NUMBER FIELD SIEVE:
PSEUDOCODES AND SOFTWARE IMPLEMENTATION

by

Theodore Kemp Winograd

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

KGaj
Dr. Kris Gaj, Thesis Director

Dr. Jens-Peter Kaps, Committee Member

Brian L. Mark
Dr. Brian Mark, Committee Member

Dr. Andre Manitius, Department Chair
of Electrical and Computer Engineering

Dr. Lloyd J. Griffiths, Dean,
Volgenau School of Engineering

Date: 12/07/2011

Fall Semester 2011
George Mason University
Fairfax, VA

Number Field Sieve: Pseudocodes and Software Implementation

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Theodore Kemp Winograd
Bachelor of Science
Virginia Polytechnic Institute and State University, 2004

Director: Dr. Kris Gaj, Associate Professor
Department of Electrical and Computer Engineering

Fall Semester 2011
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my loving wife, Rachel, who let me start this degree two weeks after we got married and my daughter, Jocelyn, who was born during my final semester at GMU.

Acknowledgments

I would like to thank the following people who made this possible: Dr. Kris Gaj, Dr. Soonhak Kwon, Dr. Jens-Peter Kaps, Dr. Patrick Baier, and Dr. Brian Mark. I would also like to thank Per Leslie Jensen for permission to use his source code as well as Jason Papadopoulos for providing his guidance.

Table of Contents

List of Tables

List of Figures

Abstract

NUMBER FIELD SIEVE: PSEUDOCODES AND SOFTWARE IMPLEMENTATION

Theodore Kemp Winograd, M.S.

George Mason University, 2011

Thesis Director: Dr. Kris Gaj

The RSA cryptosystem has been the mainstay of modern cryptography since it was first introduced in 1978. RSA serves as the basis for securing modern e-commerce–it functions as the primary key exchange mechanism for the Secure Sockets Layer (SSL) protocol. It is used by US Government Personal Identity Verification (PIV) smart cards and the Department of Defense Common Access Card (CAC) for authenticating users, digitally signing and encrypting email.

Due to the importance of this algorithm, cryptanalysts have been working for decades to identify weaknesses in the algorithm. Because the security of the RSA algorithm rests on the computational infeasibility of factoring large numbers, a good deal of research has been in the field of factorization. Of note was the introduction of the Number Field Sieve in 1993, which remains the fastest known algorithm for factoring large numbers.

One of the most difficult aspects of the Number Field Sieve is the complexity of the algorithm, requiring a great deal of number theory simply to understand how the individual steps of the algorithm function. To this end, there are very few implementations of the algorithm that are coupled with concise and detailed descriptions of the algorithm. This thesis describes an implementation of the Number Field Sieve implemented using C++ in a straightforward manner–leaving efforts to improve this particular implementation as future

work. Based on the implementation, the author was able to derive a set of pseudocodes that can be provided to students to gain a full understanding of the number field sieve algorithm.

Finally, this thesis performs a number of experiments on this implementation–as well as other open source implementations that have been developed in the past few years. This thesis aims to identify the trade-offs within the algorithm that can be made based on the wide variety of parameters that can be applied. While some of these trade-offs are to be expected (e.g., the performance impact of using a lattice sieve over a line sieve), a more detailed understanding of the various options will aid both implementers and students in improving software implementations and–where possible–identifying methods for breaking the number field sieve algorithm into components and identifying which components are best implemented in hardware and which components are best implemented in software.

Chapter 1: Introduction

Cryptography is the art of secret writing. Throughout the centuries, there has been an arms-race between those who create cryptosystems, the *the code makers* and those who break cryptosystems *the code breakers*. Cryptosystems began as relatively simple mechanisms to hide the meaning of a particular message from prying eyes; in ancient times, little effort was required to do this as most people were illiterate and so unable to read even the plaintext of a message–let alone the ciphertext.

One of the first known cryptosystems, the Caesar cipher, "replaced each letter in a message with the letter that is three places further down the alphabet."[1] With this simple cipher, the content of the message was unreadable except by a knowledgeable observer. Of course, the Caesar cipher is an example of an algorithm that fails the fundamental tenet of cryptography: that the security of the cipher depends only upon the security of the cryptographic key. In the case of the Caesar cipher, knowledge of the algorithm itself is all that is required to decipher it. Similarly, the Caesar cipher is also an example of an algorithm that does not protect against frequency analysis: when the cipher is used to encrypt a sufficiently long message, attackers may notice that certain letters that are common within the English language (e.g., e, t and a) have been replaced by letters three places further down (e.g., h, w and d). To resolve this issue, cryptographers created new cryptosystems–ones that were immune to this simple form of cryptanalysis.

This has lead to a long arms race between cryptographers and cryptanalysis, with new cryptographic algorithms and new methods of cryptanalysis being developed over time. For hundreds of years, cryptographers have created new ciphers and cryptanalysts have created new methods for analyzing cryptographic algorithms. Figure 1.1 shows a brief history of cryptographic algorithms and cryptanalytic methods. Before the advent of computers, cryptographic algorithms were required to be simple enough that a human could easily encrypt

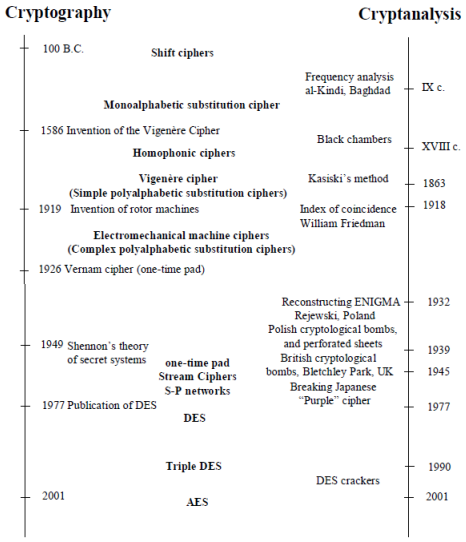

Figure 1.1: Cryptography Timeline [2, p2]

and decrypt the data. Over time, cryptographers developed mechanical devices that would perform the necessary computations on behalf of the user–only requiring knowledge of the key for encryption and decryption. One of the most famous examples of such cryptographic devices is the German ENIGMA machine. One such cryptographic device, the Lorenz, was ultimately defeated by the precursor to digital computing devices: the *Colossus*[3].

Once digital computing devices became commonplace for breaking cryptographic ciphers, cryptographers began using them to create new cryptosystems. Such cryptosystems are the precursors to those that are used today. In fact, the Data Encryption Standard (DES) [4] (first released in 1977) is still in use today as the 3DES cryptosystem–and still considered to be secure.

While classical symmetric encryption algorithms proved to be a powerful force in designing and implementing strong cryptosystems, they still required knowledge of a secret key to be passed from one party to another. In previous eras, these cryptographic keys would simply be a string of letters and numbers memorized on a piece of paper. However,

with modern cryptosystems like DES requiring 56-bit keys–and computers' ability to process large amounts of data, consistently transferring cryptographic keys using out-of-band methods became a problem in and of itself. In fact, the downfall of the ENIGMA machine was primarily due to the fact that German key distribution was *not* out-of-band! This lead to the introduction of public-key cryptography by Whitfield Diffie and Martin Hellman in 1976 [5]. The Diffie-Hellman key exchange allowed for distribution of cryptographic keys to occur over an insecure channel, making it much simpler to secure communication across computing networks. Shortly thereafter, Ronald Rivest, Adi Shamir and Leonard Adleman introduced the RSA algorithm in 1978 [6] (discussed in Section 2.2).

RSA has become the backbone of modern e-commerce: it is the basis of the Secure Sockets Layer (SSL) protocol used to secure online purchases, it is used by US government Personal Identity Verification (PIV) and Department of Defense (DoD) Common Access Card (CAC) for digitally signing and encrypting email as well as for authenticating to US Government Web sites. To this end, a large chunk of cryptanalysis research has been performed against the RSA algorithm. Of particular note is that the RSA algorithm *assumes* that factoring large numbers is a computationally infeasible problem. In particular, in 1993 Buhler, Lenstra and Pomerance introduced the Number Field Sieve, the fastest known algorithm for factoring large numbers. To that end, a large amount of research has been done to develop efficient implementations of the Number Field Sieve (see Section 3).

1.1 Goals of this Thesis

When research for this thesis first began in 2007, it was intended to be an analysis of various different open source implementations of the NFS algorithm:

- Chris Monico's *GPL'd Implementation of the General Number Field Sieve* [7]
- Per Leslie Jensen's *Pleslie's Number Field Sieve* [8]
- Chris Card's *factor-by-gnfs* [9]

- Jason Papadopoulos' *msieve* [10]

Results of running each of these different implementations quickly showed that the various design choices available in NFS were being applied differently across implementations. To truly describe the effects of these various design choices and compare the results, it became apparent that a full discussion of the NFS algorithm would be necessary. While such discussions have been provided over the years (see [11], [12], [8] and [13] are but a few examples), few have provided both the pseudocode necessary and tests required to validate an implementation of NFS. As such, the direction of this thesis turned to developing a set of pseudocodes and discussion of validations that folks can use to develop an implementation of NFS. This thesis aims to improve upon the efforts put forth by [8] and add insight into alternative algorithms for each of the components of NFS. Nevertheless, there are many improvements that need to be made for this implementation to reach the level of performance and stability attained by other open source implementations of NFS. The author hopes that this implementation can serve as a good learning tool for future Masters' students to have a good framework for developing implementations of NFS. To that end, part of this thesis discusses efforts underway to integrate this implementation with the CrypTool [14] as well as a cross-platform graphical user interface developed specifically for this implementation.

Chapter 2: Background

Factoring large numbers is considered to be a computationally difficult problem. There are a number of different techniques, each with an increasing level of complexity and speed, that solve the problem. However, mathematicians have yet to come up with a method of factoring numbers in *polynomial* time. In fact, a large number of factorization algorithms complete in *exponential* time! This chapter introduces the mathematics required for the rest of this paper as well as the concept of asymmetric cryptography (specifically the RSA algorithm).

2.1 Mathematical Background

This section discusses the mathematical background necessary to understand the rest of this thesis. One of the goals of this thesis is to minimize the amount of mathematics required to implement the number field sieve in an effective fashion, but there will always be a set of mathematical concepts that will be required.

2.1.1 Groups

From [15, p75]: A *group* $(G, \cdot)$ consists of a set G with binary operation $\cdot$ on G satisfying the following three axioms.

1. The group operation is *associative*. That is, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in G$.
2. There is an element $1 \in G$, called the *identity element*, such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in G$.
3. For each $a \in G$ there exists an element $a^{-1} \in G$, called the *inverse* of a, such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

A group G is *abelian* if $a \cdot b = b \cdot a$ for all $a, b \in G$.[15, p75]

A group G is *finite* if $|G|$ is finite, with the number of elements referred to as the group's *order*. [15, p75]

A group can also be written in a different convention, $(G, +)$, where the binary operation $+$ satisfies the three axioms as follows:

1. The group operation is *associative*: $a + (b + c) = (a + b) + c$ for all $a, b, c \in G$.
2. There is an identity element $0 \in G$, such that $a + 0 = 0 + a = a$ for all $a \in G$.
3. For each $a \in G$ there exists an inverse element $-a \in G$, such that $-a+a = -a+a = 0$.

2.1.2 Rings

A ring $(R, \bigoplus, \bigodot)$ consists of a set R with two binary operations arbitrarily denoted $\bigoplus$ (addition) and $\bigodot$ (multiplication) on R, satisfying the following axioms.[15, p76]

1. $(R, \bigoplus)$ is an abelian group with identity denoted 0. [15, p76]
2. The operation $\bigodot$ is associative (e.g., $a \bigodot (b \bigodot c) = (a \bigodot b) \bigodot c$ for all $a, b, c \in R$) [15, p77]
3. There is a multiplicative identity 1, with $1 \neq 0$ such that $1 \bigodot a = a \bigodot 1 = a$ for all $a \in R$ [15, p77]
4. The operation $\bigodot$ is distributive over $\bigoplus$: $a \bigodot (b \bigoplus c) = (a \bigodot b) \bigoplus (a \bigodot c)$ and $(b \bigoplus c) \bigodot a = (b \bigodot a) \bigoplus (c \bigodot a)$ for all $a, b, c \in R$ [15, p77]

The ring is a *commutative ring* if $a \bigodot b = b \bigodot a$ for all $a, b \in R$. [15, p77]

2.1.3 Fields

A *field* is a commutative ring in which all non-zero elements have multiplicative inverses. [15, p77]

The *characteristic* of a field is 0 if $\overbrace{1+1+\cdots+1+1}^{m \text{ times}}$ is never equal to 0 for any $m \geq 1$. Otherwise, the characteristic of the field is the least positive integer m such that $\sum_{i=1}^{m} 1$ equals 0. [15, p77]

A subset **F** of a field **E** is a *subfield* of **E** if **F** is itself a field with respect to the operations of **E**. If this is the case, **E** is also referred to as an *extension field* of **F**. [15, p77]

2.1.4 Sets of Polynomials

According to [15, p78], if **R** is a commutative ring, then a *polynomial* in the indeterminate x over the ring **R** is an expression of the form: $f(x) = a_n \cdot x^n + \cdots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ where each $a_i \in \mathbf{R}$ and $n \geq 0$.

It follows that if **R** is a commutative ring then the *polynomial ring* $\mathbf{R}[x]$ is the ring formed by the set of all polynomials in the indeterminate x having coefficients from **R**. The two operations are polynomial addition and polynomial multiplication, with coefficient arithmetic performed in **R**. [15, p78]

For example, $(\mathbf{Z}_2[x], \bigoplus)$ is an abelian group with an identity element 0. Similarly, $(\mathbf{Z}_2[x], \bigodot)$ is commutative with an identity element 1. Finally, $\bigodot$ is distributive over $\bigoplus$– meaning that $(\mathbf{Z}_2[x], \bigoplus, \bigodot)$ is a polynomial ring. [16]

Performing division on a polynomial is performed as follows: $g(x) = q(x) \cdot h(x) + r(x)$ where $deg(r(x)) < deg(h(x))$.

An *irreducible* polynomial over a field $\mathbf{F}[x]$ must have a degree of 1 and *cannot* be written as a product of two polynomials in $\mathbf{F}[x]$. [15, p78]

$\mathbf{F}[x]/f(x)$ refers to the set of polynomials whose degree is less than $deg(f(x))$ with addition and multiplication performed modulo $f(x)$. This also means that $\mathbf{F}[x]/f(x)$ is a commutative ring. If $f(x)$ is irreducible, then $\mathbf{F}[x]/f(x)$ is a field!

2.1.5 Finite Fields

A *finite field* is a field $\mathbf{F}$ that contains a finite number of elements and its *order* is the number of elements in $\mathbf{F}$. A finite field has the following properties: [15, p80]

1. The order of $\mathbf{F}$ is p^m for p prime and integer $m \geq 1$.

2. For each prime p, there is a unique finite field with an order p^m called $\mathbf{F}_{p^m}$ or $GF(p^m)$.

Finite fields are also referred to as *Galois Fields* because they were originally described by Évariste Galois.

2.1.6 Primitive Element

For a particular finite field $\mathbf{F}_q$, there is a multiplicative group denoted as $\mathbf{F}_q^*$. The group $\mathbf{F}_q^*$ is cyclic with an order of $q-1$. This means that $a^q = a$ for all $a \in \mathbf{F}_q$. There also exists an element of $\mathbf{F}_q^*$ for which each power (e.g., $a^1, a^2, a^3, ..., a^{q-1}$) results in a unique element of $\mathbf{F}_q^*$; this is the *primitive element* or *generator* of $\mathbf{F}_q^*$. [15, p81]

2.1.7 Primitive Polynomial

If x is a primitive element for $\mathbf{F}_{p^m}^*$, then the irreducible polynomial $f(x) \in \mathbf{Z}_p[x]$ of degree m is a *primitive polynomial* of the group $\mathbf{F}_{p^m} = \mathbf{Z}_p[x]/f(x)$. [15, p84].

2.1.8 Chinese Remainder Theorem

The Chinese Remainder Theorem allows for the deconstruction of a congruence mod n into a system of congruences mod factors of n. The Chinese remainder theorem is as follows: [17]

$n \leftarrow n_1 \cdot n_2 \cdots n_M$

for any i, j: $gcd(n_i, n_j) = 1$

for each A such that $0 \leq A \leq N-1$ **do**

$A \leftrightarrow (a_1 = A \mod n_1, a_2 = A \mod n_2, \ldots, a_M = A \mod n_M)$

end for

With the proper values of a_i and n_i, A can be reconstructed using (2.1). [17]

$$A = \sum_{i=1}^{M} \frac{a_i \cdot N}{n_i} \tag{2.1}$$

For example, given a number $n \equiv 26 \mod 99$ we can also construct a set of congruences that are equivalent: [18, p72]

$$n \equiv 26 \mod 99 \Rightarrow \begin{cases} n \equiv 8 \mod 9 \\ n \equiv 4 \mod 11 \end{cases} \tag{2.2}$$

Using the Chinese Remainder Theorem, we can calculate $n \equiv 26 \mod 99$ from the two congruences shown in (2.2).

The two congruences $a \mod m$ and $b \mod n$ can ge solved if $gcd(m, n) = 1$. Thus we solve $b + nk \equiv a \mod m$ and calculate $x \mod mn \equiv b + nk \mod mn$. Using the example in (2.2), $gcd(9, 11) = 1$, so this can be solved as: [18, p73]

$$4 + 11 \cdot k = 4 + 11 \cdot 2 \equiv 4 \mod 11$$

$$x \equiv 4 + 11 \cdot 2 \mod (9 \cdot 11) \equiv 26 \mod 99$$

2.2 Asymmetric Cryptography

In 1978, Rivest, Shamir and Adleman introduced the RSA algorithm. It is an implementation of asymmetric cryptography in which a user, *Alice* has both a public key and a private key. The public key, as implied by its name, can be distributed throughout the world granting anyone, *Bob*, the ability to encrypt a message and send it to the user. Once the message is encrypted, it is computationally infeasible for a third party, *Eve*, to decrypt the message.

In fact, even Bob is unable to decrypt the message. Only through knowledge of the secret key can Alice decrypt the message.

This is accomplished through the use of a trap door function. In mathematics there are a number of operations and functions that are computationally infeasible in general, but in special cases they are easy to perform. For example, factoring a large number is computationally infeasible. The RSA algorithm takes advantage of this fact and uses Euler's theorem and Euler's Totient function to provide a trap door mechanism.

The Totient function is described as follows: [18, p77]

Let $\phi(n)$ be the number of integers $1 \leq a \leq n$ such that $gcd(a, n) = 1$.

Euler's Theorem can be described as follows: [18, p77]

if $gcd(a, n) = 1$ **then**

$a^{\phi(n)} \equiv 1 \mod n$

end if

In general, calculating the Totient of a composite number requires factoring that number, because the Totient function is multiplicative: $\phi(m \cdot n) = \phi(m) \cdot \phi(n)$. [18, p77] Additionally, the Totient of a prime number is very easy to calculate: $\phi(p) = p - 1$. [18, p77]

With this knowledge, Rivest, Shamir and Adleman constructed a cryptosystem: [18, p138]

Alice chooses secret primes p and q and computes $n = p \cdot q$

Alice chooses e with $gcd(e, \phi(n)) = 1$

Alice computes d with $d \cdot e \equiv 1 \mod (\phi(n))$

Alice makes n and e public, and keeps p, q, d secret.

Bob encrypts m as $c \equiv m^e \mod n$ and sends c to Alice

Alice decrypts by computing $m \equiv c^d \mod n$

When Bob encrypts the message m, he gets $c \equiv m^e \mod n$. Alice receives this message and calculates c^d, which is equivalent to $(m^e)^d \mod n \equiv m^{ed} \mod n$. From the algorithm,

$e \cdot d \equiv 1 \mod (\phi(p) \cdot \phi(q))$, so $m^{ed} \mod n$ can be written as $m^{1+k \cdot \phi(p) \cdot \phi(q)} \mod n \equiv m^{1+k \cdot \phi(n)}$. This can be rewritten as $m \cdot (m^{\phi(n)})^k \mod n \equiv m \cdot 1^k \mod n \equiv m \mod n$. [18, p140]

Factoring n would allow an attacker to easily decipher the message n, because it would provide the missing elements required to calculate d. Based on the algorithm, $d \equiv e^{-1} \mod \phi(n)$ because $gcd(e, \phi(n)) = 1$. By constructing n such that Alice knows $\phi(n)$ it is trivial for Alice to perform the necessary calculations and decipher the message!

Due to the power of public key cryptography, the RSA algorithm has grown to become one of the most important factors of e-commerce. The Transport Layer Security (TLS) [19] algorithm uses RSA to authenticate users and computers as well as to agree upon and distribute symmetric keys to initiate secure communication channels over untrusted networks (e.g., the Internet). Due to the importance of the RSA algorithm, RSA launched the RSA challenge, [20] in which prizes were awarded to teams who successfully factored a series of larger composite numbers. The most recent factorization was the RSA-768 challenge number in 2010. By performing a linear regression on previous factorization records, see Figure 2.1, a 1024-bit number will be factored by 2028. will be factored by the year 2028. To this end, NIST issued guidance on acceptable key sizes for use within the

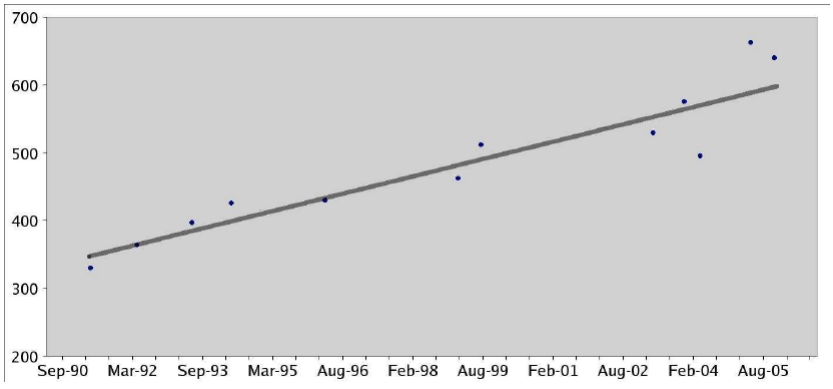

Figure 2.1: Linear Regression of Factorization Records [21]

Federal government: 1024-bit keys were considered secure through 2010, 2048-bit keys are secure through 2030 and 3072-bit keys are considered secure past 2030. [22]

2.3 Factoring

Factoring is a computationally hard problem. Conceptually, the easiest factoring method is trial division, which has the following algorithm:

```
Choose a composite number n to factor
Choose an integer i = 2
for i = 1 → ⌊√n⌋ do
  if n mod i = 0 then
    return i
  end if
end for
return 1
```

There are some improvements that can be made to this algorithm, but it is the least efficient algorithm for factoring numbers. To that end a number of different algorithms are available to factor large numbers, each with an increasing level of complexity.

A large number of these algorithms are rooted in the *Fermat factorization* method[18, p149], which states that n can be written as a difference of two squares: $n = x^2 - y^2$. Thus, n can be rewritten as $n = (x + y)(x - y)$. Finding the factors of n can be performed using the following algorithm:

```
for i = 1 → n do
  a ← n + i^2
  b ← ⌊√a⌋
```

```
    if a = b^2 then
      return i
    end if
  end for
  return 1
```

This method works because where $n = x^2 - y^2$ can also be written as $n + y^2 = x^2$. As such, finding a value of y such that $n + y^2$ is square is provides the values of x^2 and y^2 such that $n = x^2 - y^2$. Thus, the factors of n could be written as $n = (\sqrt{x^2} + \sqrt{y^2})(\sqrt{x^2} - \sqrt{y^2})$.

Like trial division, this method of factorization can take an inordinate amount of time to complete. To that end, a number of algorithms exist that aim to simplify finding the two squares, $x^2 + y^2$ and the corresponding factors, $(x + y)$ and $(x - y)$. Specifically, the quadratic sieve and the number field sieve use this method of factorization.

Another method is Pollard's ρ. In the ρ method, one looks for a cycle in a randomly generated formula, $f(x)$. For example, the formula $x_{i+1} = x_i^2 + a \mod n$ will usually form a cycle for any starting value x_0 [23]. The length of time it takes x_n to become cyclic is proportional to $\sqrt{n}$. Through the Chinese remainder theorem, $x_i \mod n$ is equivalent to both $x_i \mod p$ and $x_i \mod q$ [23]. Substituting these values into the function gives:

$$x_{i+1} = x_i^2 + a \mod p x_{i+1} = x_i^2 + a \mod q$$

As such, the function will fall into shorter cycles of lengths proportional to $\sqrt{p}$ and $\sqrt{q}$. Similarly, if $gcd(x_{i+1} - x_i, n) = p$ then x_{i+1} and x_i have the same value $\mod p$ and that p is a factor of n [23].

Pollard's algorithm can be summarized as follows: [24]

$b \leftarrow c \leftarrow 2$

Choose random a

repeat

$f(x) \leftarrow x^2 + a$

$b \leftarrow f(b) \mod n$

$c \leftarrow f(f(c)) \mod n$

$d \leftarrow gcd(b - c, n)$

until $d \neq N$ and $d \neq 1$

d is a non-trivial factor of n

There are a wide variety of algorithms available for factoring large numbers. In fact, the minifactoring step of the number field sieve (see Section 4.5) leverages these algorithms. It is important to note that while none of these methods are as fast as the number field sieve for factoring *large* numbers, all of them are faster than the number field sieve for factoring *numbers that are not large*

Chapter 3: Previous Work

There has been a good deal of previous work on the Number Field Sieve since it was introduced in 1993 [25]. As discussed in Section 1.1, one of the goals of this thesis is to synthesize much of the information currently available on the topic into a single document, easing the barrier to entry for new players who are interested in improving the performance of Number Field Sieve implementations.

3.1 Overviews of the Number Field Sieve

In 1988, Pollard introduced the idea of the Number Field Sieve in his paper *Factoring with cubic integers* [26]. In addition, the volume *The development of the number field sieve* was published in 1993. [27] Since then, a number of different papers have been published to provide an overview of the Number Field Sieve.

- In 1993, H. Lenstra Jr. provided the first full description of the Number Field Sieve in [11] with a description of the implementation by Bernstein in [28].
- In 1996, Huizing published a paper describing a software implementation of the Number Field Sieve. [29]
- In 1998, as part of his Masters Thesis, Matt Briggs put together a more in-depth overview of the Number Field Sieve, providing a set of examples so that the reader may follow through the process. [12]
- In 2008, Ekkelkamp published a paper on *Predicting the Sieving Effort for the Number Field Sieve* [30]. This paper provided an overview of the number field sieve as well as a new method for estimating the amount of time it will take for the Number Field Sieve to run through for a set of parameters.

3.2 Algorithmic Improvements

After the introduction of the number field sieve, a number of researchers have worked to improve its efficiency. While the asymptotic efficiency of the algorithm has not been improved substantially–it is still a sub-exponential factoring algorithm, these efforts combined with improvements in computational speed have made it so that the number field sieve is increasingly capable of factoring larger and larger numbers, with the potential to factor a 1024-bit number within the next twenty years.

3.2.1 Polynomial Selection

- In 1993, polynomial selection was first described in [11] by H. Lenstra, Jr. using base-m expansion. See Section 5.4 for a discussion of the algorithm.
- In 1999, Brian Murphy developed an influential paper on polynomial selection for the Number Field Sieve as part of his PhD work. Specifically, he introduced the α approximation, which provides an heuristic for determining the effectiveness of a particular polynomial. [31]
- In 2006, Kleinjung improved upon Murphy's work, describing new methods for choosing polynomials appropriate for the Number Field Sieve. [32]

3.2.2 Sieving

- Line sieving was first described in [11] by H. Lenstra Jr.
- A more detailed example of the line sieving algorithm was provided by Geiselmann. [33]
- In 1993, Pollard describes the Lattice sieving method [34]. This is traditionally considered an improvement over Line Sieving methods and is the basis of most modern implementations of the Number Field Sieve.

- In 1994, Golliver, A. K. Lenstra and McCurley improved upon the Lattice sieving method using trial division. [35]

- In 2005, Franke and Kleinjung improved upon the Lattice sieving method, using continued fractions to improve its performance and distribute it across multiple processes. [36]

3.2.3 Matrix Step

- In 1990, Wiedemann introduced an algorithm for solving sparse linear equations over finite fields. [37]

- In 1994, Coppersmith improved upon the Wiedemann algorithm and devised the Block Wiedemann algorithm which provides a faster method for solving sparse linear equations over finite fields. [38]

- In 1995, Montgomery introduced the Block Lanczos algorithm, which can be used to solve systems of linear equations in a parallel manner using an improvement of the Lanczos algorithm, originally developed in 1950. [39]

3.2.4 Square Root Step

- In 1993, H. Lenstra, Jr. introduced the original square root step. [11]

- In 1993, Couveignes introduced a slightly different method of performing the square root step than that introduced by Lenstra and discussed by Briggs [12] and Jensen [8]. [40] Couveignes' method is used in this thesis.

3.3 Software Implementations

There have been a number of software implementations published under Open Source licenses:

- Chris Monico's *GPL'd Implementation of the General Number Field Sieve* [7]: provides a full implementation of multiple elements of the algorithm. Specifically, it includes Franke's implementation of Lattice sieving.

- Per Leslie Jensen's *Pleslie's Number Field Sieve* [8]: provides an easy-to-understand implementation of the number field sieve.

- Chris Card's *factor-by-gnfs* [9]: provides a full implementation of the number field sieve written in C++ with optimizations.

- Jason Papadopoulos' *msieve* [10]: a full implementation of the number field sieve that is capable of working together with GGNFS. The GGNFS/msieve combination has been used to successfully factor numbers up to 180-digits.

There are additional software implementations that have been used and discussed since the advent of the number field sieve, including:

- In [11], H. Lenstra, Jr. et al. introduced an implementation of the number field sieve.

- In 2005, Anand developed implementations of the Wiedemann and Block Wiedemann algorithms, which are commonly used to implement the linear algebra step. [41]

The majority of implementations that have been used to factor RSA challenge values have been closed source. While these are important implementations for the development of improvements to the algorithm, they raise the barrier to entry for new researchers into the field.

3.4 Hardware Implementations

The CAIRN [42] hardware implementation was used by the author to get a better grasp of the Number Field Sieve algorithm. The CAIRN implementation has seen numerous improvements with CAIRN 1 [43], CAIRN 2 [44], and CAIRN 3 [42].

The CAIRN implementations are among the few practical implementations of the Number Field Sieve. Other implementations, focused primarily on the mini-factoring step, include:

- Khaleeluddin Mohammed's implementation of Elliptic Curve factoring for the mini-factoring step of the number field sieve [45]
- An FPGA implementation of trial division by small primes by GMU [46]
- Ramakrishna Bachimanchi's FPGA implementations of Rho and P-1 methods of factoring for the mini-factoring step of the number field sieve [24]
- An implementation of the mini-factoring step in hardware by GMU [47]

A number of theoretical designs exist:

- In 1999 and 2000, Shamir introduced TWINKLE [48], which is based on optoelectronic devices (fast LEDs) [21].
- In 2003, Shamir and Tromer introduced TWIRL [49], which leverages fast communication between chips located on the same 30 cm diameter wafer [21].
- In 2003 and 2004, Geiselmann and Steinwandt introduced Mesh Based Sieving [50], which aimed to improve upon the performance of TWINKLE using mesh-based circuits to improve the latency between hardware components.
- In 2005, Franke et. al introduced SHARK [51], which relies on an elaborate butterfly switch connecting large number of chips [21].
- In 2007, Geiselmann and Steinwandt introduced the Non-Wafer Scale Sieving Hardware [33], which is based on moderate sized chips and implements line sieving rather than lattice sieving.

Chapter 4: Pseudocode

This chapter describes the Number Field Sieve and presents a set of psuedocodes that describe the function of the algorithm. By breaking the algorithm into its discrete components, the reader can better understand the steps necessary to develop a full implementation of the number field sieve. Chapter 5 details the implementation that was developed based on the pseudocodes presented in this chapter.

4.1 Number Field Sieve Overall Algorithm

The Number Field Sieve is a complex algorithm, but it has a succinct number of steps–with each step getting more complex as the algorithm processes data. The steps are:

1. Polynomial selection
2. Factor base generation
3. Sieving
4. Mini-factoring
5. Linear algebra
6. Square root

Each step is covered in its own section within this chapter. However, prior to launching the NFS algorithm, the following inputs must be provided by the user:

1. n: A large composite number to be factored
2. $B_{algebraic\_fudge}$ and $B_{rational\_fudge}$: *Fudge* factors that are used in the sieving step

3. A: The range of values of a to sieve

4. c : The multiplication factor to determine how many values of b to sieve at a time.

The following listing describes the overall Number Field Sieve algorithm, including additional portions of the linear algebra step. While these portions are illustrated in this listing, they are discussed in detail in Sections 4.2, 4.3, 4.4, 4.5, 4.6 and 4.9.

INPUTS:

n : the number to factor

A : boundary of the sieving range for a

$B_{algebraic\_fudge}$: an empirical fudge factor used to determine if a value is likely smooth over the algebraic factor base

$B_{rational\_fudge}$: an empirical fudge factor used to determine if a value is likely smooth over the algebraic factor base

c : the multiplication factor for use in the equation to determine the number of values of b to be tested.

NOTE: $B_{algebraic\_fudge}$ and $B_{rational\_fudge}$ are independent of a and b in this implementation, but need not be.

OUTPUTS:

q: a non-trivial factor of n

VARIABLES:

$f(x)$: polynomial [for the algebraic side]

d : degree of polynomial $f(x)$

m : root of polynomial $f(x) \mod n$, i.e. an integer, such that $f(m) \mod n = 0$

RFB : rational factor base

AFB : algebraic factor base

RFB and AFB are sets of pairs (p, r) fulfilling conditions specific to the given base.

QCB : quadratic character base

QCB is a set of pairs (q, s) fulfilling conditions specific to the given base.

We denote the numbers of such pairs by $\#RFB$, $\#AFB$, $\#QCB$, respectively.

$relations$: set of pairs (a, b) obtained after sieving the number of pairs in this set is denoted by $\#relations$

$good\_relations$: set of pairs (a, b) that represent vectors in $matrix$

$good\_pairs$: the set of pairs (a, b) that are passed to the sieving step

$matrix$: binary matrix used in the linear algebra step

V_{zero} : The zero vector.

$solutions$: set of solutions to the equation $matrix \cdot solution = V_{zero}$

$solution$: binary vector of the size $(\#relations)$ fulfilling the equation $matrix \cdot solution = V_{zero}$

PSEUDOCODE:

BEGIN

{The values for B, d and m are nominal and may be changed based on empirical evidence}

$(m, f(x)) \leftarrow polynomial\_selection(n)$

$(RFB, AFB, QCB) \leftarrow factor\_bases(n, m, f(x))$

$\#relations\_required \leftarrow 1 + \#RFB + \#AFB + \#QCB$

$B_{max} \leftarrow -1$

$\#good\_relations \leftarrow 0$

$matrix \leftarrow$ an empty matrix

while $(\#good\_relations < \#relations\_required)$ **do**

$B_{min} \leftarrow B_{max} + 1$

$B_{max} \leftarrow (\#relations\_required - \#good\_relations) \cdot c + B_{max}$

$relations \leftarrow line\_sieving(n, f(x), m, RFB, AFB, QCB, A, B_{min}, B_{max}, B_{algebraic\_fudge},$

$B_{rational\_fudge})$

{Build the matrix using the existing numbers}

$(matrix, relations\_found, \#relations\_found) \leftarrow mini\_factoring(matrix,$

$\#relations\_required, relations,$

$m, f(x), RFB, AFB, QCB$

add $relations\_found$ to $good\_relations$

$\#good\_relations \leftarrow \#good\_relations + \#relations\_found$

end while

$(matrix, freecols) \leftarrow solve\_matrix(matrix)$

{Identify the number of free variables that we have}

$freevals \leftarrow 0$

for $(k \leftarrow 0; k < size(freecols); k \leftarrow k + 1)$ **do**

if $(freecols[k] == 1)$ **then**

$freevals \leftarrow freevals + 1$

end if

end for

for $(numSols \leftarrow 1; numSols < freevals; numSols \leftarrow numSols + 1)$ **do**

$x \leftarrow GetSolutionSet(matrix, freecols, numSols)$

for $(i \leftarrow 0; i < size(x); i \leftarrow i + 1)$ **do**

if $(x[i] == 1)$ **then**

Add $good\_relations[i]$ to the list of $good\_pairs$

```
    end if
  end for
  factor ← sqrt_step(good_pairs, f, n, m)
  if (factor > 1) then
    return factor
  end if
 end for
END
```

The goal of the Number Field Sieve is to calculate two congruent squares such that the following statement is true:

$$x^2 \equiv y^2 \mod n, 0 \leq x \leq y \leq n, x \neq y, x + y \neq n \tag{4.1}$$

Where n is the number to be factored. If the above statement is true, then $x^2 \equiv y^2 \mod n$ and, by definition, $n \mid x^2 - y^2$ and $n \mid (x + y) \cdot (x - y)$. Assuming $n = p \cdot q$, then the following is also true: $x^2 \equiv y^2 \mod pq$ and $pq \mid (x + y) \cdot (x - y)$. Thus, p divides either $(x + y)$ or $(x - y)$ and describes one of the factors of n.

These squares are identified using polynomials. In particular, using Proposition 2.4.1 from [12], "Given a monic, irreducible polynomial $f(x)$ with integer coefficients, a root $\theta \in \mathbf{C}$ of $f(x)$, and an integer $m \in \mathbf{Z}/n\mathbf{Z}$ for which $f(m) \equiv 0 \mod n$, the mapping $\phi : \mathbf{Z}[\phi] \to \mathbf{Z}/n\mathbf{Z}$ with $\phi(1) \equiv 1 \mod n$ and which sends ϕ to m is a surjective ring homomorphism."

From [12], if there is a set U of pairs of integers (a, b) such that:

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

and

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

If $\beta \in \mathbf{Z}[\theta]$ and $y \in \mathbf{Z}$ and we let $\phi(\beta) = x \in \mathbf{Z}/n\mathbf{Z}$, then:

$$x^2 \equiv \phi(\beta)^2 \equiv \phi(\prod_{(a,b)\in U} (a + b\theta))$$

$$\equiv \prod_{(a,b)\in U} \phi(a + b\theta) \equiv \prod_{(a,b)\in U} (a + bm) \equiv y^2 \mod n$$

Thus, finding a monic, irreducible polynomial $f(x)$ with integer coefficients meeting these needs can be performed using one of the following algorithms.

Throughout this discussion, the number 45113 will be factored with the following parameters:

- $n = 45113$
- $B_{algebraic\_fudge} = 0$
- $B_{rational\_fudge} = 0$
- $A = 1000$
- $c = 50$

The number 45113 is used to parallel the example in [12], both showing that this implementation of the number field sieve is valid as well as providing a check against the implementation.

4.2 Polynomial Selection

Polynomial selection is one of the most important aspects of the number field sieve–the polynomial is used to choose the (a, b) pairs that will ultimately be used to generate the

factorization of n. To that end, a great deal of research has been performed in identifying good polynomials. One of the most important such documents is Murphy's PhD thesis [31], which introduced Murphy's *alpha approximation*, an heuristic for estimating how effective a given polynomial will be in producing valid (a, b) pairs without empirical testing (e.g., running the sieving step against the given polynomial to determine how efficient it is in creating (a, b) pairs). In generally, a smaller value of $\alpha(f(x))$ indicates a better polynomial.

Murphy's Alpha Approximation

$$\alpha = \sum_{p_{prime} \leq B} (1 - q_p \frac{p}{p+1}) \frac{\log p}{p-1}$$

return α

Where B is the smoothness bound, and q_p is the number of distinct roots of $f(x)$ mod p.

There are a number of different methods for generating a polynomial described in the literature–due to time restrictions the implementation discussed in this thesis is limited to the simplest polynomial selection algorithm described in [52].

INPUTS:

n : the number to factor

OUTPUTS:

$f(x)$: polynomial [for the algebraic side]

m : root of polynomial $f(x)$ mod n, i.e. an integer, such that $f(m) \mod n = 0$

PSEUDOCODES:

VERSION 1 (based on Crandall and Pomerance)

`http://books.google.com/books?id=RbEz-_D7sAUC&lpg=PP1&pg=PA293#v=onepage&q&f=false`

BEGIN

$d \leftarrow \lfloor \frac{3 \cdot log(n)}{log(log(n))}^{\frac{1}{3}} \rfloor$

if d is not acceptable **then**

choose d

end if

$m \leftarrow n^{\frac{1}{d}}$

if m is not acceptable **then**

choose m

end if

Write n in base m: $n = m^d + c_{d-1} \cdot m^{d-1} \cdots + c_0 \cdot m^0$

$f(x) = x^d + c_{d-1} \cdot x^{d-1} \cdots + c_0 \cdot x^0$

return $(m, f(x))$

END

VERSION 2 (based on Jensen, p. 61)

`http://pgnfs.org/DOCS/thesis.pdf`

BEGIN

choose m_0 such that $\lfloor n^{\frac{1}{d+1}} \rfloor \leq m_0 \leq \lceil n^{\frac{1}{d}} \rceil$

choose X_1, X_2 such that $0 < X_1 < X_2 < 0.5$

choose a_d such that $X1 < \frac{abs(a_d)}{m_0} < X2$

$m_\delta \leftarrow \lfloor \frac{n}{a_d}^{\frac{1}{d}} \rfloor$

$m \leftarrow m_0$

while a_d has a large b-smooth co-factor **do**

write n in base m: $n = a_d \cdot m^d + a_{d-1} \cdot m^{d-1} \cdots + a_0 \cdot m^0$

$f_m(x) = a_d \cdot x^d + a_{d-1} \cdot x^{d-1} \cdots + a_0 \cdot x^0$

if $\alpha(f_m(x))$ is good **then**

add $f_m(x)$ to F'

end if

$m \leftarrow m + m_\delta$

choose a_d such that $X_1 < \frac{abs(a_d)}{m_0} < X_2$

end while

choose $f(x)$ from F' by performing small sieving experiments

return $(m, f(x))$

END

VERSION 3 (based on Murphy, PhD Thesis, 1999)

`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.1081`

BEGIN

Fix an interval a_d in which each $a_d \ll m$ for each m

Select X_1, X_2 such that $X_1 \leq \frac{abs(a_d)}{m} \leq X_2$

Choose c such that c is a cofactor of a_d and is a product of many small primes p^k

for each a_d in the interval **do**

$m_1 = \lfloor \frac{N}{a_d}^{\frac{1}{d}} \rfloor$

Find values m surrounding m_1 such that: $abs((a_{d-1})^m) \leq X^m$

Add m to M'

end for

for each m in M' **do**

write n in base m: $n = a_d \cdot m^d + a_{d-1} \cdot m^{d-1} \cdots + a_0 \cdot m^0$

$f_m(x) = a_d \cdot x^d + a_{d-1} \cdot x^{d-1} \cdots + a_0 \cdot x^0$

if $\alpha(f_m(x)))$ is good **then**

choose $f_m(x)$

end if

end for

return $(m, f(x))$

END

Each of these techniques provides ever more efficient polynomials. As mentioned above, the algorithm from [52] is potentially the least efficient. The algorithm from [8] is a more complicated version akin to that discussed by by Murphy in his dissertation. [31]

In 1999, [31] introduced some heuristics that allow implementations to better determine how efficient a given polynomial is before using a trial sieving step.

In his research, [31] identified several properties that influence the smoothness yield of a polynomial in the number field sieve:

- *Size:* "Here we have verified that $d = 4, 5, 6$ are the relevant degrees for non-monic base-m polynomials and N in the range of interest. For most of this range, $d = 5$ is the best degree." [31, p54]

- *Root Properties:* Murphy introduces the α-approximation, which measures the root properties of polynomial in the number field sieve. Thus, where root properties are measured using $\alpha(F)$, a polynomial with $\alpha(F) \ll 9$ will be preferred.

- *Choice of d:* Murphy found that on average, $d = 4$ is better than $d = 6$ and $d = 5$, but the difference is not so great that these considerations should enter into the choice of d. [31, p54]

For 45113, the values of d, m and $f(x)$ are calculated to be:

- $d = \lfloor \frac{3 \cdot log(45113)}{log(log(45113))}^{\frac{1}{3}} \rfloor = 2$, but we will choose $d = 3$
- $m = n^{\frac{1}{d}} = 45113^{\frac{1}{3}} = 35$, but we will choose $m = 31$
- $f(m) = c_3 \cdot m^3 + c_2 \cdot m^2 + c_1 \cdot x + c_0 = 1 \cdot m^3 + 15 \cdot m^2 + 29 \cdot m + 8$

4.3 Factor Bases

After polynomial selection, the algorithm requires a set of *factor bases* which are used to test the smoothness of the the results of the (a, b) pairs over both the *algebraic* polynomial and the *rational* polynomial. Generating the rational factor base is conceptually simple: it is simply the pairs (r, p) where $r = m \mod p$ for all primes less than the desired threshold for the factor base (e.g., all primes p less than B_{RFB}). The algebraic base is calculated using a similar construct for the algebraic field: all pairs (r, p) where $0 \leq r \leq p - 1$ and $f(r) = 0 \mod p$. A third factor base, called the *quadratic character base*, which is used in the linear algebra step to better define (a, b) pairs that will result in the desired congruent squares.

INPUTS:

n : the number to factor
m : root of polynomial $f(x) \mod n$, i.e. an integer, such that $f(m) \mod n = 0$
$f(x)$: polynomial [for the algebraic side]

OUTPUTS:

RFB : rational factor base

AFB : algebraic factor base

RFB and AFB are sets of pairs (p, r) fulfilling conditions specific to the given base.

We denote the numbers of such pairs by $\#RFB$, $\#AFB$ respectively.

QCB : quadratic character base

QCB is a set of pairs (q, s) fulfilling the conditions for the given base.

We denote the size of the QCB by $\#QCB$.

PSEUDOCODE:

VERSION 1 (based on Crandall and Pomerance)

`http://books.google.com/books?id=RbEz-_D7sAUC&lpg=PP1&pg=PA293#v=onepage&q&f=false`

BEGIN

$B_{RFB} = B_{AFB} = \lfloor \frac{8}{9}^{\frac{1}{3}} \cdot (ln(n))^{\frac{1}{3}} \cdot (ln(n))^{\frac{2}{3}} \rfloor$

if B_{RFB} is not acceptable **then**

choose B_{RFB}

end if

if B_{AFB} is not acceptable **then**

choose B_{AFB}

end if

$B_{QCB} \leftarrow max(B_{RFB}, B_{AFB})$

$RFB = \{(p, r) : p \text{ prime}, p \leq B_{RFB}, r = m \mod p\}$

$\#RFB =$ size of RFB

$AFB = \{(p, r) : p \text{ prime}, p \leq B_{AFB}, 0 \leq r \leq p-1, f(r) = 0 \mod p\}$

$\#AFB =$ size of AFB

$\#QCB = k = \lfloor 3 \cdot ln(n) \rfloor$

if $\#QCB$ is not acceptable **then**

choose $\#QCB$

end if

{QCB consists of pairs with values of q starting at B_{QBC} such that the size of QCB is $\#QCB$}

$QCB = \{(q, s) : q \text{ prime}, q >= B_{QCB}, 0 \leq s \leq q-1, f(s) = 0 \mod q, f'(s) \neq 0 \mod q\}$

return (RFB, AFB, QCB)

END

VERSION 2 (based on Jensen, p. 61-62)

`http://pgnfs.org/DOCS/thesis.pdf`

Primary difference compared to VERSION 1

$\#AFB = c \cdot \#RFB$, where $2 \leq c \leq 3$

$\#QCB < 100$

The algebraic and rational factor bases are then used within the sieving step to determine whether a particular (a, b) pair, when evaluated via the algebraic and rational polynomials, respectively, are *smooth* over their corresponding factor bases.

The largest prime for the rational factor base will be set to 29. For 45113, the rational factor base will be calculated as $(p, m \mod p)$ as shown in the Table 4.1: The largest prime for the algebraic factor base will be set to 109. The algebraic factor base will be

Table 4.1: Rational Factor Base

(2, 1)	(3, 1)	(5, 1)
(7, 3)	(11, 9)	(13, 5)
(17, 14)	(19, 12)	(23, 8)
(29, 2)		

calculated as $AFB = \{(r,p) : p \leq 109, 0 \leq r \leq p-1, f(r) = 0 \mod p\}$ and is shown in Table 4.2: There will be five elements of the quadratic character base, calculated as

Table 4.2: Algebraic Factor Base

(2, 0)	(7, 6)	(17, 13)	(23, 11)
(29, 26)	(31, 18)	(41, 19)	(43, 13)
(53, 1)	(61, 46)	(67, 2)	(67, 6)
(67, 44)	(73, 50)	(79, 47)	(79, 23)
(79, 73)	(89, 62)	(89, 28)	(89, 73)
(97, 28)	(101, 87)	(103, 47)	

$QCB = \{(q,s) : p \leq 109, 0 \leq r \leq p-1, f(r) = 0 \mod p\}$ and is shown in Table 4.3:

Table 4.3: Quadratic Character Base

(107, 80)	(107, 4)	(107, 8)	(109, 99)	(113, 108)

4.4 Sieving Step

Technically, the sieving step is an extremely simple aspect of the number field sieve. The goal is to parse all of the (a, b) pairs on a particular plane–starting with $b = B_{min}$ and $-A \leq a \leq A$ and incrementing b until $b > B_{max}$.

INPUTS:

n : number to be factored

$f(x)$: polynomial [for the algebraic side]

m : the value such that $f(m) = 0 \mod n$

RFB : rational factor base

AFB : algebraic factor base

QCB : quadratic character base

A : boundary of the sieving range for a

B_{min} : the starting point for the value b

B_{max} : the ending point for the value b

$B_{algebraic\_fudge}$: the fudge factor for algebraic sieving

$B_{rational\_fudge}$: the fudge factor for rational sieving

We denote the size of respective bases by $\#RFB$, $\#AFB$, and $\#QCB$.

OUTPUTS:

$relations$: set of pairs (a, b) such that $norm(a - b \cdot \alpha)$ is smooth over AFB and $a - b \cdot m$ is smooth over RFB

the number of pairs in this set is denoted by $\#relations$

VARIABLES:

A : boundary of the sieving range for a:

$-A \leq a \leq A$

$RS[a]$: array with index a, indicating the level

to which $G(a, b)$ is divisible by primes p from RFB

$AS[a]$: array with index a, indicating the level

to which $F(a, b)$ is divisible by primes p from AFB

PSEUDOCODE:

BEGIN

$d \leftarrow degree(f(x))$

$A \leftarrow \lfloor \frac{8}{9}^{\frac{1}{3}} \cdot (\ln n)^{\frac{1}{3}} \cdot (\ln n)^{\frac{2}{3}} \rfloor$

$F(x, y) \leftarrow y^d \cdot f(\frac{x}{y})$

$G(x, y) \leftarrow y \cdot (\frac{x}{y} - m) \leftarrow x - y \cdot m$

$b \leftarrow B_{min}$

repeat

{Added the threshold calculation here}

for $(a \leftarrow -A; a \leq A; a \leftarrow a + 1)$ **do**

$RS[a] \leftarrow -\log_{\sqrt{2}} G(a, b) + B_{rational\_fudge}$

$AS[a] \leftarrow -\log_{\sqrt{2}} F(a, b) + B_{algebraic\_fudge}$

end for

for all $(p, r) \in RFB$ **do**

$log\_p \leftarrow \log_{\sqrt{2}} p$

$a \leftarrow b \cdot r \mod p$

while $(a \leq A)$ **do**

$RS[a] \leftarrow RS[a] + log\_p$

$a \leftarrow a + p$

end while

$a \leftarrow b \cdot r \mod p$

$a \leftarrow a - p$

while $(a \geq -A)$ **do**

$RS[a] \leftarrow RS[a] + log\_p$

$a \leftarrow a - p$

end while

end for

for all $(p, r) \in AFB$ **do**

$log\_p \leftarrow \log_{\sqrt{2}} p$

$a \leftarrow b \cdot r \mod p$

while $(a \leq A)$ **do**

$AS[a] \leftarrow AS[a] + log\_p$

$a \leftarrow a + p$

end while

$a \leftarrow b \cdot r \mod p$

$a \leftarrow a - p$

while $(a \geq -A)$ **do**

$AS[a] \leftarrow AS[a] + log\_p$

$a \leftarrow a - p$

end while

end for

for $(a \leftarrow -A; a < A; a \leftarrow a + 1)$ **do**

{Now the threshold test is for positive values}

if $RS[a] > 0$ and $AS[a] > 0$ and $gcd(a, b) = 1$ **then**

add a pair (a, b) to the set of *relations*

end if

end for

$b \leftarrow b + 1$

until $b > B_{max}$

return *relations*

Each (a, b) pair is processed through the algebraic and rational polynomials, which are as follows:

- $F(x, y) = y^d \cdot f(\frac{x}{y})$, where $f(x)$ is the polynomial from Section 5.4
- $G(x, y) = x - y \cdot m$, where m was chosen in Section 5.4

The results are tested for smoothness, which for performance purposes is calculated using a simple heuristic. Rather than calculating $F(a, b)$ and $G(a, b)$ for each (a, b), the line sieve relies on the following fact:

- For any (a, b) pair and the set of pairs $(p, r) \in AFB$, the value $b \cdot r \mod p + k \cdot p$ identifies a value of a such that the corresponding $F(a, b)$ is divisible by p
- For any (a, b) pair and the set of pairs $(p, r) \in RFB$, the value $b \cdot r \mod p + k \cdot p$ identifies a value of a such that the corresponding $G(a, b)$ is divisible by p

Each element of RS and AS are initialized as follows:

- $AS[a] \leftarrow -\log_{\sqrt{2}} F(a, b) + B_{algebraic\_fudge}$
- $RS[a] \leftarrow -\log_{\sqrt{2}} G(a, b) + B_{rational\_fudge}$

For each value of $a = b \cdot r \mod p + k \cdot p$, $\log(p)$ is added to a vector $AS[a]$, for the set of (p, r) pairs in AFB and a vector $RS[a]$ for the set of (p, r) pairs in RFB. Thus,

$AS[a] = \log_{\sqrt{2}}(p_1) + \log_{\sqrt{2}}(p_2) + \log_{\sqrt{2}}(p_3) + \cdots + \log_{\sqrt{2}}(p_{max})$ for a set of primes p in AFB. By adding the logarithms of p_i instead of multiplying each p_i out, the performance of the algorithm is increased. Ultimately, if a given value of $AS[a]$ is larger than 0, it implies that the prime-factorization of $F(a, b)$ is smooth over the factor base. Because this will not be completely accurate if there are any prime powers in the factorization, it is important to include a *fudge factor* (e.g., $B_{algebraic\_fudge}$), then that number is likely to have a prime-factorization of a sufficient number of elements of the factor base such that it is truly smooth over the factor base. Once the algebraic side has been completed, the algorithm repeats using the exact same methodology for the polynomial $G(a, b)$ and the RFB.

The sieving step will produce the relations shown in Table 4.4.

Table 4.4: Sieved Relations

(-73, -1)	(-47, -1)	(-28, -1)	(-13, -1)	(-6, -1)	(-2, -1)	(-1, -1)
(1, -1)	(2, -1)	(3, -1)	(4, -1)	(8, -1)	(13, -1)	(14, -1)
(15, -1)	(23, -1)	(32, -1)	(56, -1)	(61, -1)	(104, -1)	(116, -1)
(-5, -2)	(1, -2)	(3, -2)	(25, -2)	(33, -2)	(-16, -3)	(-8, -3)
(2, -3)	(5, -3)	(17, -3)	(19, -4)	(-132, -5)	(-43, -5)	(14, -5)
(37, -5)	(48, -5)	(54, -5)	(313, -5)	(-43, -6)		

4.5 Mini Factoring and Matrix Generation

The mini-factoring step includes the following function: $factor(c, base, a, b)$, which performs the following operations:

INPUTS:

c : the number to factor

$base$: either the AFB or RFB

a : the value of a

b : the value of b

NOTES:

Both $F(x, y)$ and $G(x, y)$ are known

OUTPUT:

v : an exponent vector representing $c = \prod_{p \in FB} p^i$

PSEUDOCODE:

BEGIN

Calculate the prime factorization of c such that $U = \{q^i : q^i \mid c\}$
$v = \{\}$
if $base = RFB$ **then**
  for Each $p \in RFB$ **do**
    if $p^i \in U$ for any i **then**
      Append i to v
    else
      Append 0 to v
    end if
  end for
else if $base = AFB$ **then**
  for Each $(p, r) \in AFB$ **do**

if $p^i \in U$ for any i and $a - br \mod p = 0$ **then**

Append i to v

else

Append 0 to v

end if

end for

end if

The prime factorization can be calculated using any factoring algorithm available. It has been implemented using elliptic curves, Pollard's ρ, Pollard's $\rho - 1$ and many other algorithms. In this implementation, trial division over the factor base was implemented.

For the relation $(-8, -3)$ (chosen by [12]), the exponent vectors of $factor(F(-8, -3), AFB, -8, -3)$ and $factor(G(-8, -3)), RFB, -8, -3)$ will be calculated. $G(-8, -3) = -8 - (-3) \cdot m = -8 + 3 \cdot 31 = 85 = 5 \cdot 17$. Thus, the exponent vector would be: $U = \{0010001000\}$, representing $2^0 \cdot 3^0 \cdot 5^1 \cdot 7^0 \cdot 11^0 \cdot 13^0 \cdot 17^1 \cdot 19^0 \cdot 23^0 \cdot 29^0$.

Once the requisite number of (a, b) pairs have been identified, the algorithm constructs a matrix to solve for a system of linear equations. These equations represent each of the (a, b) pairs, with each column representing the following:

- The first entry is 1 if $G(a, b) < 0$ and 0 if $G(a, b) \geq 0$
- For each element in $e \in factor(G(a, b), RFB)$, add the result of $e \mod 2$ to the vector
- For each element in $d \in factor(F(a, b), AFB)$, add the result of $d \mod 2$ to the vector
- For each $(q, s) \in QCB$, calculate the *Jacobi symbol* of $\left(\frac{a-bs}{q}\right)$ and add element of value 1 if the *Jacobi symbol* is -1 and a 0 otherwise.

If there were not enough rows added to the matrix, more (a, b) pairs will be identified via further iterations of the sieving step.

INPUTS:

$matrix$: the matrix that will be used to perform the linear algebra step

$\#relations\_required$: the number of relations that we need to factor n

$relations$: the (a, b) pairs

m : the value such that $f(m) = 0 \mod n$

$f(x)$: sieving polynomial

RFB : rational factor base

AFB : algebraic factor base

RFB and AFB are sets of pairs (p, r) fulfilling conditions specific to the given base.

QCB : quadratic character base

QCB is a set of pairs (q, s) fulfilling conditions specific to the given base.

We denote the sizes of such pairs by $\#RFB$, $\#AFB$, $\#QCB$, respectively.

NOTES:

$d \leftarrow degree(f(x))$

$F(x, y) \leftarrow y^d \cdot f(\frac{x}{y})$

$G(x, y) = x - y \cdot m$

For each (a, b) in relations there is an exponent vector showing the prime factorization of $F(a, b)$ over RFB and G(a, b) over AFB

OUTPUT:

matrix : a matrix meeting the requirements set forth in Pomerance on page 293

$relations\_found$: a copy of *relations* with all invalid relations removed

$\#relations\_found$: the number of valid relations added to the matrix

PSEUDOCODE:

BEGIN

$\#relations\_found \leftarrow 0$

$t \leftarrow \#RFB$

$u \leftarrow \#AFB$

for all $(a, b) \in relations$ **and** matrix rows $\leq \#relations\_required$ **do**

$\{factor(c, base, a, b)$ returns the exponent vectors of c over the factor base $base\}$

$(e_1, \cdots, e_t) \leftarrow factor(G(a, b), RFB, a, b)$

if $(e_1, \cdots, e_t) = V_{zero}$ **then**

remove (a, b) from *relations*

break

end if

$\{factor(c, base, a, b)$ returns the exponent vectors of c over the factor base $base\}$

$(d_1, \cdots, d_u) \leftarrow factor(F(a, b), AFB, a, b)$

if $(d_1, \cdots, d_u) = V_{zero}$ **then**

remove (a, b) from $relations$

break

end if

$\#relations\_found \leftarrow \#relations\_found + 1$

if $G(a, b) < 0$ **then**

$vector[0] \leftarrow 1$

else

$vector[0] \leftarrow 0$

end if

for $(col \leftarrow 1; col \leq t; col \leftarrow col + 1)$ **do**

$vector[col] \leftarrow e_{col} \mod 2$

end for

for $(col \leftarrow t + 1; col \leq t + u; col \leftarrow col + 1)$ **do**

$vector[col] \leftarrow d_{col-t} \mod 2$

end for

for $(col \leftarrow t + u + 1; col \leq t + u + \#QCB; col \leftarrow col + 1)$ **do**

$(q, s) \leftarrow QCB.next$

{Jacobi symbol}

if $\left(\frac{a-bs}{q}\right) = -1$ **then**

$vector[col] \leftarrow 1$

else

$vector[col] \leftarrow 0$

end if

end for

$matrix.append\_row(vector)$

end for

return $(matrix, relations\_found, \#relations\_found)$

For the relation $(-8, -3)$, the vector will be written as:

$$\{00010001000000000000000000000001000010010\}$$

4.6 Linear Algebra Step

Once the matrix has been constructed, the algorithm solves the system of linear equations. There are a number of different algorithms that have been introduced in the literature, most notably the Wiedemann, Block Wiedemann, Lanczos and Block Lanczos algorithms. For simplicity, this implementation uses the much less efficient *Gaussian elimination* technique, in which operations are performed on columns and rows of the matrix in order to determine the *upper triangular form* of the matrix, which is as follows:

To do this, the system of linear equations (below) [53]

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{4.2}$$

where b_1 through b_m are 0 into the *augmented matrix equation*: [53]

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{4.3}$$

Converting the matrix into its *upper triangular form* provides the information needed to

solve the system of linear equations. [53]

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{mn} & b'_m \end{bmatrix} \tag{4.4}$$

This allows us to solve the equation for the first row and substitute the result into the second row and continue for each row of the matrix [53]. The result of the Gaussian elimination usually provides a number of rows where there are an infinite number of possible solutions–meaning the variable associated with that row is a *free variable*. Solutions for the system can be calculated by assigning an arbitrary number to the free variable and computing the solutions for the rest of the variables in the matrix. As such, the linear algebra step returns both the reduced form of the matrix and the list of free variables for later processing.

INPUTS:

M - the matrix

OUTPUTS:

$freecols$ - the column representing the free variables
M - the matrix in reduced Echelon form

NOTE:

Free variables are the set of variables within the solution such that if values for

the free variables are chosen, a solution will be generated.

PSEUDOCODE:

Set the size of $freecols$ to the number of rows in M

{The matrix is built row-by-row, so we transpose.}

$M \leftarrow M^T$

$(M, freecols) \leftarrow GaussianElimination(M)$

return $(M, freecols)$

4.7 GaussianElimination

See page 72 of Per Leslie Jensen's masters thesis

INPUTS:

M : the Matrix

OUTPUTS:

M : the matrix in reduced echelon form

$freecols$: the free variables

VARIABLES:

$freecols$: a binary vector of length $\#rows$

t : used to sort the rows into reduced echelon order

h : used to specify columns with free variables

PSEUDOCODE:

```
#rows ← M.rows()
#cols ← M.cols()
for i ← 0; i < #freecols; i ← i + 1 do
    freecols[i] ← 0
end for
h ← 0
for (i ← 0; i < #rows and h < #cols; i ← i + 1) do
    next ← false
    if (M[i][h] is 0 then
        t ← i + 1
        while (t < #rows and M[t][h] is 0) do
            t ← t + 1
        end while
        if (t < #rows) then
            swap rows M[i] and M[t]
        else
            freecols[h] ← 1
            i ← i − 1
            next = true
        end if
    end if
    if (next is false) then
        for (j ← i + 1; j < #rows; j ← j + 1) do
            if M[j][h] is 1 then
                {Add rows}
                M[j] ← M[j] + M[i]
            end if
```

end for

for $(j \leftarrow 0; j < i; j \leftarrow j + 1)$ **do**

if $M[j][h]$ **is** 1 **then**

{Add rows}

$M[j] \leftarrow M[j] + M[i]$

end if

end for

end if

$h \leftarrow h + 1$

end for

return $(M, freecols)$

4.8 Get Solution Set

INPUTS:

M - the Matrix

$freecols$ - the vector of free variables

num - the number of solutions to use

OUTPUT:

res - a vector where 1 represents the column of an (a, b) pair in the solution

PSEUDOCODE:

for $(i \leftarrow 0; i < M.cols(); i \leftarrow i + 1)$ **do**

$res[i] \leftarrow 0$

```
end for
j ← −1
i ← num
while i > 0 do
  j ← j + 1
  while freecols[j] = 0 do
    j ← j + 1
  end while
  i ← i − 1
end while
res[j] ← 1
for i = 0; i < M.rows() − 1; i ← i + 1 do
  if M[i][j] = 1 then
    h ← i
    while h < j do
      if M[i][h] = 1 then
        res[h] = 1
        break
      end if
      h ← h + 1
    end while
  end if
end for
return res
```

4.9 Square Root Step

The square root step takes all of the (a, b) pairs identified by the matrix step and calculates u^2 as the product of a set of *algebraic numbers*, $(a - b \cdot \alpha)$. An algebraic number is defined as a number that is a root of a polynomial with integer coefficients. Thus, $(a - b \cdot \alpha)$ is a root of the polynomial $h(x) = a - b\alpha$. The square root step generates an algebraic number for each (a, b) pair: $(a - b \cdot \alpha) \mod f(\alpha)$ together with $f'(\alpha)^2 \mod f(\alpha)$. This allows for the construction of a polynomial, $\gamma(\alpha)^2 \mod f(\alpha)$, for which each (a, b) pair is a root.

Because $\gamma(\alpha)^2$ is an algebraic number, it is important to have a mapping function that maps $\gamma(\alpha)^2$ back to $\mathbf{Z}$. To this end, the number field sieve uses the *norm* function, which is a mapping function allowing for the following:

- $norm : \mathbf{Z}[\alpha] \rightarrow \mathbf{Q}(\alpha)$
- $norm : \mathbf{Z}[\alpha] \rightarrow \mathbf{Z}$
- $norm : \mathbf{Z} \rightarrow \mathbf{Q}$
- $norm : \mathbf{Q}(\alpha) \rightarrow \mathbf{Q}$

A norm function also has the following properties:

- $norm(f(x)^i) = norm(f(x))^i$
- $norm(-f(x)) = -norm(f(x))$
- $norm(g(x) \cdot f(x)) = norm(g(x)) \cdot norm(f(x))$
- $norm(c \cdot f(x)) = c \cdot norm(f(x))$

A good example of the norm function can be seen below:

INPUT

$g(x)$: the polynomial for which norm will be calculated

$f(x)$: the sieving polynomial

OUTPUT

r : the norm of $g(x)$

VARIABLES:

mat : a matrix of size $o + p \times o + p$

PSEUDOCODE:

$c \leftarrow$ coefficient of x^p in $f(x)$
$c \leftarrow c^o$
for $(0 \leq i < o)$ **do**
  for $(0 \leq j \leq p)$ **do**
    $mat[i][i + j] \leftarrow$coefficient of x^{p-j} in $f(x)$
  end for
end for
for $(0 \leq i < p))$ **do**
  for $(0 \leq j \leq o)$ **do**
    $mat[o + i][i + j] \leftarrow$coefficient of x^{o-j} in $g(x)$
  end for
end for
$r \leftarrow det(mat)/c$
return r

Thus, $\gamma(\alpha)^2$ represents u^2 and through the Chinese remainder theorem, the algorithm calculates $\sqrt{\gamma(\alpha)^2}$, using $norm(\gamma(\alpha))$ to produce an integer value u. Finally, the algorithm calculates $v = \sqrt{f'(m)^2 \prod_{(a,b)\in pairs}(a - b \cdot m)} \mod n$.

INPUT:

$pairs$: a list of the (a, b) pairs which produce a square in $\mathbb{R}$ and $\mathbb{A}$

$f(x)$: the sieving polynomial

n : the number to factor

m : the value m from $f(m) = 0 \mod n$

OUTPUT:

$fact$: one of the factors of n

PSEUDOCODE:

$gamma\_alpha\_2 \leftarrow 1$

$norm \leftarrow 1$

$u \leftarrow 0$

for all $(a, b) \in pairs$ **do**

  if $u < |a|$ **then**

    $u \leftarrow |a|$

  end if

  if $u < |b|$ **then**

    $u \leftarrow |b|$

  end if

  $norm \leftarrow norm(a - b\alpha, f(x)) \cdot norm$

  $gamma\_alpha\_2 \leftarrow gamma\_alpha\_2 \cdot (a - b\alpha) \mod f(\alpha)$

end for

$gamma\_alpha\_2 \leftarrow gamma\_alpha\_2 \cdot (f'(\alpha))^2 \mod f(\alpha)$

$(ps, M) \leftarrow choose\_ps(m, f(\alpha), gamma\_alpha\_2(\alpha), u, n, \#pairs)$

$xps \leftarrow find\_xps(f(\alpha), ps, gamma\_alpha\_2(\alpha), m)$

$u \leftarrow calculate\_crt(xps, ps)$

$v \leftarrow \sqrt{f'(m)^2 \prod_{(a,b)\in pairs} (a - b \cdot m)} \mod n$

$fact \leftarrow gcd(u - v, n)$

return $fact$

To use CRT to calculate $\sqrt{u^2}$, the algorithm identified a four prime numbers meeting the following requirements:

1. $\epsilon = 0.01$
2. $d = degree(f(x))$
3. $u =$ largest absolute value of a or b from the (a, b) relations returned from the matrix step
4. $\#s =$ the size of the (a, b) pairs returned from the matrix step
5. $n =$ the number to be factored
6. $p_1 \cdot p_2 \cdot p_3 \cdot p_4 \cdot (\frac{1}{2} - \epsilon) \geq d^{\frac{d+5}{2}} \cdot n \cdot (2 \cdot u\sqrt{d} \cdot n^{\frac{1}{3}})^{\frac{\#s}{2}}$
7. Both $f(x)$ and $gamma\_alpha\_2(\alpha)$ are irreducible over each p_i

The pseudocode below describes one method to find these four prime numbers.

INPUT

m : the root of $f(x) \mod n$ where n is the number we are factoring

$f(x)$: the sieving polynomial

$gamma\_alpha\_2(\alpha)$: the product of all smooth $(a - b \cdot \alpha)$ pairs over AFB multiplied by $f'(\alpha)$.

u : the largest absolute value found in a or b over all (a, b) pairs in $gamma\_alpha\_2(\alpha)$

n : the number to be factored

$\#s$: the size of the (a, b) pair list

CONSTANTS

$\epsilon \leftarrow 0.1$

OUTPUT

ps : the list of primes for which $\#ps = 4$

M : the product of all the primes in ps

PSEUDOCODE:

$d \leftarrow degree(f(x))$

choose $x \leq d^{\frac{d+5}{2}} \cdot n \cdot (2 \cdot u\sqrt{d} \cdot n^{\frac{1}{3}})^{\frac{\#s}{2}}$

$p \leftarrow (2 \cdot x)^{\frac{1}{4}}$

$M \leftarrow 1$

while $x \leq (\frac{1}{2} - \epsilon) \cdot M$ **do**

  $p \leftarrow next\_prime(p)$

  if $\#ps > 4$ **then**

    $M \leftarrow M/ps.front$

    remove $ps.front$

  end if

```
    if f(x) mod p is irreducible then
      if gamma_alpha_2(α) mod p is irreducible then
        add p to ps
        M ← M * p
      end if
    end if
  end while
  return (ps, M)
```

The four primes for the example 45113 are: 75516479, 75516409, 75516457, and 75516473.

In the second phase of the algorithm, $\sqrt{u^2}$ is calculated over the four primes selected in the function $find\_xps$. By performing the following: $h_p(\alpha) = gamma\_alpha\_2(\alpha) \mod p$ and $f_p(\alpha) = f(\alpha) \mod p$ for each of the four values of p calculated above, the square root of the algebraic function can be calculated over the finite field $\mathbb{F}_p^d$. With this value, the correct sign of $g_p(\alpha)$ will be calculated by comparing $norm(g_p(\alpha))$ to $\sqrt{norm(gamma\_alpha\_2(\alpha))} \mod p$. If these two values don't match, then $-g_p(\alpha)$ is congruent to $\sqrt{gamma\_alpha\_2(\alpha)} \mod p$. Thus, $g_p(m) \mod p$ is congruent to $\sqrt{u^2} \mod p$. These steps are repeated for each of the four values of p calculated above as per the pseudocode.

```
  INPUT

f(α) : the sieving polynomial
ps : a list of primes such that f(α) is irreducible over each p and gamma_alpha_2 is a
square root over each p
gamma_alpha_2(α) : the product of all smooth (a − b · α) pairs over AFB multiplied
by f′(α).
m : the root of f(α) mod n where n is the number we are factoring
```

OUTPUT

xps : the list of values for $\gamma(m) \mod p$ for each p in ps

PSEUDOCODE:

for all $p \in ps$ **do**

$d \leftarrow degree(f(\alpha))$

$h_p(\alpha) \leftarrow gamma\_alpha\_2(\alpha) \mod p$

$f_p(\alpha) \leftarrow f(\alpha) \mod p$

$g_p(\alpha) \leftarrow \sqrt{gamma\_alpha\_2(\alpha)}$ over ${\mathbb{F}_p}^d$

$nm \leftarrow \sqrt{norm(gamma\_alpha\_2(\alpha), f(\alpha))} \mod p$

$g_n \leftarrow norm(gp(\alpha), f(\alpha))$

if $nm \neq g_n$ **then**

$g_p(\alpha) \leftarrow -g_p(\alpha) \mod p$

$g_n \leftarrow norm(g_p(\alpha), f(\alpha))$

end if

$x_p \leftarrow g_p(m) \mod p$

add x_p to the list xps

end for

return xps

The four pairs produced are:

- $(4461724, 75516409)$
- $(4461100, 75516457)$
- $(4460896, 75516473)$

- $(4460814, 75516479)$

With the four pairs (x_p, p) such that each value $x_p = \sqrt{u^2} \mod p$, there is enough information to use the CRT to reconstruct the value of $\sqrt{u^2}$. The algorithm takes a value $M_i = \prod p_{i1 \leq i \leq 4}/p_i$ and calculates $a_i = M^{-1} \mod p_i$. Using these two values, a values z can be computed such that $z = \sum a_i \cdot M_i \cdot x_{p_i 1 \leq i \leq 4}$.

Using z, a value r can be calculated such that $r = \lceil \frac{z}{\prod p_{i1 \leq i \leq 4}} + \frac{1}{2} \rceil$. Using r, the value of u can be calculated as $u = (z - r \cdot \prod p_{i1 \leq i \leq 4}) \mod n$.

INPUT:

xps : the values of $\gamma(m) \mod p$ for each p in ps

ps : the list of primes p over which $f(x)$ is irreducible

NOTE: the length of xps and ps is the same

n - the number to be factored

OUTPUT

u : the algebraic square root $\mod n$ of $\gamma(m)^2$

NOTE: m is the root of $f(x) \mod n$ where $f(x)$ is the sieving polynomial

PSEUDOCODE:

$z \leftarrow 0$

$t \leftarrow 0$

$prod\_p \leftarrow \prod_{p \in ps}(p)$

for $(i \leftarrow 0; i < \#xps; i \leftarrow i + 1)$ **do**

$M \leftarrow \frac{prod\_p}{ps[i]}$

$a_i \leftarrow M^{-1} \mod p$

$z \leftarrow z + a_i \cdot M \cdot xps[i]$

end for

$r \leftarrow \lceil \frac{z}{prod\_p} + \frac{1}{2} \rceil$

$u \leftarrow (z - r \cdot prod\_p) \mod n$

return u

From the example:

$$
\begin{aligned}
z &= \sum a_i \cdot M_i \cdot x_{p_i 1 \leq i \leq 4} \\
&= 2131976 \cdot 430650580094814534137119 \cdot 4461724 \\
&+ 27572267 \cdot 430650306363383456125903 \cdot 4461100 \\
&+ 12475459 \cdot 430650215119650425663327 \cdot 4460892 \\
&+ 33336766 \cdot 430650180903260507405849 \cdot 4460957 \\
&= 1450756094020444368005264708525236221 88
\end{aligned}
$$

$$
\begin{aligned}
r &= \lceil \frac{z}{\prod p_{i 1 \leq i \leq 4}} + \frac{1}{2} \rceil \\
&= 4460957
\end{aligned}
$$

$$
\begin{aligned}
u &= (z - r \cdot p_1 \cdot p_2 \cdot p_3 \cdot p_4) \mod 45113 \\
&= (145075609402044436800526470852523622188 \\
&\quad - 4460957 \cdot 3252118534252727313904314048567 1) \mod 45113 \\
&= 4861
\end{aligned}
$$

With the value of u, a factor of n can be performed as either $u - v$ or $u + v$.

The value v can be calculated as: $\prod (a - b \cdot m) \cdot f'(m)^2 = 205972529162400$. Finally, $gcd(4861 + 205972529162400, 45113) = 229$. And 229 is a factor of 45113.

Chapter 5: C++ Implementation

As part of this thesis, an implementation of the Number Field Sieve was implemented using C++ and the LiDiA [54] library. Using C++ and object-oriented programming, the implementation was able to be developed to match the pseudocodes that were developed in parallel. This way, the pseudocodes presented in Chapter 4 have been vetted by being implemented as part of a working example of the number field sieve. This chapter describes the mechanisms used to test and validate the implementation (see Section 5.1), the performance of the implementation (see Section 5.2), some experiments performed against this and other available implementations (see Section 5.3), and describes attempts to integrate an implementation of the number field sieve with the CrypTool educational software [55] (see Section 5.4).

5.1 Testing

This section will discuss the approaches to testing used for each piece of the implementation. One of the most complicated aspects of developing an implementation of the number field sieve is validating that the implementation is correct and functions as expected. Due to the complexity of the algorithm, there are many steps where a simple error can be introduced–and without a robust model for testing there is no guarantee that an end-to-end run of the implementation will be successful. To that end, as part of the development effort for this implementation, a number of tests were devised to aid in validating the implementation as it was developed. These tests range from simple smoke-test style checks to full validation of various components. Only with these tests was it possible to prove that the implementation developed was capable of correctly factoring numbers using the number field sieve algorithm–as the anecdote in Section 5.1.6 shows.

5.1.1 Testing the Polynomial Step

For the most part, testing the polynomial for the implementation used for this thesis was a straightforward process. Because the polynomial selection step used in this thesis was the simplest form of polynomial selection–simply the base-m expansion of n–where $m \approx n^{\frac{1}{d+1}}$ and d is a degree chosen based on the size of the number n to be factored. Testing that the polynomial selection implementation met the functional requirements was simply a test to validate that: $f(m) \equiv 0 \mod n$. Per [31], the base-m technique is sufficient for numbers as large as 140 base-10 digits.

Beyond simply testing the functionality of the polynomial, it is important to determine how many smooth values the polynomial produces–which is essential to speeding up the number field sieve. To this end, most implementations of the number field sieve include a small sieving step to determine how efficient a chosen polynomial is in producing smooth values.

5.1.2 Testing the Factor Bases

Testing that the factor bases were generated correctly is a straightforward activity, but it must be performed to ensure that all aspects of the number field sieve are functioning correctly. Specifically, the factor bases are defined as follows: [52, p293]

- RFB: $RFB = \{(p, r) : p \text{ prime}, p \leq B, r \equiv m \mod p\}$
- AFB: $AFB = \{(p, r) : p \text{ prime}, p \leq B, 0 \leq r \leq p-1, f(r) \equiv 0 \mod p\}$
- QCB: $QCB = \{k \text{ pairs } (q, s) : q \text{ prime}, q >= B, 0 \leq s \leq q-1, f(s) = 0 \mod q, f'(s) \neq 0 \mod q\}$

To validate these, it is important to go through the generated RFB and validate that one pair exists for each prime p and verify that $r \equiv m \mod p$. Similarly, there must be at least one pair for each prime p in the AFB, and $f(r) \equiv 0 \mod p$ for each pair. Because the (p, r) relations in the AFB are based on the roots of $f(x) \mod p$, there may be more

than one value r associated with each value p. *Complete* testing would require validating that each possible value of r is included in the results. However, generating the AFB in an iterative fashion should guarantee that all possible (p, r) relations are included in AFB. Similarly, validating the QCB simply requires ensuring that each relation (q, s) meets the criteria $f(s) = 0 \mod q$ and $f'(s) \neq 0 \mod q$.

5.1.3 Testing the Sieving Step

Testing the sieving step is one of the more complicated testing activities performed during the implementation developed for this thesis. Due to the extremely large amounts of data processed by the sieving step it is important to test not only the end of of the sieving but in-line with the sieving process.

Once sieving has been performed for all values of a where $-A \leq a < A$ for a fixed value of b, the algorithm chooses likely relations (a, b) where a and b are co-prime and the algebraic and rational thresholds have been reached. This implies that $F(a, b)$, where $F(x, y) = y^d \cdot f(\frac{x}{y})$, should be smooth over the algebraic factor base and $G(a, b)$, where $G(x, y) = x - y \cdot m$, should be smooth over the rational factor base.

During the course of the implementation, checks against smoothness were performed in-line with the line sieving. In practice, this is overkill as it duplicates effort used in the matrix generation step. In addition, the algorithm used to validate the smoothness was also not truly efficient:

```
{b is fixed at this point}
if (gcd(a, b) = 1 and AS[a] > 0 and RS[a] > 0) then
  F_RFB ← |G(a, b)|
  for all p ∈ RFB do
    while (F_RFB > 0 and F_RFB ≠ 1 and F_RFB mod p = 0) do
      F_RFB = F_RFB / p
```

end while

end for

if $F_{RFB} = 1$ **then**

$G(a, b)$ is smooth over RFB

end if

$F_{AFB} \leftarrow |F(a, b)|$

for all $p \in AFB$ **do**

while ($F_{AFB} > 0$ and $F_{AFB} \neq 1$ and $F_{AFB} \mod p = 0$) **do**

$F_{AFB} = F_{AFB}/p$

end while

end for

if $F_{AFB} = 1$ **then**

$F(a, b)$ is smooth over AFB

end if

end if

5.1.4 Testing the Matrix Step

Upon completion of the linear algebra step, the algorithm computes the value of X in the equation $A \cdot X = 0$. To that end, testing the results of the linear algebra step simply require validating that the results provided satisfy the above equation. Further, because this implementation leveraged the matrix step developed by [8], an abundance of testing had already been performed to validate it.

5.1.5 Testing the Square Root Step

Testing the square root step turned out to be the most complicated aspect of this entire endeavor. To perform this, much of the testing had to occur in parallel with the implementation. The first step was to ensure that all (a, b) pairs deemed to be part of the γ^2 are

calculated correctly. To this end, testing requires calculating the product of $norm(a - b\alpha)$ for each (a, b) pair as well as the product of $(a - b\alpha) \mod f(\alpha)$ for each (a, b) pair.

This serves two purposes: validating that the norm function is behaving properly–as the norm of the product of $a - b\alpha$ should be equal to the product of $norm(a - b\alpha)$–and it provides the necessary validation that $\sqrt{\gamma^2}$ was calculated correctly as taking the result of Couveignes' algorithm and squaring it should result in the value of γ^2.

Another heuristic for determining that the square root step was implemented correctly involves counting the prime factorizations of $G(a, b)$ and $F(a, b)$ for each (a, b) pair: each element of the RFB and AFB, respectively, should end up with an even number of powers. For example, if $G(a_1, b_1) = 2^9 \cdot 5^7$ and $G(a_2, b_2) = 2^3 \cdot 5^{19}$ then $G(a_1, b_1) \cdot G(a_2, b_2) = 2^{12} \cdot 5^{26}$. Because 12 and 26 are even, the result is a square–meaning that there is an integer u such that $u = \sqrt{G(a_1, b_1) \cdot G(a_2, b_2)}$.

5.1.6 End-to-End Testing

Once each individual portion of the number field sieve was complete, a mechanism was needed to perform end-to-end testing to validate that the implementation was correct. Ideally, an implementation that meets all of the previously discussed tests should be complete–and result in factors when provided a number generated as the product of two random prime numbers. In practice, this was not the case. In fact, the implementation had a subtle flaw in it that any randomly-generated composite number that had a factor of 3 would be correctly factorized, but any other composite number would not be correctly factorized. To fully validate that the implementation works correctly requires walking through the implementation with a known-good example of the number field sieve.

As it turns out, Briggs provides a very detailed example of the number field sieve in action in [12]. While his example uses slightly different algorithms for the linear algebra step and the square root step, the end result should be the same and all pairs generated by the sieving step should be valid given the test inputs include the same polynomial as that used by Briggs. To this end, the number 45,113 was factored in this implementation using

the value $m = 31$. Because Briggs uses the base-m expansion to generate the polynomial, the implementation generated the polynomial $f(x) = x^3 + 15x^2 + 29x + 8$, which matches that used by Briggs' example. As such, the factor bases generated by the implementation matched those in the example, making it highly likely that the sieving step would result in the same (a, b) pairs. However, comparing the results of the implementation, a different set of (a, b) pairs were generated–implying an error in the implementation of the sieving step. As it turns out, one line of the sieving step implemented $a + bm$ where it should have implemented $a - bm$. With the correct set of (a, b) pairs, the algorithm successfully completed and produced factors for 45,113: 197 and 229.

5.2 Timing Results

First, initial timing results were taken with a naive implementation of the threshold test (e.g., testing that $AS[a] > B_{algebraic\_fudge}$ and $RS[a] > B_{rational\_fudge}$). For smaller numbers, this implementation ran with the timings for each step as shown in Table 5.1. Results

Table 5.1: Sample Timing Results

n	657033396953910741871
d	3
B_{AFB}	2000
B_{RFB}	2000
$B_{algebraic\_fudge}$	50
k	10
Polynomial selection	14ms
Factor base generation	25162ms
Line sieving	3478ms
Linear algebra	222ms
Square root	27013ms

for larger numbers were not successful, see Table 5.2. Specifically, results for n_3 did not produce a single relation even after 50 hours of running. There was a suspected scaling issues in the implementation were identified: the array S for the sieving step requires a

Table 5.2: Larger Numbers

n_1	903446913989887751229829
n_2	534811055500486755544760729316203
n_3	272281914804060071572974366950855982676425838267016377021567

contiguous amount of memory. However, the errors received were *not* memory related. To this end, the *Very Sleepy* [56] profiler was deployed to identify the issue. First, the threshold test in the line sieving phase was implemented as shown:

```
if(gcd(a, b) == 1 && s[0][t] > T[0] && s[1][t] > T[1])
{
...
}
```

Very Sleepy showed that over 80% of the program's running time was being used to calculate gcd. This is due to the fact that in C++, the first item in a set of conditionals will always be tested–if the result is false, the following items will not be tested. As such, the gcd implementation as called for *every* (a, b) pair. Additionally, the gcd implementation was a naive implementation that did not account for performance. To this end, the gcd implementation was updated to use *binary* gcd [57] and the threshold test was altered as shown:

```
if(s[0][t] > T[0] && s[1][t] > T[1] && gcd(a, b) == 1)
{
...
}
```

With these simple changes to the implementation, it was able to process 24-digit numbers (see Table 5.2) in reasonable amounts of time (e.g., 644312ms). However, larger numbers still result in extreme slowdowns as the value of b increases. This is due, primarily, to

the fact that the algebraic and rational *fudge factors* increase in an almost linear fashion as the value of b increases. As such, the static choices for the fudge factors presented at the initialization phase of the algorithm become less of a barrier to bad (a, b) pairs as b increases, resulting in a large amount of time validating the smoothness of these pairs.

An improvement, as identified by [12] is to implement a slightly more complicated version of the threshold test: initializing the AS and RS arrays to: $AS[a] \leftarrow -\log_{\sqrt{2}} F(a, b) + B_{algebraic\_fudge}$ and $RS[a] \leftarrow -\log_{\sqrt{2}} G(a, b) + B_{rational\_fudge}$. Using an improved test as shown:

```
if(s[0][t] > 0 && s[1][t] > 0 && gcd(a, b) == 1)
{
...
}
```

This produced the timing results shown in Table 5.3. While this slowed increased the

Table 5.3: Sample Timing Results

n	657033396953910741871
d	3
B_{AFB}	2000
B_{RFB}	2000
m	8693523
$B_{algebraic\_fudge}$	10
$B_{rational\_fudge}$	5
k	10
Polynomial selection	16ms
Factor base generation	24939ms
Line sieving	425348ms
Linear algebra	227ms
Square root	75702ms

length of time it takes the line sieve to run by 122 times. However, this also allowed the implementation to support factoring larger numbers, one of which is shown in Table 5.4.

As shown in the tables, the most time consuming aspect of the number field sieve is simply

Table 5.4: Sample Timing Results

n	903446913989887751229829
d	3
B_{AFB}	3000
B_{RFB}	3000
m	96672039
$B_{algebraic\_fudge}$	-60
$B_{rational\_fudge}$	-40
k	10
Polynomial selection	9ms
Factor base generation	62564ms
Line sieving	6727832ms
Linear algebra	439ms
Square root	124583ms

the line sieve. In fact, where increasing the number of digits factored from 21 digits to 24 digits increases the length of time the line sieve takes by a multiple of 15, the increases in other phases were not nearly as stark: the matrix step only doubled (taking an insignificant amount of the algorithm's total time) while the amount of time required for the square root step only increased by 1.6. Also, the amount of time added to the line sieve by implementing the threshold test from [12] as it requires a large number of calculations for each possible value of a and b, making the time the line sieve takes to initialize the AS and RS arrays proportional to the amount of time required to calculate $-\log_{\sqrt{2}} F(a,b) + B_{algebraic\_fudge}$ and $-\log_{\sqrt{2}} G(a,b) + B_{rational\_fudge}$. In practice, this adds a noticeable lag to the run-time of the implementation. However, where the naive threshold test would slow down the algorithm as b increases in value, the threshold test from [12] does not–meaning that because the setup time is not related to the size of n, as n increases in size, the effect of this set-up time on the running time of the line sieve will become less noticeable.

However, with and without the improved threshold test, the profiler has shown that 25% to 50% of the time used in this implementation is spent on memory allocation and

deallocation and dealing with (a, b) pairs that do not produce smooth values of $F(a, b)$ and $G(a, b)$. This can most likely be addressed by two avenues of future work:

- Implement the polynomial selection algorithms found in [31] rather than the naive implementations that are shown in [12] and [52].
- Rework the implementation of the line sieve to reduce the amount of memory allocation and deallocation that is performed.

The value of reducing the amount of memory operations used in the sieving step is arguable as the majority of those memory operations occur in the portion of the algorithm that generates the exponent vectors. With improved polynomials and improved threshold tests, the effect of the memory operations may be greatly reduced.

It is important to note that as b increases, the values of $AS[a]$ and $RS[a]$ increase as well–regardless of which form of threshold test is performed. Another avenue for future work will be to implement improvements over the a rolling average of *good* threshold values and allow it to increase as valid (a, b) pairs are identified with larger threshold values. This will also have the benefit of reducing the number of *gcd* operations and mini-factoring operations that are currently performed in the implementation.

5.3 Experiments

Because GGNFS is widely considered to be the most efficient implementation of the number field sieve, an initial set of experiments were performed using the GGNFS suite of programs on Enigma. The Enigma machine has the following properties:

- Owned and operated by GMU
- Quad Xeon 2.8 GHz processor
- 4 GB RAM
- Several hundred GB of hard disk space

As such, Enigma provides a platform on which these experiments can easily be repeated.

The GGNFS suite of programs divides the NFS process into a number of steps. Each step is defined as its own application with its own set of inputs and parameters. This allows users of the GGNFS suite to tune a large number of NFS parameters to the specific number and polynomial that will be used. Table 5.5 outlines the parameters supported by the basic GGNFS polynomial selection function. GGNFS also provides an implementation of Franke's polynomial selection algorithm, but it is not suitable for numbers smaller than 98 digits. In addition, there is no documentation describing the parameters that the improved polynomial selection code accepts. The second phase of the GGNFS suite generates the

Table 5.5: Polynomial Selection Parameters

n	The number to factor
d	The desired degree of the polynomial
j_0 and j_1	Murphy's sieving-for-root properties
$maxs1$	Threshold for the initial polynomial test
$maxskew$	The maximum acceptable skew for the polynomial
$lc1$	The maximum relative size of the leading coefficient
lcp	The number of primes to choose from for leading coefficient divisors
$leave$	The number of bits of the leading coefficient that will be random

factor base. The parameters supported for factor base generated are outlined in Table 5.6. The third phase of the GGNFS suite performs sieving. As GGNFS provides two sieving implementations (line and lattice), there are slightly different parameters available for each. Table 5.7 shows the parameters for each. Once sieving is complete, the GGNFS suite provides an application that will process the relations from one or more sieve runs into a format suitable for generating the matrix. The matrix step consists of three applications: one for building, pruning, and solving the matrix. The parameters available during the matrix step are given in Table 5.8. The final step of the GGNFS suite, as with the final step of the NFS algorithm, is the square root step. As with the other steps, GGNFS provides

Table 5.6: Factor Base Parameters

n	The number to factor
f	The polynomial chosen for NFS
m	The value at which $f(m) = 0 \ mod \ n$
rl	The maximum size of the rational factor base
al	The maximum size of the algebraic factor base
mpr	The max large rational prime for large-prime variation
mpa	The max large algebraic prime for large-prime variation
p	The number of large rational and algebraic primes

several parameters that may be of use, identified in Table 5.9. As there is a relationship between skew and the results of the sieving step, an initial line of experiments would be to perform some tests based on the skew results of the polynomial. In particular, Monico states in the documentation that most skews are between 1500 or 2000. Similarly, j_0 and j_1 are used to find "nearby" polynomials that may have more small roots, providing smoother ideals. Nevertheless, there will be some trade-off between the amount of time spent in polynomial selection and the amount of time spent performing the rest of the algorithm.

In addition, the effects of sieving parameters (e.g., a and b) were also evaluated, aside from the obvious values affecting the performance of the sieving step. It is, nevertheless, expected that lattice sieving will perform far better than line sieving.

According to the GGNFS documentation, the two most important parameters are $maxrelssinff$ and wt. $maxrelsinff$ defines the maximum number of relations per relation-set when preparing the matrix. It is expected that this value and the sparseness of the matrix will have a direct impact on the performance of the matrix solution step.

The following test procedures were performed to gain a better understanding of the number field sieve algorithm:

1. Generate multiple n's of different sizes (40-digit, 80-digit, more if time allows)

2. Generate multiple polynomials for each value of n with different parameters

Table 5.7: Sieving Parameters

n	The number to factor
f	The polynomial chosen for NFS
m	The value at which $f(m) = 0 \ mod \ n$
$(r\|a)lambda$	How far from perfect sieve values to look for good relations.
q_0	The initial value of q for the lattice sieve
$qintsize$	The q range size for the lattice sieve
a_0	The initial value of a for the line sieve
a_1	The final value of a for the line sieve
b_0	The initial value of b for the line sieve
b_1	The final value of b for the line sieve

Table 5.8: Matrix Parameters

$minff$	The minimum number of FFs
$maxrelsinff$	The maximum relation-set weight
wt	The weight factor determining the sparseness of the matrix.

3. Generate multiple sieving runs for each polynomial with different parameters, to see how these parameters affect the efficiency of the algorithm

4. Generate multiple matrices for each sieving run with different parameters, to see how to modify the length of the matrix step versus the sieving step

5. Tabulate the time spent in each step

6. Analyze the results

Tests were developed for a 40-digit number, but they were not effectual because with the appropriate sieving parameters, NFS can factor a 40-digit number in less than 30 seconds. As such, more tests were performed on the 80-digit number.

Table 5.9: Square Root Parameters

depnum	The specific dependency to try from the matrix solution
knowndiv	The product of known small divisors of n

Table 5.10 shows the results of different sieving parameters on a polynomial generated to have a skew of 1500 for an 80-digit number. The results column indicates the result of the matrix generation step, showing the number of columns generated against the minimum required by the algorithm. Table 5.11 shows the results for a polynomial with a skew of 1625. Higher skews are not provided because higher skew polynomials were not found even when specified. As expected, the polynomial with a higher skew produces better results.

Table 5.10: Results of a 1500 skew polynomial

A_0	A_1	B_0	B_1	*Time*	*Results*
$-2 * 10^6$	$2 * 10^6$	1	2000	4m51s	2233/71537
$-2 * 10^6$	$2 * 10^6$	1	20000	33m43s	3949/71537
$-2 * 10^6$	$2 * 10^6$	1	100000	182m15	5748/71537

Table 5.11: Results of a 1625 skew polynomial

A_0	A_1	B_0	B_1	*Time*	*Rels*
$-2 * 10^6$	$2 * 10^6$	1	2000	5m47s	2935/71297
$-2 * 10^6$	$2 * 10^6$	1	20000	35m0s	5805/71297
$-2 * 10^6$	$2 * 10^6$	1	100000	185m56s	8829/71297

Nevertheless, the effect of the higher skew is almost negligible. More interesting results are shown in Table 5.12, which illustrates the effects of altering the sieving parameter a vs. the sieving parameter b. Because sieving must find pairs $a + bm$ that are smooth over the factor base, it is interesting to notice that the size of a has little effect on the number of relations produced per second vs. the size of b. When running the sieving step on Enigma, low

Table 5.12: Results of altering the value of a vs b

A_0	A_1	B_0	B_1	$Time$	$Rels$
$-4 * 10^6$	$4 * 10^6$	1	2000	18m0s	5062/71297
$-6 * 10^6$	$6 * 10^6$	1	2000	19m30s	6862/71297
$-8 * 10^6$	$8 * 10^6$	1	2000	26m14s	8529/71297
$-10 * 10^6$	$10 * 10^6$	1	2000	43m21s	10103/71297
$-2 * 10^7$	$2 * 10^7$	1	2000	41m17s	17017/71297
$-4 * 10^7$	$4 * 10^7$	1	4000	138m54s	57503/71297
$-4 * 10^7$	$4 * 10^7$	1	8000	253m97s	71297/71297
$-6 * 10^7$	$6 * 10^7$	1	2000	103m53s	41891/71297
$-8 * 10^7$	$8 * 10^7$	1	2000	130m2s	52969/71297

values of b result in 0.002 relations per second. With the value of A at $2 * 10^6$, the value of b increases much faster, resulting in as much as 0.014 relations per second for higher values of b. In contrast, a can be as high as $8 * 10^7$ without effecting the amount of time per relation. At $8 * 10^7$, the amount of time necessary for each relation begins to increase slightly, but not at the rate that higher values of b provide. This shows that it is possible to perform line sieving with larger values of a in a significantly reduced amount of time. In fact, when compared to the 60m required for lattice sieving to process this 80-digit number, it seems it may be possible to achieve similar performance (perhaps only twice as slow) with line sieving.

It was also interesting to note that the parameters for matrix generation and processing had no effect on the algorithm. In fact, if an invalid parameter was supplied (either too small or too large) GGNFS would ignore it and calculate a better, more optimal parameter.

5.4 CrypTool Integration

CrypTool [55] is educational software that aims to assist students in learning about cryptography. CrypTool provides tools for performing encryption, decryption, generating message authentication codes (MACs), digital signatures, and many other cryptographic operations

using both modern and historical ciphers. In addition, CrypTool provides some mechanisms for performing cryptanalysis against both classical and modern ciphers, including the RSA algorithm that serves as a bedrock for modern electronic commerce.

CrypTool's RSA analysis tools include several different factoring algorithms, including trial division, Pollard's algorithm, the quadratic sieve, and several others[55]. These algorithms are very powerful against small RSA keys (less than about 100 digits), but larger keys require the more efficient Number Field Sieve. While it would be impractical for students to use a single computer to factor large RSA keys, it will be beneficial to provide a user interface for the complex algorithm. Part of the implementation effort involved working to extend the CrypTool software by providing support for the Number Field Sieve (NFS).

By extending CrypTool to support NFS, students will be able to see first-hand how the security of RSA depends on the difficulty of factoring large numbers. Similarly, students will also be able to examine how the different NFS parameters can drastically effect the amount of time necessary to factor a large number–anywhere between 15 minutes and two hours for a 50-digit number. Similarly, because students can generate RSA keys of any size in CrypTool, they will be able to feasibly generate and perform attacks against small RSA keys.

CrypTool is a C++ application written using Microsoft Visual C++ .NET 2003 and the Perl scripting language for the Windows Platform. There is a port of CrypTool for Linux [58] in progress, but it is not advanced enough for the purposes of this effort. As mentioned previously, CrypTool provides tools for performing cryptographic operations using classical and modern ciphers as well as tools for performing cryptanalysis. All of these tools are available from the CrypTool menu bar. When a particular operation is desired, CrypTool will open a dialog box for the user to provide input about the key or the type of operation to be performed, allowing for a graphical user interface to what are otherwise "black box" algorithms.

The CrypTool source code relies on the Microsoft Foundation Classes (MFC) GUI library for interacting with the user, preventing it from easily being ported to other platforms.

Nevertheless, the MFC library is a very straightforward interface for developing GUI applications. By relying on a single main menu to launch dialog boxes for each application, the CrypTool developers have ensured that it is relatively painless to introduce new functionality into CrypTool: new functionality need only be added as a menu item and the appropriate dialog box can be instantiated.

To aid in distribution, all of the libraries that CrypTool requires are statically linked to the CrypTool application, precluding users from downloading additional software–or even running a software installer. As such, any extensions to CrypTool should aim to keep with this tradition.

5.4.1 GGNFS and Msieve

Originally, the CrypTool effort was to integrate msieve and GGNFS into the CrypTool framework. Both msieve and GGNFS rely on Murphy's α approximation to generate polynomials. Both implementations provide line sieves, but msieve's line sieve uses the bucket sort algorithm outlined by Aoki and Ueda.[59] For the linear algebra step, both applications provide an implementation of the Lanczos algorithm.

In testing performed to compare the GGNFS and msieve implementations, GGNFS was found to factor a 100-digit number in 10 hours (via its lattice sieve implementation) while msieve factored the same number in a little over four days via its line sieve implementation. When compared to a commercial NFS implementation provided by the MAGMA computational algebra system, which took over six days to complete, GGNFS proved the most efficient candidate. Nevertheless, the GGNFS source code is very convoluted with little documentation and (at the time) had not been updated in several years.

Msieve, on the other hand, had a number of benefits over GGNFS:

- msieve relies on its own multi-precision implementation
- msieve does not perform lattice-based sieving as suggested by portions of the readme
- msieve implements a bucket-sort version of the line sieve, as outlined by Kazumaro

Aoki and Hiroki Ueda. [59]

- msieve is actively developed
- msieve is written to be a library that can be included by other applications (the msieve application is a demonstration of how these library functions may be used)
- the msieve code is well-commented

The primary drawback of the msieve implementation was that it is hard-coded to factor only numbers larger than 97 digits via NFS. Nevertheless, the code changes necessary to disable this check and provide parameters for smaller numbers were negligible. As a testament to msieve's performance, the msieve Web site indicates that it is used by the NFSNet project to perform the final stages of NFS.

Partway through this integration effort, it was discovered that Jason Papadopoulos has been granted commit access to the GGNFS subversion repository with the intent of merging the GGNFS and msieve code-bases. Once complete, the more efficient GGNFS sieving code will work seamlessly with the more efficient msieve matrix processing code.

5.4.2 Implementation Results

The original goal of the implementation phase of this effort was to integrate both the msieve and GGNFS code-base into CrypTool, allowing users to choose which algorithms to use with different steps of the NFS. In particular, the NFS extension would allow users to perform the polynomial selection step separate from the rest of the algorithm. Because polynomial selection is a non-deterministic function, this will allow users to record and re-use the results of this step before running the rest of the algorithm. Similarly, users would be able to see first hand how polynomial selection can affect the rest of the algorithm. Users would also have the ability to input polynomials generated using other tools to test their effectiveness. For the rest of the algorithm, users will be able to supply the NFS parameters described in Table I to monitor how the choice of parameters affects the amount of time spent running the NFS algorithm. Ultimately, users would be allowed to specify the algorithms used for

Table 5.13: NFS Implementation Parameters

General Parameters	
n	The number to factor
$f(x)$	The polynomial NFS will use
m	A value of x at which $f(x) = 0$
P_{max}	The largest prime to be used in each factor base
RFB	The size of the rational factor base
RFB_{max}	The largest value of the rational factor base
AFB	The size of the algebraic factor base
AFB_{max}	The largest value of the algebraic factor base
QCB	The size of the quadratic character base (optional)
Polynomial Selection Parameters	
α	The maximum allowed value of Murphy's α approximation in selecting the polynomial
d	The degree of the polynomial (usually 3 or 5)
Line Sieving Parameters	
A	For line sieving, the sieving interval such that $-A < a < A$

each step of the NFS algorithm in addition to specifying NFS parameters. Users would be able to store the results of each intermediate step and perform analyses of the performance of each option.

Because msieve has no external library dependencies and provides a Win32 implementation from their Web site, it was chosen as the first NFS implementation to integrate with CrypTool. Nevertheless, several roadblocks were encountered while working on this phase of the effort. In particular, the Win32 implementation of msieve and CrypTool required functions specific to their respective compilers. At the beginning of this effort, it seemed relatively simple to port the msieve code, but the resulting binary consistently failed to factor different 40-digit numbers (see Figure 5.1). In the interim, it is likely that these issues have resolved with msieve and CrypTool now supporting similar compilers. After some research on the GGNFS Web site[60], Visual Studio project files for the GGNFS suite that support various implementations of the Microsoft Visual Studio compiler were identified: Visual C++ 6.0, Visual C++ .NET 2003, Visual C++ 2005 and Visual C++ 2008. While

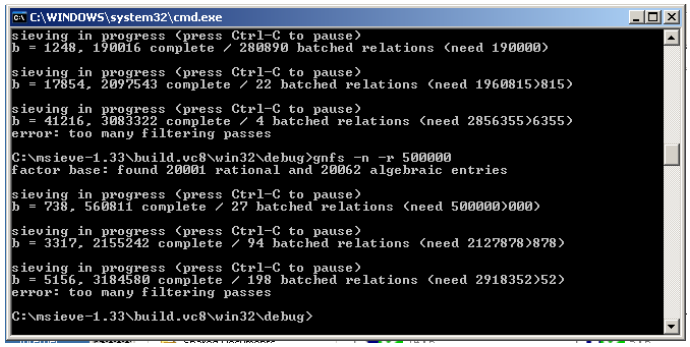

Figure 5.1: Msieve Error

work on GGNFS has begun anew under Jason Papadopoulos, there were still a number of issues to deal with:

- GGNFS relied on a different version of the GMP library than CrypTool
- The Windows patches for GMP are no longer available [61]
- The Visual Studio project files did not function easily out-of-the-box

After a good deal of work, it was possible to compile and run GGNFS and GMP in Windows. Several rounds of testing resulted in the same output and findings as testing in Linux. As such, it became possible to work on the CrypTool merge.

A CrypTool menu item was created (see Figure 5.2) and a Dialog Box was developed for NFS (see Figure 5.3), polynomial selection (see Figure 5.4), factor base generation (see Figure 5.5) and sieving (see Figure 5.6). This lead to the final road blocks in the phase of the integration effort: updating CrypTool's GMP library and integrating the polynomial selection code. Updating the GMP library proved straightforward once all references to the older library file were removed from the CrypTool project and replaced with the newer version. CrypTool continued to function without any issues, indicating the two GMP versions are binary compatible (except for the functions required by GGNFS).

The final hurdle in this phase was converting the polynomial selection code to C++ in

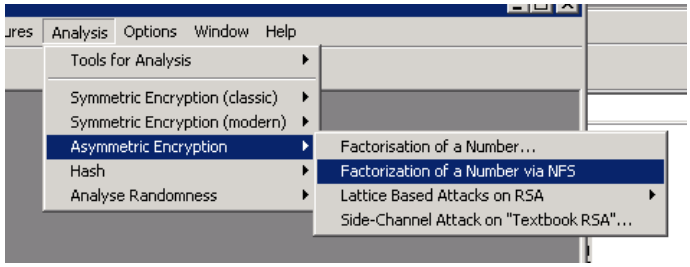

Figure 5.2: CrypTool NFS Menu

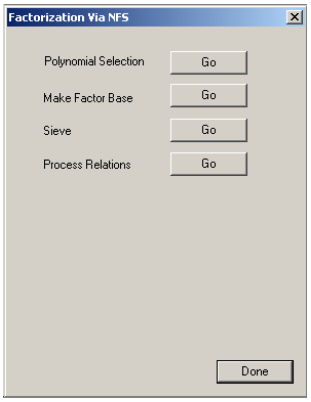

Figure 5.3: NFS Dialog

such a way that it would compile, run and output could be displayed to the dialog. As such, the current version of the CrypTool extension includes support for every NFS step up to matrix generation. The biggest obstacle in compiling the polynomial selection code was getting the LN2 identifier and other mathematical constants to compile. In the GGNFS project, the compiler does not complain about these identifiers–nor does the GCC compiler in Linux. After much searching, I found there is a $\#define$ that must be set to include math.h constants. Even though it is not explicitly set in the GCC or main GGNFS Visual C++ project, somehow it is enabled when those projects are compiled. Nevertheless, once the appropriate $\#define$ was enabled, CrypTool began to correctly compile the polynomial selection code.

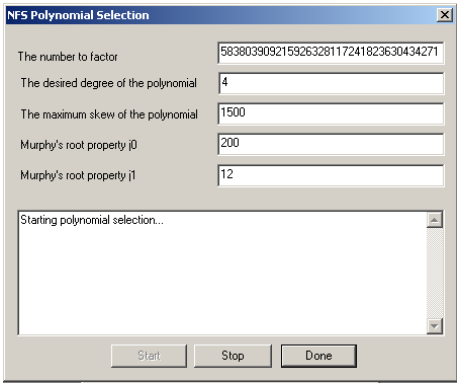

Figure 5.4: Polynomial Selection

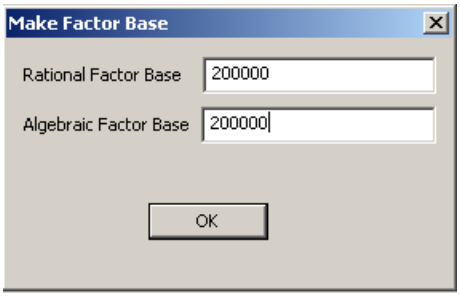

Figure 5.5: Factor Base

Finally, to allow users to cancel the polynomial selection process, the polynomial selection code was modified to be launched in a separate thread, allowing the CrypTool user interface to remain responsive while searching for appropriate polynomials. Because of all of these roadblocks, efforts focused on implementing the standalone version of the number field sieve described throughout this chapter–as well as a graphical user interface for the standalone implementation described in Section 5.5.

5.5 Graphical User Interface

With the goal of eventually integrating the standalone number field implementation with the CrypTool interface, a graphical user interface was added to this implementation. By using a Windows implementation of the GCC tools (MinGW [62]), all of the underlying

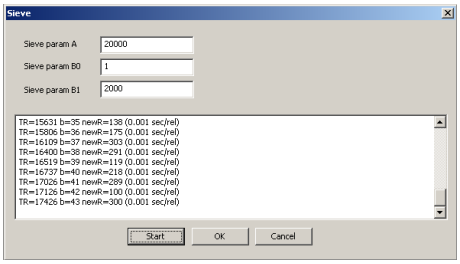

Figure 5.6: Sieving

libraries were compiled for the Microsoft Windows platform, resulting in a command-line implementation of the number field sieve for Windows. By modifying the command-line implementation using the wxWidgets library [63], a cross-platform graphical user interface was developed allowing users on Windows, Linux, and Mac OS X to use the implementation developed to support this thesis for learning purposes, see Figure 5.7. Specifically,

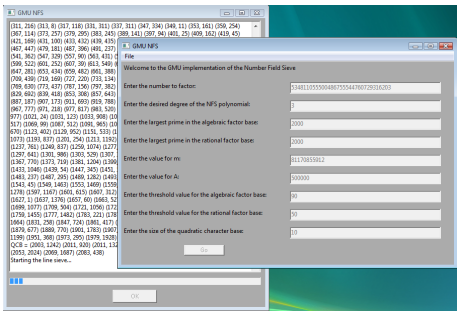

Figure 5.7: GMU NFS GUI

any platform with GNU compiler tools and a wxWidgets library will be able to run this implementation. By using the GNU tools to port all of the underlying libraries, a future effort to integrate these GNU-compiled binaries with the Microsoft Visual Studio-compiled CrypTool may be more successful.

Chapter 6: Summary and Conclusions

One of the primary goals at the beginning of this thesis was to identify a document that outlined the number field sieve in a manner accessible to those without multiple years of research using or implementing the number field sieve algorithm. This document provided that insight–giving readers a simple source to gain an understanding of how to implement the number field sieve. Couple with this insight is an implementation of the number field sieve using C++, allowing individuals to parse through the C++ code an gain an understanding of how a simple implementation of the number field sieve can function. Due to the modular nature of the C++ language, it is expected that future work will include updating this implementation to improve performance and provide modular mechanisms for swapping out individual components of the number field sieve–as simple as adding implementations that use alternate libraries or implementations of alternate algorithms (e.g., the Block Wiedeman algorithm in place of Gaussian elimination for the matrix step).

This implementation was developed with scalability in mind–specifically within the sieving step. Minor modifications to the sieving step will allow it to use hard drive space on the system to store and access all of the data components used in the line sieving algorithm. By minimizing the number of points where 32-bit or 64-bit integer values, this implementation will avoid many of the pitfalls associated with other implementations–specifically a hard limit where integer overflows begin to affect functionality.

Completion of the graphical user interface provides additional utility for this implementation. The GUI provides a learning environment where students can easily modify parameters to the algorithm–and even modify the source code of the implementation. By using a cross-platform framework, students will not be limited to a specific version of an OS or specific hardware to run this particular implementation of NFS. Additionally, by relying solely on open source software libraries, this implementation is fully extensible and

allows students to extend or modify the existing codebase to improve upon it or develop alternative implementations of the various steps of the number field sieve. In fact, one of the primary issues that students often run into while studying the number field sieve is that the algorithm is extremely complex and, in most cases, there is insufficient time to implement more than one specific aspect of the algorithm. As noted in [41], many projects require students to develop *test input* to analyze their implementations. By using this implementation of NFS, students will be able to generate input from an running implementation of NFS and–where possible–integrate their code with this framework and, over time, develop a GMU implementation of NFS that encompasses all aspects of the number field sieve. Additionally, through the pseudocodes outlined in this thesis and the implementation provided, students will also be able to identify areas where software implementations of the number field sieve can benefit from improved performance.

While there are numerous benefits from using the standalone implementation developed as part of this thesis, integrating it with CrypTool will provide a GUI interface consistent with other learning environments within the cryptographic engineering space. With further modifications of the source of this implementation–specifically those that will allow it to link and build with the Microsoft Visual Studio compilers–students could use the CrypTool interface to better understand weaknesses of RSA by performing the steps necessary to generate small RSA keys, encrypt messages, and attack those messages all within the same user interface.

Bibliography

Bibliography

[1] S. Singh. (2007, April) Caesar cipher. [Online]. Available: http://www.simonsingh.net/The_Black_Chamber/caesar.html

[2] K. Gaj. (2009, November) ECE 646 - lecture 6. [Online]. Available: http://teal.gmu.edu/courses/ECE646/viewgraphs_F08/lecture6_historical_3.pdf

[3] B. J. Copeland, "Colossus: its origins and originators," *IEEE Annals of the History of Computing*, vol. 26, no. 4, pp. 38–45, October-December 2004.

[4] National Institute of Standards and Technology. (1999, October) Data encryption standard. [Online]. Available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[5] W. Diffie and M. Hellman, "New directions in cryptography," in *IEEE Transactions on Information Theory*, vol. 22, Nov 1976, pp. 644–654.

[6] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, February 1978.

[7] C. Monico. (2011, January) GPL'd implementation of the general number field sieve. [Online]. Available: http://www.math.ttu.edu/~cmonico/software/ggnfs/

[8] P. L. Jensen, "Integer factorization," Masters Thesis, University of Copenhagen, Copenhagen, Denmark, Dec 2005.

[9] C. Card. (2011, March) Number field sieve implementation. [Online]. Available: http://sourceforge.net/projects/factor-by-gnfs/

[10] J. Papadopoulos. (2009, August) Integer factorization source code. [Online]. Available: http://www.boo.net/~jasonp/qs.html

[11] J. Buhler, H. Lenstra Jr., and C. Pomerance, "Factoring integers with the number field sieve," in *The development of the number field sieve*, ser. Lecture Notes in Mathematics, A. Lenstra and H. Lenstra Jr., Eds. Berlin: Springer-Verlag, 1993, vol. 1554, pp. 50–94.

[12] M. E. Briggs, "An introduction to the general number field sieve," Masters Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, April 1998.

[13] M. Case. (2003) A beginners guide to the general number field sieve. [Online]. Available: http://islab.oregonstate.edu/koc/ece575/03Project/Case/paper.pdf

[14] Deutsche Bank. (2011, April) Cryptool - education tool for cryptography and cryptanalysis. [Online]. Available: http://www.cryptool.de

[15] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. New York, NY: CRC Press, 1997.

[16] K. Gaj. (2008, February) ECE 746 - lecture 3. [Online]. Available: http://teal.gmu.edu/courses/ECE746/viewgraphs_S08/lecture3_math_2.pdf

[17] ——. (2009, November) ECE 646 - lecture 9. [Online]. Available: http://ece.gmu.edu/coursewebpages/ECE/ECE646/F10/viewgraphs_F09/F09_lecture9_RSA_basics_6.pdf

[18] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*. Upper Saddle River, NJ: Prentice Hall, 2002.

[19] T. Dierks and C. Allen. (1999, January) RFC 2246: The TLS protocol version 1.0. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[20] RSA Laboratories. The RSA challenge numbers. [Online]. Available: http://www.rsa.com/rsalabs/node.asp?id=2093

[21] K. Gaj. (2011, November) ECE 646 - lecture 8. [Online]. Available: http://ece.gmu.edu/coursewebpages/ECE/ECE646/F11/viewgraphs_F11/ECE646_lecture8_RSA_2.pdf

[22] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. (2007, March) NIST special publication 800-57: Recommendations for key management–part 1: General (revised). [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf

[23] E. W. Weisstein. Pollard rho Factorization Method. From MathWorld–a Wolfram web resource. [Online]. Available: http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html

[24] R. Bachimanchi, "FPGA and ASIC implementation of rho and p-1 methods of factoring," Masters Thesis, George Mason University, Fairfax, VA, USA, May 2007.

[25] A. K. Lenstra, "Integer factoring," in *Designs, Codes and Cryptography*, vol. 19. Springer Netherlands, Mar 2000, pp. 101 – 128.

[26] J. M. Pollard, "Factoring with cubic integers," in *The Development of the Number Field Sieve*, ser. Lecture Notes in Mathematics, vol. 1554/1993. Springer Berlin / Heidelberg, 1993, pp. 43 – 49.

[27] A. K. Lenstra and H. W. Lenstra, Eds., *The development of the number field sieve*, ser. Lecture Notes in Mathematics, vol. 1554. Berlin Heidelberg: Springer-Verlag, 1993.

[28] D. J. Bernstein and L. A. K, "A general number field sieve implementation," in *The Development of the Number Field Sieve*, ser. Lecture Notes in Mathematics, vol. 1554/1993. Springer Berlin / Heidelberg, 1993, pp. 103 – 126.

[29] R. M. Huizing and M. Huizing, "An implementation of the number field sieve," *Experimental Mathematics*, vol. 5, pp. 231–253, 1996.

[30] W. Ekkelkamp, "Predicting the sieving effort for the number field sieve," in *Lecture Notes in Computer Science*, vol. 5011/2008. Springer Berlin / Heidelberg, May 2008, pp. 167 – 179.

[31] B. Murphy, "Polynomial selection for the number field sieve integer factorisation algorithm," Ph.D. dissertation, Australian National University, Jul 1999.

[32] T. Kleinjung, "On polynomial selection for the general number field sieve," *Mathematics of Computation*, vol. 75, no. 256, pp. 2037–2047, Jun 2006.

[33] W. Geiselmann and R. Steinwandt, "Non-wafer-scale sieving hardware for the NFS: Another attempt to cope with 1024-bit," in *Advances in Cryptology – EUROCRYPT 2007*, ser. Lecture Notes in Computer Science, vol. 4515/2007. Springer Berlin / Heidelberg, Jun 2007, pp. 466–481.

[34] J. M. Pollard, "The lattice sieve," in *The Development of the Number Field Sieve*, ser. Lecture Notes in Mathematics, vol. 1554. Springer Berlin / Heidelberg, 1993, pp. 43–49.

[35] R. A. Golliver, A. K. Lenstra, and K. S. McCurley, "Lattice sieving and trial division," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, L. M. Adleman and M. D. Huang, Eds., vol. 877/1994. Springer Berlin / Heidelberg, 1994, pp. 18 – 27.

[36] J. Franke and T. Kleinjung, "Continued fractions and lattice sieving," in *Special-Purpose Hardware for Attacking Cryptographic Systems SHARCS*, 2005.

[37] D. H. Wiedemann, "Solving sparse linear equations over finite fields," in *IEEE Transactions on Information Theory*, ser. IT-32, 1986, pp. 54–62.

[38] D. Coppersmith, "Solving homogeneous linear equations over GF(2) via block wiedemann algorithm," in *Math. Comput.*, ser. 62, vol. 205, 1994, pp. 330–350.

[39] P. L. Montgomery, "A block lanczos algorithm for finding dependencies over GF(2)," in *Advances in Cryptology – EUROCRYPT 95*, ser. Lecture Notes in Computer Science, vol. 921/1995. Springer Berlin / Heidelberg, 1995, pp. 106–120.

[40] J.-M. Couveignes, "Computing a square root for the number field sieve," in *The Development of the Number Field Sieve*, ser. Lecture Notes in Mathematics, vol. 1554/1993. Springer Berlin / Heidelberg, 1993, pp. 95 – 102.

[41] C. Anand, "Factoring of large numbers using number field sieve - the matrix step," December 2007.

[42] T. Izu, J. Kogure, and T. Shimoyama. (2007, Sep) CAIRN 3: An FPGA implementation of the sieving step with the lattice sieving. [Online]. Available: http://www.hyperelliptic.org/tanja/SHARCS/talks07/sharcs2007_cairn3_3.pdf

[43] ——, "A status report: An implementation of a sieving algorithm on a dynamic reconfigurable processor (extended abstract)," in *Special-Purpose Hardware for Attacking Cryptographic Systems SHARCS*, 2005.

[44] ——, "An FPGA implementation of the sieving step in the number field sieve method," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science, vol. 4727. Springer Berlin, Aug 2007, pp. 364–377.

[45] K. Mohammed, "Hardware implementation of the elliptic curve method of factoring," Masters Thesis, George Mason University, Fairfax, VA, USA, August 2006.

[46] G. Southern, C. Mason, L. Chikkam, P. Baier, and K. Gaj, "FPGA implementation of high throughput circuit for trial division by small primes," in *Proceedings of the Special Purpose Hardware for Attacking Cryptosystems, SHARCS 2007*, 2007.

[47] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, R. Bachimanchi, and M. Rogawski, "Area-time efficient implementation of the elliptic curve method of factoring in reconfigurable hardware for application in the number field sieve," in *IEEE Transactions on Computers*, vol. 59, Sep 2010, pp. 1264–1280.

[48] A. Shamir, "Factoring large numbers with the TWINKLE device (extended abstract)."

[49] A. Shamir and E. Tromer, "Factoring large numbers with the TWIRL device," in *Crypto 2003*, ser. Lecture Notes in Computer Science, vol. 2729. Springer-Verlag, 2003, pp. 1 – 26.

[50] W. Geiselmann and R. Steinwandt, "A dedicated sieving hardware," in *PKCS 2003*.

[51] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke, "SHARK a realizable special hardware sieving device for factoring 1024-bit integers," in *SHARCS*, 2005.

[52] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*. New York, NY: Springer, 2005.

[53] E. W. Weisstein. Gaussian elimination. From MathWorld–a Wolfram web resource. [Online]. Available: http://mathworld.wolfram.com/GaussianElimination.html

[54] Darmstadt University of Technology. (2010, February) Lidia a c++ library for computational number theory. [Online]. Available: http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/

[55] CrypTool. [Online]. Available: http://www.cryptool.com/

[56] R. Mitton. (2011, November) Very sleepy. [Online]. Available: http://www.codersnotes.com/sleepy

[57] P. E. Black. (2009, September) binary GCD in Dictionary of Algorithms and Data Structures. [Online]. Available: http://www.nist.gov/dads/HTML/binaryGCD.html

[58] CrypTooLinux. [Online]. Available: http://www.cryptoolinux.net

[59] K. Aoki and H. Ueda, "Sieving using bucket sort," in *Advances in Cryptology - ASIACRYPT 2004*, ser. Lecture Notes in Computer Science, vol. 3329. Springer Berlin, Nov 2004, pp. 92–102.

[60] C. Monico and J. Papadopoulos. GNU general number field sieve. [Online]. Available: http://sourceforge.net/projects/ggnfs

[61] Wingmp. [Online]. Available: http://na-inet.jp/na/bnc/wingmp.html

[62] MinGW. (2011) MinGW: Minimalist GNU for Windows. [Online]. Available: http://www.mingw.org

[63] wxWidgets. (2011, July) wxWidgets: cross-platform GUI. [Online]. Available: http://www.wxwidgets.org

Curriculum Vitae

Theodore K Winograd received his Bachelor of Science in Computer Engineering in 2004 from Virginia Tech. He currently works as an Associate at Booz Allen Hamilton, providing cyber security consulting services to Booz Allen Hamilton clients. He began his graduate degree at George Mason for Computer Engineering in 2005 and is currently a member of the Cryptographic Engineering Research Group (CERG) at George Mason University.