligned}$$

So I have computed that n is the product of 1777 and 1811, and this is in fact the prime factorization.

[1] Birth year of the great Carl Friedrich Gauss(1777-1855)
[2] Birth year of the legendary Evariste Galois(1811-1832)

CHAPTER 7

Implementing GNFS

"Machines take me by surprise with great frequency."

- Alan Turing(1912-1954)

In Chapter 6 I described the number field sieve algorithm in details and this description has been used to implement the algorithm and the implementation has been used to correct the algorithms.

Implementing all the steps of the algorithm is a large task and it seems like the few people who have made a working version have felt that their hard work should not be accessible to everyone, and have thus kept the implementation to themselves. This is far from optimal as some interesting views on the different parts could have emerged from making it publicly available.

There exist few public implementations, and all except one are unusable for large factorizations because they are implemented with a lot of limitations for example by predefining the degree of the polynomial and limiting the size of the factor bases and the size of the numbers that can be tried.

There exists one public workable implementation[1] written in C, but the code is somewhat incomprehensible and is only used as-is instead of being further developed by other than Chris himself. The code lacks an underlying framework to make it more modularized and the interfaces between the different steps are much to complex.

My implementation of the number field sieve has taken months to develop and consulting previous work done on the number field sieve did not always help me past the obstacles and especially the square root step involved much more work than expected. As the days passed it was more and more clear to me that this thesis was needed to make the number field sieve more digestible.

My implementation has the following purposes

- To show that the algorithms for the different steps in Chapter 6 work in practice.
- To be a general framework for number field sieve implementations, by providing a clear interface between the various steps.

[1] http://www.math.ttu.edu/~cmonico/software/ggnfs/

- To provide a reference implementation for future developers

My implementation is not

- the fastest implementation.
- kept a secret.
- limited in input, degree of the polynomial or size of factor bases.

It is available from the Internet[2] and hopefully it will evolve into a project with many participants.

In the rest of this Chapter I will go through my Implementation step by step and discuss the choices made for further details the source code can be consulted. I will also describe how to use my implementation.

7.1 pGNFS - A GNFS Implementation

My implementation of the number field sieve is named *pGNFS*, and it is implemented in standard C++, because it makes it very portable and the GCC compiler[3] generates fairly good executables without too much overhead (as does other C++-compilers). The easiest language to implement the algorithm in would probably be some higher level algebra language like Magma or Maple but they lack the performance and portability of the C++ language.

It is clear from the start that we cannot avoid doing some large integer arithmetic, so we need tools to enable us to work with large integers. We can either implement some routines and representation ourselves or use some of the publicly available libraries. My first idea was to use my own large integer library[4] but it is highly specialized for the Intel Pentium 4 architecture so I decided to use a more portable library. I have chosen to use NTL[5] for the most part, and use GiNaC[6] for the square root step.

NTL is developed by Victor Shoup and can be run with GnuMP[7] as the underlying representation, and since GnuMP is the fastest large integer arithmetic library available it is a good choice. NTL extends GnuMP's capabilities and implements polynomials and matrixes over finite fields, which gives a large set of tools that ease the implementation work.

GiNaC is used for the square root step because it supports symbolic manipulation of expressions which is very useful for example when working with the elements from $(\mathbb{Z}_p[x]/f(x))[y]/(y^2 - S)$. GiNaC is not as fast as GnuMP and NTL for doing integer arithmetic and is therefore not used in any of the other steps.

NTL, GnuMP and GiNaC do not limit the architectures the implementation can run on as they all can be compiled on most platforms.

[2] http://www.pgnfs.org
[3] http://www.gnu.org
[4] http://www.polint.org
[5] http://shoup.net/ntl/
[6] http://www.ginac.de
[7] http://swox.com/gmp/

7.1.1 Input/Output

As seen in Chapter 6 there are some variables to set and this mean that the number field sieve is not a "black box" factoring algorithm. It is dependent on choosing the right parameters and in Section 7.2 I will give an overview of selecting parameters for the algorithm. The parameters cannot be found algorithmic but only by emperically testing different values.

I have chosen to have one input file with values for all of the different steps this makes it more easy to use and to move an instance of a problem to another machine somewhere in between steps.

The values are put in a text file denoted `input` in the rest of this Chapter, it is the only parameter the programs are called with.

The different steps results in some more data these are creates in new files which are named after the input, i.e. if `pgnfs_polyselector` is called with the text file `mycomposite.txt` it results in a file `mycomposite.txt.poly`.

`input` holds the following values:

n	The integer to be factored
a_{factor}	A desired co-factor of the leading coefficient of the polynomial
p_{bound}	Prime bound for sieving experiments in the polynomial step
$\mathbf{RFB}_{\text{bound}}$	Bound for primes in **RFB**.
$\mathbf{AFB}_{\text{bound}}$	Bound for primes in **AFB**.
$\mathbf{QCB}_{\text{num}}$	Number of desired elements in **QCB**.
p_{file}	Absolute path of the file containing at least all primes up to $\mathbf{AFB}_{\text{bound}}$
C	Defining the sieving interval, i.e. sieving (a_i, b_i) where $a_i \in [-C; C]$

7.1.2 Polynomial Selection

The polynomial selection step is implemented in `pgnfs_polyselector` and follows the description in Section 6.2.1. It does not do any sieving experiments but it runs "forever" and every time it finds a polynomial with a higher score it overwrites `input.poly` with the new polynomial. It uses a_{factor} as a cofactor to the leading coefficient to "artificially" give it some smoothness.

Summary

Program:	`pgnfs_polyselector`
Use file(s):	`input`
Generate file(s):	`input.poly`

7.1.3 Factor Bases

The factor bases **RFB**,**AFB** and **QCB** are created naively as it is not a time consuming process compared to for example sieving.

The program `pgnfs_makebases` use the p_{file} for the primes needed and generates a file for each of the factor base. The files are stores in ascii-format

which means that they are storage consuming but can easily be compressed if they were to be distributed.

Summary	
Program:	`pgnfs_makebases`
Use file(s):	`input, input.poly`
Generate file(s):	`input.rfb,input.afb,input.qcb`

7.1.4 Sieving

Line sieving as described in 6.2.3 is implemented in `pgnfs_sieve`. It appends found relations to the file `input.rel` and current sieve line is stored in `input.line` so the program can be terminated at any given time and started again continuing from where it was stopped.

This also enables a primitive way of distributing the workload to multiple machines as they can be given a different sieving line to start from and the different results can then be merged.

Summary	
Program:	`pgnfs_sieve`
Use file(s):	`input, input.poly,input.rfb, input.afb`
Generate file(s):	`input.line,input.rel`

7.1.5 Linear Algebra

Matrix building and Gaussian elimination as described in 6.2.4 are implemented in `pgnfs_linearalg`. The program builds the matrix in memory and solves it before writing the matrix and the vector of free variables to file.

The matrix being built can be sparse and a possible improvement would be to compress it to a more dense matrix as the matrix dimensions can be very large and the matrix thus hard to represent in memory.

Summary	
Program:	`pgnfs_linearalg`
Use file(s):	`input,input.poly,input.rfb, input.afb,input.qcb,input.rel`
Generate file(s):	`input.dep,input.mat`

7.1.6 Square Root

The square root step is implemented in `pgnfs_sqrt` GiNaC's square root operation is used for the rational square root and the algebraic square root is implemented exactly as described in Section 6.2.5. It continues to try different solutions to a non-trivial factor is found or there is no more solutions to try.

The calculations use integer arithmetic with very large integers and should continuously be reduced to avoid unnecessary large computations.

Summary

Program:	`pgnfs_sqrt`
Use file(s):	`input,input.poly,input.rel,` `input.mat,input.dep`
Generate file(s):	`stdout`

7.1.7 Example of Input

The extended example in Section 6.3 is run by calling the described five programs in sequence with the following input file

example: **Input file for extended example**

```
3218147
12
1000
60
198
6
/home/pleslie/PROJECTS/Thesis/SRC/TEST/primes.txt
200
```

7.2 Parameters

Th number field sieve is, as mentioned, not a "black box" factoring machine. It needs the right parameters, and these are not intuitively selected.

The values are empirically studied and guidelines for selecting parameters are also dependent on the implementation. As described previously it is the polynomial selected that dictates the success. A bad polynomial would making the sieving step complicated as it does not generate smooth norms.

In the table below I have given parameters for successful factorizations with pGNFS and parameters used for record factorizations by others. These can be used as rough estimates for choosing the right parameters. The numbers are not necessarily the smallest that can be used, but they works.

name	digits of n	$\mathbf{RFB}_{\text{bound}}$	$\mathbf{AFB}_{\text{bound}}$	$\mathbf{QCB}_{\text{num}}$	C
`ext-example`	7	60	200	5	200
`mytest10`	10	100	300	10	1000
`mytest20`	20	400	1300	10	10000
`mytest40`	40	40000	100000	40	50000
`mytest60`	60	250000	700000	50	200000
RSA-140[16]	140	8000000	17000000	100	9000000000
RSA-155[15]	155	17000000	17000000	100	12000000000

Bibliography

"My method to overcome a difficulty is to go round it."

- George Pólya(1887-1985)

[1] T. Okamoto A. Menezes and S. Vanstone, *Reducing elliptic curve logarithms to logarithms in finite fields*, IEEE Transactions on Information Theory **39** (1993), 1639–1646.

[2] Niels Henrik Abel, *Recherches sur les fonctions elliptiques*, Journal für die reine und angewandte Mathematik **2** (1827), 101.

[3] Len M. Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*, Proceedings of the 20th FOCS (1979), 55–60.

[4] M. Agrawal, N. Kayal, and N. Saxena, *Primes is in p*, 2002.

[5] R. B. J. T. Allenby, *Rings, fields and groups 2nd edition*, Butterworth-Heinemann, 1991.

[6] Emil Artin, *Galois theory*, Dover Publications Inc., 1944.

[7] Daniel J. Bernstein and Arjen K. Lenstra, *A general number field sieve implementation*, in Lenstra and Hendrik W. Lenstra [39], pp. 103–126.

[8] Ian Blake, Gadiel Seroussi, and Nigel Smart, *Elliptic curves in cryptography*, Cambridge University Press, 1999.

[9] Boneh and Durfee, *Cryptanalysis of RSA with private key d less than* $n^{0.292}$, IEEETIT: IEEE Transactions on Information Theory **46** (2000).

[10] R. Brent, *Some integer factorization algorithms using elliptic curves*, 1986.

[11] Richard P. Brent, *An improved Monte Carlo factorization algorithm*, BIT **20** (1980), 176–184.

[12] ———, *Factorization of the tenth Fermat number*, Mathematics of Computation **68** (1999), 429–451.

[13] Matthew E. Briggs, *An introduction to the general number field sieve*, Master of science in mathematics thesis (1998).

[14] Joe P. Buhler, Jr. Hendrik W. Lenstra, and Carl Pomerance, *Factoring integers with the number field sieve*, in Lenstra and Hendrik W. Lenstra [39], pp. 50–94.

[15] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gerard Guillerm, Paul C. Leyland, Joel Marchand, Francois Morain, Alec Muffett, Chris Putnam, Craig Putnam, and Paul Zimmermann, *Factorization of a 512-bit RSA modulus*, Theory and Application of Cryptographic Techniques, 2000, pp. 1–18.

[16] Stefania Cavaller, Bruce Dodson, Arjen Lenstra, Paul Leyland, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, and Paul Zimmermann, *Factorization of rsa-140 using the number field sieve*, Proceedings of Asiacrypt '99 (1999).

[17] D. Chaum, *Blind signatures for untraceable payments.*, CRYPTO '82 (1983), 199–203.

[18] Clifford C. Cocks, *A note on non-secret encryption*, CESG report (classified until december 1997) (1973).

[19] D. Coppersmith, *Evaluating logarithms in gf(2n)*, Proc. 16th ACM Symp. Theory of Computing (1984), 201–207.

[20] ———, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Transactions on Information Theory **30** (1984), 587–594.

[21] ———, *Small solutions to polynomial equations, and low exponent rsa vulnerabilities.*, Journal of Cryptology **10** (1997), 223–260.

[22] Joan Daemen and Vincent Rijmen, *Rijndael for aes.*, AES Candidate Conference, 2000, pp. 343–348.

[23] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **IT-22** (1976), no. 6, 644–654.

[24] William Dunham, *Euler - the master of us all*, The Mathematical Association Of America, 1999.

[25] T. ElGamal, *A public-key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **IT-31** (1985), 10–18.

[26] J. H. Ellis, *The possibility of secure non-secret digital encryption*, CESG report (classified until december 1997) (1970).

[27] Andreas Enge, *Elliptic curves and their applications to cryptography, an introduction*, Kluwer Academic Publishers, 1999.

[28] Evariste Galois, *Mèmoire sur la division des fonctions elliptiques de première espèce*, Manuscrits et papiers inédits de Galois par M.J. Tannery (1907).

[29] Carl Friedrich Gauss, *Disquisitiones arithmeticae (english translation by arthur a. clarke)*, Yale University Press, 1801.

[30] Jr. H. W. Lenstra, *Elliptic curve factorization, personal communication via samuel wagstaff jr.*, 1985.

[31] Darrel Hankerson, Alfred Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag NY, 2004.

[32] R. M. Huizing, *An implementation of the number field sieve*, CWI Report **NM-R9511** (1995).

[33] J. Håstad, *Solving simultaneous modular equations of low degree*, SIAM Journal of Computing **17** (1988), 336–341.

[34] Christian U. Jensen, Arne Ledet, and Noriko Yui, *Generic polynomials - constructive aspects of the inverse galois problem*, Cambridge University Press, 2002.

[35] Paul Kocher, Joshua Jaffe, and Benjamin Jun, *Differential power analysis*, Lecture Notes in Computer Science **1666** (1999), 388–397.

[36] Paul C. Kocher, *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems.*, Advances in Cryptology - CRYPTO '96 - Lecture Notes in Computer Science **1109** (1996), 104–113.

[37] Kenji Koyama, Ueli M. Maurer, Tatsuaki Okamoto, and Scott A. Vanstone, *New public-key schemes based on elliptic curves over the ring z_n*, CRYPTO, 1991, pp. 252–266.

[38] Franz Lemmermeyer, *Reciprocity laws: From euler to eisenstein*, Springer-Verlag Berlin and Heidelberg GmbH & Co., 2000.

[39] Arjen K. Lenstra and Jr. Hendrik W. Lenstra (eds.), *The development of the number field sieve*, Lecture Notes in Mathematics, vol. 1554, Springer-Verlag, Berlin, 1993.

[40] Steven Levy, *Crypto - secrecy and privacy in the new code war*, Penguin Books, 2000.

[41] Robert James McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report **42-44** (1978), 114–116.

[42] Alfred J. Menezes, *Elliptic curve public key cryptosystems*, Klüwer Academic Publishers, 1993.

[43] Alfred J Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1996.

[44] Ralph E. Merkle, *Secrecy, authentication, and public key systems*, Ph.D. thesis, Stanford University, Dept. of Electrical Engineering, 1979.

[45] Peter L. Montgomery, *Square roots of products of algebraic numbers*, Mathematics of Computation 1943–1993: a half-century of computational mathematics (Walter Gautschi, ed.), American Mathematical Society, 1994, pp. 567–571.

[46] ———, *A block lanczos algorithm for finding dependencies over gf(2).*, EUROCRYPT, 1995, pp. 106–120.

[47] Francois Morain, *Implementation of the atkin-goldwasser-kilian primality testing algorithm*, Tech. Report RR-0911, Département de Mathématiques, Université de Limoges, 1988.

[48] Michael A. Morrison and John Brillhart, *A method of factoring and the factorization of F_7*, Mathematics of Computation **29** (1975), 183–205.

[49] Brian Antony Murphy, *Polynomial selection for the number field sieve integer factorisation algorithm*, PhD - thesis (1999).

[50] Winfried B Müller and Rupert Nöbauer, *Some remarks on public-key cryptosystems*, Sci. Math. Hungar **16** (1981), 71–76.

[51] ———, *Cryptanalysis of the dickson-scheme*, Advances in Cryptology - EUROCRYPT '85 - Lecture Notes in Computer Science **219** (1986), 50–61.

[52] Phong Nguyen, *A Montgomery-like square root for the number field sieve*, Lecture Notes in Computer Science **1423** (1998), 151–??

[53] Tatsuaki Okamoto and Shigenori Uchiyama, *A new public-key cryptosystem as secure as factoring*, Lecture Notes in Computer Science **1403** (1998), 308–318. MR 1 729 059

[54] R. G. E. Pinch, *Extending the Wiener attack to RSA-type cryptosystems*, Electronics Letters **31** (1995), no. 20, 1736–1738.

[55] S. Pohlig and Martin E. Hellman, *An improved algorithm for computing logarithms over gf(p) and its cryptographic significance*, IEEE Transactions on Information Theory **24** (1978), 106–110.

[56] John M. Pollard, *Theorems on factorization and primality testing.*, Proceedings of the Cambridge Philosophical Society **76** (1974), 521–528.

[57] ———, *Monte carlo method for factorization.*, BIT **15** (1975), 331–334.

[58] ———, *Monte carlo methods for index computation (mod p)*, Mathematics of Computation **32** (1978), 918–924.

[59] ———, *Lattice sieving*, Lecture Notes in Mathematics **1554** (1991), 43–49.

[60] Carl Pomerance, *The quadratic sieve factoring algorithm*, Advances in cryptology: EUROCRYPT '84 (Berlin) (Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, eds.), Lecture Notes in Computer Science, vol. 209, Springer-Verlag, 1985, pp. 169–182.

[61] Martin O. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, MIT Technical Report **TR-212** (1979).

[62] Hans Riesel, *Prime numbers and computer methods for factorization (2nd ed.)*, Birkhauser Verlag, Basel, Switzerland, Switzerland, 1994.

[63] Laura Toti Rigatelli, *Evariste galois 1811-1832 (english translation by john denton)*, Birkhäuser Verlag, 1996.

[64] Ron L. Rivest, Adi Shamir, and Len Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), 120–126.

[65] Ronald L. Rivest and Robert D. Silverman, *Are "strong" primes needed for rsa?*, 1998.

[66] Bruce Schneier, *The Blowfish encryption algorithm*, Dr. Dobb's Journal of Software Tools **19** (1994), no. 4, 38, 40, 98, 99.

[67] René Schoof, *Counting points on elliptic curves over finite fields*, Journal de Théorie des Nombres de Bordeaux **7** (1995), 219–254.

[68] Daniel Shanks, *Class number, a theory of factorization, and genera*, Proceedings of Symposia in Pure Mathematics **20** (1969), 415–440.

[69] Joseph H. Silverman and John Tate, *Rational point on elliptic curves*, Springer-Verlag, 1992.

[70] Simon Singh, *Fermat's last theorem*, Fourth Estate, 1997.

[71] Nigel P. Smart, *The discrete logarithm problem on elliptic curves of trace one*, Journal of Cryptology **12** (1999), no. 3, 193–196.

[72] Peter Smith and Michael J. J. Lennon, *LUC: A new public key system*, Tech. report, 1993.

[73] Ian Stewart, *Galois theory, second edition*, Chapman & Hall/CRC, 1989.

[74] Douglas R. Stinson, *Cryptography, theory and practice*, CRC Press, 1995.

[75] A. E. Western and J. C. P. Miller, *Tables of indices and primitive roots*, Royal Society Mathematical Tables **9** (1968), xxxvii–lxii.

[76] M. Wiener, *Cryptanalysis of short rsa secret exponents*, IEEE Transactions on Information Theory **36** (1990), 553–558.

[77] H. C. Williams and J. O. Shallit, *Factoring Integers Before Computers*, Mathematics of Computation 1943–1993: a half-century of computational mathematics (Providence) (Walter Gautschi, ed.), American Mathematical Society, 1991, pp. 481–531.

[78] Hugh C. Williams, *A modification of the rsa public-key encryption procedure*, IEEE Transactions On Information Theory **IT-26 No. 6** (1980), 726–729.

[79] ———, *A $p+1$ method of factoring*, Mathematics of Computation **39** (1982), 225–234.

[80] Malcolm J. Williamson, *Non-secret encryption using a finite field*, CESG report (classified until december 1997) (1974), 1–2.

[81] F. W. Winterbotham, *The ultra secret: The inside story of operation ultra, bletchley park and enigma*, Orion mass market paperback, 2000.