Wesleyan University

Factorization and Collision Algorithms in Algebraic Cryptography

by

Joshua Murphy

Faculty Advisor: David Pollack

Associate Professor of Mathematics

A Thesis in Mathematics
submitted in partial fulfillment of the
requirements for the degree of Master of Arts
at Wesleyan University

Middletown, CT April, 2017

Contents

Acknowledgements

First, I would like to thank my advisor Professor Pollack. Throughout this process, he has been incredibly helpful and supportive, either with mathematical insight or emotional support. I could not have done this without him, and I am deeply grateful for his time and commitment.

I would also like to thank Professor Chan and Professor Rasmussen for being on my committee. Their time put in to provide feedback and consider my work is much appreciated. I also want to thank the whole Wesleyan Mathematics Department, for their help throughout these three years.

Finally, a special thanks to my Mother for her endless support and unwavering patience.

Chapter 1

Introduction

1.1 Birth of Algebraic Cryptography

In 1976, Diffie and Hellman revolutionized the field of cryptography with their paper "New Directions in Cryptography" [5]. The key insight that Diffie and Hellman brought to the table was the notion of a public key. They defined a public key cryptosystem as a cryptosystem with what they referred to as a one-way function $f : X \to X$, and a trapdoor. A one way function is a function that is easy to compute on its domain. However, without any additional information $f^{-1}(x)$ is difficult to find. The so-called trapdoor is an extra piece of information that allows one to easily evaluate $f^{-1}(x)$.

Public key cryptosystems are applied as follows: for a particular chosen $x \in X, f(x)$ and the function f itself are assumed to be public information, whereas x itself is kept private. In this case, x is in fact the information those using the encryption desire to keep secure. Evaluating $f^{-1}(x)$ is difficult, unless one possesses the so-called trap door, which allows for the recovery of $x = f^{-1}(x)$ easily. The party that has the trap door is the one receiving the information.

In that way, once the message x is chosen and made public, the receiving party can use the trap door to compute $f^{-1}(x)$ and obtain the original message. As presented, these cryptosystems rely on a hardness criteria for evaluating inverses. In the context of this document, we will say that computing the inverse of a function is hard if the algorithms that do it are sub-exponential or worse.

In the standard presentation of cryptography, the public key cryptosystem is explained with the use of Alice, Bob, and Eve. Alice and Bob want to share a secret despite having only insecure communication. Their adversary Eve wants to find out the information that they are sharing.. Since Alice and Bob may only communicate insecurely, all information they exchange is seen by Eve. The problem of Alice and Bob communicating a message securely over insecure channels initially appeared impossible, however Diffie and Hellman's idea to use these one-way functions makes this possible.

To be clear, it is important to note that there is no proof that such one-way functions do in fact exist [6]. This is not as concerning as it may seem, as mathematicians and computer scientists believe enough in the difficulty of the problems that encryption methods are based on. Further, such a proof would be a massive undertaking, as it implies $P \neq NP$. However, there are a variety of proposed one-way functions used by present-day public key encryption algorithms. Note that their use relies on the assumption that the computation of $f^{-1}(x)$ in that particular case is computationally difficult, rather than any guarantee to that effect.

The Diffie-Hellman (D-H) key exchange was the first published encryption system utilizing a public key, and is based on the discrete logarithm problem (DLP). Let G be a group with operation $\star$. Then, the *DLP* for G is to find, for any two particular elements $g, h \in G$, an integer x such that

$$\underbrace{g \star g \star g \ldots \star g}_{x \text{ times}} = h.$$

We note that the D-H key exchange is not a true public key cryptosystem, as it does not allow Alice to share a particular piece of information with Bob. Instead, it allows them to agree on a shared piece of information, namely $A' \equiv B' \pmod{p}$. On the other hand, we will see later that RSA allows Alice to share a particular message m with Bob securely.

In the paper [5], Diffie and Hellman use the DLP for a finite field $\mathbb{F}_p$. In this case, g is a primitive root of $\mathbb{F}_p$, and h is some nontrivial element. Then, the problem is to find x so that

$$g^x \equiv h \pmod{p}.$$

The D-H key exchange works as follows. As discussed before, Alice and Bob want to share a secret through insecure channels, and are against their adversary Eve. First, Alice and Bob choose a large prime p, and some non-zero integer g mod p. These p and g are publicly available. Then, Alice and Bob choose secret integers a and b respectively, and compute both

$$A \equiv g^a \pmod{p},$$

and its counterpart

$$B \equiv g^b \pmod{p}.$$

Then, they send the values of A and B to each other – note that Eve sees A and B as well. They then find

$$A' \equiv B^a \pmod{p},$$

and similarly

$$B' \equiv A^b \pmod{p}.$$

It is clear then that $A' \equiv B' \bmod p$. This is Alice and Bob's shared secret, and importantly, Eve has no way of knowing, as the values of a and b were kept secret.

What Eve does know are the values of A and B, and so g^a and g^b, and she also knows g and p. Thus, if Eve can solve the DLP, she can find a and b, in which case g^{ab} is easy to compute. In reality, what Eve needs to solve is slightly different from the DLP. This problem, known as the Diffie-Hellman problem (DHP), is computing the value of $g^{ab} \pmod{p}$ from known values of $g^a \pmod{p}$ and $g^b \pmod{p}$. Note that it is unknown whether or not the DLP and DHP are equivalent, though certainly the DLP implies the DHP.

The first true public key cryptosystem, RSA, is named for its authors: Ron Rivest, Adi Shamir, and Leonard Adleman in [18]. RSA works in the following way. First, Bob chooses secret primes p and q, and an exponent e with $\gcd(e, (p-1)(q-1)) = 1$, and shares $(N = pq, e)$ as his public key. Then, Alice chooses an integer m which encodes her message, and uses Bob's public key to compute $c \equiv m^e \pmod{N}$, and sends c, the ciphertext, to Bob. Then, Bob finds d so that $ed \equiv 1 \pmod{(p-1)(q-1)}$. He then computes $m' \equiv c^d \pmod{N}$.

Seeing that $m' = m$ is not obvious. The group $(\mathbb{Z}/N\mathbb{Z})^*$ has order $\phi(N) = (p-1)(q-1)$.

Lagrange's theorem thus tells us that for all $x \in (\mathbb{Z}/N\mathbb{Z})^*$, $x^{(p-1)(q-1)} \equiv 1 \pmod{N}$. Further, since $ed \equiv 1 \pmod{(p-1)(q-1)}$, there exists an $s \in \mathbb{Z}$ such that

$$ed - s(p-1)(q-1) = 1.$$

But then, using Lagrange's theorem, we have that (where the congruences are taken $\pmod{N}$),

$$\begin{aligned} c^d &\equiv (m^e)^d \\ &\equiv m^{ed} \\ &\equiv m^{1+s(p-1)(q-1)} \\ &\equiv m \cdot m^{s(p-1)(q-1)} \\ &\equiv m \cdot (m^{(p-1)(q-1)})^s \\ &\equiv m. \end{aligned}$$

Now that we understand how RSA works, we discuss what Eve can do. In attempting to break the RSA public key cryptosystem, Eve knows everything shared publicly, but does not know the values of p or q. If Eve can factor N, then clearly she can decrypt m. However, she does not need to factor N; what she actually must do is solve congruences of the form $x^e \equiv c \pmod{N}$, and it is possible that there is a way to do this efficiently with only Eve's information, though such a method is unknown. This calls back to the previously mentioned statement that we are unsure whether or not one-way functions actually exist.

The above discussion of RSA makes it clear exactly how critical the ability to factor large numbers is, as RSA is both one of the best known public key cryptosystems and it is still used today. This importance gives rise to Chapter

2 of this paper, which focuses on modern factorization algorithms. In a similar fashion, though the D-H key exchange is neither as well-known nor as prevalent, it is nonetheless still a relevant method of encryption. In practice, most methods of attacking the D-H key exchange use what is known as a collision algorithm. This topic is the focus of Chapter 3.

The dynamic between the security of RSA and the ability to factor large numbers points to the heart of an interesting issue in the field. Those who want to break the security of RSA are attempting to push the bounds on the size of numbers that are reasonably able to be factored. The reason for this is straightforward. If the use of RSA presumes adversaries can only factor up to a certain number of digits in a realistic amount of time, they can use larger numbers as the basis of their security. Conversely, those who want to keep information secure are doing the same thing - attempting to push at the size of numbers they can factor. If they can factor larger numbers, they must assume that they must choose even larger numbers for RSA to remain secure, and so on. While doing this, they are also attempting to write better encryption algorithms that use a field that would make it more difficult to break the algorithm via brute force, such as making use of the theory of elliptic curves. This creates an arms race in the information security field, where both parties are attempting to create faster algorithms that can deal with larger numbers to exactly opposite ends.

Finally, as an interesting aside, in this introduction we have discussed Diffie and Hellman, as well as Rigest, Shamir, and Adleman as the first to discover their particular cryptosystems. However, it has been discovered that the D-H key exchange algorithm and RSA were both discovered prior to their publication by members of the British intelligence community [6].

1.2 Background Information

Throughout this paper, many of the topics discussed require only the background knowledge from a standard first year graduate sequence in mathematics. There are certain topics, such as the number field sieve in particular, which likely require some further knowledge. We aim to summarize that supplemental information here.

1.2.1 Run time

The run time of a program is, as one would hope, the time it takes for a program to run. In this document we will be concerned with both average run time and worst case run time. We often think of run times as functions of the size of the input. We look at two ways of talking about run time.

The O-notation was introduced by Paul Bachmann (1894). We say that $f(x) = O(g(x))$ if and only if there exists some $M \in \mathbb{R}$ and $x_0 \in \mathbb{R}$ such that $|f(n)| < M|g(n)|$ for all $x \geq x_0$ [8]. When dealing with algorithms, we say that, for example, an algorithm A that is $O(1)$ runs in constant time. In such a case, we would say A is $O(1)$. That is, regardless of the size of the input, A's run time will have a constant bound for any input. If A is $O(N)$, where N is the size of the input of A, then A is linear.

Now that the concept of run times have been established, we present some common algorithms and their run times. The Euclidean algorithm to evaluate $\gcd(a, b)$ has run time $O(\log b)$, which is very fast. In particular, we say that the Euclidean algorithm runs in polynomial time (of $\log b$). On the other hand, something that appears relatively simple such as factoring by trial division, is actually quite slow, with run time $O(\sqrt{n})$ as we discuss in the next section. This means

trial division has exponential time growth. To get a sense for how woefully slow trivial division is, to factor a number of one hundred digits, which we will see later is a very relevant length, on an Intel i7 4770K processor would take somewhere around $2.4 * 10^{81}$ years to do so via trial division.

Another representation of run time uses the L notation. We define

$$L_x[v, \lambda] = \exp(\lambda(\log x)^v(\log\log x)^{1-v}),$$

for $x, v, \lambda \in \mathbb{R}$ with $x > e$ [12]. To parse L notation, note that the critical parameter is v, the most important things to keep in mind are as follows:

If an algorithm has $v = 1$, then we have $L_N[1, \lambda] = N^\lambda$ for some $\lambda > 0$, and so the algorithm is exponential in the size of N.

If an algorithm has $v = 0$, then we have $L_N[0, \lambda] = (\log N)^\lambda$ for some $\lambda > 0$, and so the algorithm is said to be polynomial in the size of N (a polynomial of $\log N$).

Finally, any algorithm that runs in time $L_N[m, \lambda]$, for $0 < m < 1$ and some $\lambda > 0$ are said to run in sub-exponential time. Of note, most modern factorization algorithms are sub-exponential in nature, with many in particular having run time $L_N[\frac{1}{2}, \lambda]$.

1.2.2 Algebraic number theory

To understand the description of the number field sieve presented in this document, some familiarity with algebraic number theory is necessary. We note that the information below is far from exhaustive, and rather contains information we

found relevant to the sieve. The content of this section draws heavily from [13].

We assume that all rings are commutative with $0 \neq 1$.

Let B/A be a ring extension. Then, $b \in B$ is defined to be *integral over* A if it is a root of a monic polynomial in $A[x]$. We say B is *integral over* A if every element of B is integral over A. Further, the set of elements of B integral over A is denoted $\bar{A}$, the *integral closure* of A in B, and is itself a ring. Finally, A is integrally closed in B if every element of B which is integral over A belongs to A itself. If A is an integral domain, it is called an *integrally closed domain* if it is integrally closed in its field of fractions.

For another useful fact, suppose A is an integrally closed domain, F is its field of fractions, and B the integral closure of A in L, where L/F is a field extension. Then, B is integrally closed, and if L/F is algebraic, L must be the field of fractions of B.

In particular, define a number field to be a finite extension of $\mathbb{Q}$. For a number field K, the integral closure of $\mathbb{Z}$ in K is called the ring of integers of K, denoted $\mathcal{O}_K$. Also, $\mathcal{O}_K$ is an integrally closed domain, with field of fractions K.

A ring R is *Noetherian* if every ideal of R is finitely generated as an R-module. Another way to see if a ring is Notherian is whether if it satisfies the following equivalent conditions:

(1) The ascending chain condition on ideals of R: if $\mathfrak{a}_1 \subseteq \mathfrak{a}_2 \subseteq \ldots$ is a chain of ideals of R, then there an $m \in \mathbb{Z}$ with $m \geq 1$ such that $\mathfrak{a}_k = \mathfrak{a}_m$ for all $k \geq m$.

(2) Every nonempty set of ideals of R contains a maximal element under inclu-

sion.

We then have that $\mathcal{O}_K$ is a Noetherian ring, and non-zero ideals of $\mathcal{O}_K$ are free $\mathbb{Z}$-modules of rank n, where $n = [K : \mathbb{Q}]$.

Define an integrally closed Noetherian domain in which every nonzero prime ideal is maximal to be a *Dedekind domain*. Note that every PID is a Dedekind domain. Further, $\mathcal{O}_K$ is a Dedekind domain. This leads us to a relevant theorem regarding factoring ideals.

Theorem 1. *Let $\mathcal{O}$ be a Dedekind domain. Each nonzero ideal $\mathfrak{a}$ of $\mathcal{O}$ admits a factorization $\mathfrak{a} = \mathfrak{p}_1 \dots \mathfrak{p}_r$ into nonzero prime ideals of $\mathcal{O}$. This factorization is unique up to the order of the factors.*

Note that a Dedekind domain is not necessarily a unique factorization domain. So, even though elements may not factor uniquely up to units, we do have a concept of factoring ideals. This notion of factoring ideals is critical in the number field sieve.

1.2.3 Smooth Numbers

Define a number whose prime factors are all less than or equal to some integer B to be a *B-smooth number*. Define the function $\psi(X, B)$ to be equal to the number of B-smooth integers n with $1 < n \leq X$ [6].

We present a theorem due to Canfield, Erdős, and Pomerance in [4] regarding the behavior of ψ.

Theorem 2. *Let $\epsilon \in \mathbb{R}$ be fixed so that $0 < \epsilon < 1/2$, and let X and B increase*

together so that

$$(\ln X)^{\epsilon} < \ln B < (\ln X)^{1-\epsilon}.$$

Then,

$$\psi(X, B) = X \cdot u^{-u(1+o(1))}.$$

Here, $u = \frac{\ln X}{\ln B}$ *for convenience.*

To be precise, we are using o notation for the first time. We say that $f(x) = o(g(x))$ if $f(x)/g(x) \to 0$ as $x \to \infty$.

Define a function $L(X) = e^{\sqrt{(\ln X)(\ln \ln X)}}$. Then, the above theorem gives rise to the following.

Corollary 1. *For a fixed value of* c *with* $0 < c < 1$*,*

$$\psi(X, L(X)^c) = X \cdot L(X)^{-(1/2c)(1+o(1))},$$

as $x \to \infty$*.*

In the use of B-smooth numbers in our factorization algorithms that appear in Chapter 2, we need to find at least as many as $\pi(B)$ B-smooth numbers. Here, $\pi(B)$ is the prime-counting function. To do that, in turns out that if we take B to be roughly $L(N)^{1/\sqrt{2}}$, then $\psi(X, B)$ should produce enough B-smooth numbers to successfully complete the algorithm [6].

Chapter 2

Factorization Algorithms

2.1 Introduction

As discussed in the introduction, factoring large numbers, and so factorization algorithms, are a vital component of the modern cryptographic landscape. That said, factorization of numbers is a question that has been thought about for many centuries.

The most basic attempt at factoring a number is trial division. To perform trial division on an integer N, one tests whether or not the integers $2, 3, \ldots, \lfloor\sqrt{N}\rfloor$ divide n. This naive approach has run time $O(\sqrt{N})$. There are some simple improvements that can be made to trial division. For example, it is clear that only prime numbers must be checked, which gives approximate run time $O(\pi(N))$, where $\pi(N)$ is counts the number of primes less than N. We note that we are really measuring the number of operations required here, which changes as numbers increase in size. However, we are only off by a factor of $\log N$. Although trial division is reasonable for smaller N, recall that the security of RSA is based on the difficulty of factoring large N. So, desirable factorization algorithms must

able to factor values of N much larger than trial division is suitable for.

2.2 Pollard's $p-1$ algorithm

Let $N = pq$, where we aim to find the prime factors p and q, and suppose we have an integer L with the following properties:

$$p-1 \quad \text{divides} \quad L, \quad \text{and} \quad q-1 \quad \text{does not divide} \quad L.$$

Then, there exist $i, j, k \in \mathbb{Z}$ with $0 < k < q-1$ such that

$$L = i(p-1), \quad \text{and} \quad L = j(q-1) + k.$$

Let $a \in \mathbb{Z}$ be randomly chosen, and apply Fermat's little theorem. We get

$$a^L = a^{i(p-1)} = a^{(p-1)i} \equiv 1^i \equiv 1 (\mathrm{mod}\, p),$$

$$a^L = a^{j(q-1)+k} = a^{(q-1)j} a^k \equiv 1^j \cdot a^k \equiv a^k (\mathrm{mod}\, q).$$

Here we are assuming that neither p nor q divide a, which is very likely as N large implies both p, q large.

Since k is non-zero, it is unlikely that $a^k \equiv 1 (\mathrm{mod}\, q)$. Thus, there is a high likelihood that p divides $a^L - 1$, but q does not divide $a^L - 1$. But then in fact we have found a factor of N, since we can recover p as

$$p = \gcd(a^L - 1, N).$$

The difficulty then, lies with finding such an L with the proper divisibility

conditions on our prime factors. The idea Pollard came up with [14] to resolve this issue is that if $p-1$ is the product of only small primes, then $p-1$ will divide $n!$ for some reasonably sized n.

So, for each $n = 2, 3, 4, \ldots$ we choose some a and compute

$$\gcd(a^{n!} - 1, N).$$

There are 3 possible outcomes. If the greatest common divisor is 1, move on to the next value of n. If the greatest common divisor equals N, then we can move to another value of a (or stop, if we think this algorithm will not work for N). If the result is between 1 and N, then the greatest common divisor is a non-trivial factor of N, which is exactly what we wanted.

So, in the particular case where one of N's factors is a product of many small primes, Pollard's $p-1$ algorithm is an efficient method for factoring N. One important take away of this algorithm involves the choice of primes p and q in the RSA algorithm discussed in the introduction. Perhaps unintuitively, the $p-1$ algorithm shows that whether or not $p-1$ and $q-1$ factor into small primes is relevant to factoring $N = pq$. The lesson from the $p-1$ algorithm can then be framed as follows: when choosing small primes p and q for RSA, one must be wary that $p-1$ and $q-1$ are not B-smooth numbers for some suitably small B. Relevant sizes of B itself depend on many things such as computation power and the nature of the algorithm it is being applied in.

2.2.1 Example factorization using $p-1$ algorithm

We aim to use Pollard's $p-1$ factorization algorithm to factor $N = 92483$. We begin by computing $2^{a!} - 1 \pmod{92483}$, for $a = 2, 3, \ldots$. For each of these values,

we then find $\gcd(2^{a!}-1, 92483)$. If the greatest common divisor is 1, we continue, and if it's not 1 then we have found a non-trivial factor of N. In the following computations, we omit $a = 2, 3, 4, 5$, however in each case $\gcd(2^{a!}-1, N) = 1$.

$$2^{6!} - 1 \equiv 83567 \pmod{92483}$$
$$\gcd(83567, 92483) = 1$$

$$2^{7!} - 1 \equiv 4488 \pmod{92483}$$
$$\gcd(4488, 92483) = 1$$

$$2^{8!} - 1 \equiv 73677 \pmod{92483}$$
$$\gcd(73677, 92483) = 1$$

$$2^{9!} - 1 \equiv 8028 \pmod{92483}$$
$$\gcd(8028, 92483) = 1$$

$$2^{10!} - 1 \equiv 6681 \pmod{92483}$$
$$\gcd(8028, 92483) = 1$$

$$2^{11!} - 1 \equiv 60421 \pmod{92483}$$
$$\gcd(60421, 92483) = 23$$

This final line tells us that $p = 23$ is a nontrivial factor of $N = 92483$, and

23 is prime. The other factor is then $q = N/p = 92483/23 = 4021$, which is also prime. Note that the exponent 11! worked here, since $p = 23$ so $p - 1 = 22 = (2)(11)$, a product of small primes, all of which divide 11!. On the other hand, $q - 1 = 4020 = (2^2)(3)(5)(67)$, which is not a product of small primes.

2.3 Factorization by difference of squares

Recall that $a^2 - b^2 = (a+b)(a-b)$. Then, to obtain a factorization of N, we need only find $x, y \in \mathbb{Z}$ such that $N + y^2 = x^2$. Then, we have

$$N = x^2 - y^2 = (x + y)(x - y).$$

The difficulty in this method lies in the fact that when N is large, it is hard to find such a y. Often such a factorization for some multiple kN instead will work to factor N. That is,

$$kN = x^2 - y^2 = (x + y)(x - y).$$

In this case, it is often true that N shares a nontrivial factor with each of $(x + y)$ and $(x - y)$. Then, taking $\gcd(N, x + y)$ and $\gcd(N, x - y)$ will allow the recovery of such a factor. Note that this process is identical to finding distinct $x, y \in \mathbb{Z}$ such that $x \equiv y \, (mod \, N)$.

The notion that we can take this conceptually simple approach and broaden it is a genius idea. Our goal will be to search for pairs (x, y) such that $x^2 \equiv y^2 \pmod{N}$, but x is not congruent to $\pm y \pmod{N}$. The process of doing this and the utilizing that information to make a systematic approach to finding a factorization of N is the subject of the following three step process.

2.3.1 Relation building

Find integers $x_1, x_2, \ldots, x_r$ so that $z_i \equiv x_i^2 \, (mod \, N)$ is a product of small primes, where z_i is the least residue $\pmod N$. We refer to $z_i \equiv x_i^2 \pmod N$ a relation.

The method of building these relations varies based on the type of algorithm we are using. Though they follow the same general structure, building relations in the number field sieve, for example, looks dramatically different from building relations in the quadratic sieve. For that reason, we will not say much more about relation building at this moment. However, it is the subject of much of the next two sections.

2.3.2 Elimination

Find a product $z_{i_1} z_{i_2} \ldots z_{i_s}$ of some of the z_i's so that each of the powers of primes in the product is even, and so $z_{i_1} z_{i_2} \ldots z_{i_s} = y^2$ for some $y \in \mathbb{N}$.

Suppose that each of the x_i's is B-smooth, and that there are t primes, $A = \{p_1, p_2, \ldots, p_t\}$ less than B. This set A is called the *factor base*. Then, there are exponents e_{ij} such that

$$z_1 \equiv p_1^{e_{11}} p_2^{e_{12}} \dots p_t^{e_{1t}},$$
$$z_2 \equiv p_1^{e_{21}} p_2^{e_{22}} \dots p_t^{e_{2t}},$$
$$\vdots$$
$$z_r \equiv p_1^{e_{r1}} p_2^{e_{r2}} \dots p_t^{e_{rt}}.$$

We aim to find some combination of z_i's so that in their product, each of the p_i's has an even exponent. Alternatively, we aim to find $u_1, u_2, \dots, u_r \in \{0, 1\}$ so that

$$z_1^{u_1} z_2^{u_2} \dots z_r^{u_r}$$

is a perfect square. This product may be written in its prime factorization as follows:

$$\prod_{i=1}^{r} z_i^{u_i} = \prod_{j=1}^{t} p_j^{\sum_{i=1}^{r} e_{ij} u_i}.$$

This is all we need, though it is currently obscured in notation. To make it more clear, recall that we have the exponents,

$$e_{11}, e_{12}, \dots, e_{1t}, e_{21}, e_{22}, \dots, e_{2t}, \dots, e_{r1}, e_{r2}, \dots, e_{rt}.$$

We are looking for appropriate integers $u_1, u_2, \dots, u_r \in \{0, 1\}$ so that

$$e_{11}u_1 + e_{21}u_2 + \ldots + e_{r1}u_r \equiv 0 \,(\mathrm{mod}\, 2)$$
$$e_{12}u_1 + e_{22}u_2 + \ldots + e_{r2}u_r \equiv 0 \,(\mathrm{mod}\, 2)$$
$$\vdots$$
$$e_{1t}u_1 + e_{2t}u_2 + \ldots + e_{rt}u_r \equiv 0 \,(\mathrm{mod}\, 2).$$

Then, it is clear that this system congruences is just a system of linear equations over $\mathbb{F}_2$, and so well-understood linear algebra like Gaussian elimination allows us to solve it.

2.3.3 GCD Computation

Let $x = x_{i_1}x_{i_2}\ldots x_{i_s}$, and compute $d = \gcd(N, x - y)$. Then, as

$$x^2 = (x_{i_1}x_{i_2}\ldots x_{i_s})^2 = x_{i_1}^2 x_{i_2}^2 \ldots x_{i_s}^2 = z_{i_1}z_{i_2}\ldots z_{i_s} \equiv y^2 \,(mod\, N),$$

it is often true that d is a nontrivial factor of N.

GCD computation is both widely known and runs very quickly, so little will be said of this step moving forwards.

Going forwards, both of the following methods of integer factorization rely on the brilliant ideas presented above of factoring by a difference of two squares.

2.4 The Quadratic Sieve

The quadratic sieve is an integer factorization algorithm developed in 1982 by Pomerance [16]. It is based on the three-step process described above. To understand the quadratic sieve, and the number field sieve that follows, it is important to grasp the notion of sieving. The Sieve of Eratosthenes is an ancient Greek process to make lists of prime numbers. We start by making a list of numbers up to an upper bound, where the goal is to find the primes less than that upper bound. Consider the list of numbers $2 - 99$. We then circle the smallest prime, 2, and crossing off each number that is a multiple of 2. The next number that is neither circled nor crossed off is 3, so we circle 3 and cross off multiples of 3. Then, the smallest number neither circled nor crossed off is 5, so we circle 5 and cross out multiples of 5. This process is called sieving by 5. We sieve all primes less than 10 from the list, and in this way find all primes smaller than 100. These primes are simply the numbers that are not crossed out.

Suppose that instead of crossing out numbers when they were a multiple of the prime we are currently sieving by, we divide that number by the prime currently in use instead. Thus, when we sieve 42 by 2, it will become 21. When we sieve by 3, it will become 7, and then when we sieve by 7 it will become 1. Adapting the sieve in this way allows us find B-smooth numbers, where B is a number between the last prime we sieved by, and the next prime.

We can now begin to consider Pomerance's quadratic sieve. Let N be the integer we want to factor, let $F(T) = T^2 - N$, and set $a = \lfloor \sqrt{N} \rfloor + 1$. Then, consider for some $b > a$,

$$C = \{F(a), F(a+1), \ldots, F(b)\}.$$

The goal is to choose a $B \in \mathbb{N}$ large enough so that, after sieving by primes, we have found enough B-smooth numbers to factor N using step 2 from the previous section. Let A be the factor base with respect to B.

To sieve, let $p \in A$. Then, we aim to find those numbers in C that p divides. That is, we find those t with $a < t < b$ satisfying

$$t^2 \equiv N \,(mod\, p).$$

If this has no solutions, then we discard p as it divides none of C. Else, there are two solutions to this congruence for any prime $p \neq 2$. If $p = 2$, there is only one solution. Call these solutions $t = \alpha_p$ and $t = \beta_p$. Then, it follows that each of

$$F(\alpha_p), F(\alpha_p + p), F(\alpha_p + 2p), \ldots$$

and each of

$$F(\beta_p), F(\beta_p + p), F(\beta_p + 2p), \ldots$$

is divisible by p. Thus, we can sieve a factor of p from every pth entry in C, beginning with the smallest x so that $x \equiv \alpha_p \,(mod\, p)$, and similarly we can sieve a factor of p from every pth entry beginning with the smallest x so that $x \equiv \beta_p \,(mod\, p)$.

Once the sieving has been complete for all primes $p \in A$, the result is that some members of C have been sieved all the way down to 1. That is, we have

$$F(k_1), F(k_2), \ldots, F(k_r)$$

sieved to 1, with $a < k_i < b$ for all $i = 1, 2, \ldots, r$ and $r \leq (b - a) + 1$. But then, we have r equations of the form

$$F(k_i) = (k_i)^2 - N.$$

These give rise to r relations of the form

$$(k_i)^2 \equiv p_1^{j_1} \cdot p_2^{j_2} \cdots p_s^{j_s},$$

as we have factored $F(k_i)$ over the factor base A. We can then utilize the steps from the elimination process discussed in the previous section to finish factoring N.

In his paper [16], Pomerance shows that the Quadratic Sieve is $L_N[\frac{1}{2}, r]$ for some r, where r depends on the elimination algorithm used in step two of the three step process. In particular, the Quadratic Sieve is sub-exponential on the size of the input, unlike trial division.

2.5 The Number Field Sieve

The number field sieve is one of the state of the art factorization algorithms. Here, we present in some detail a description of the Number Field Sieve, as given in [12]. We note that the algorithm presented is the special number field sieve, which we can use to factor integers of a particular form. The special number field sieve has been adapted to its general form, which allows for factorization of any integers.

2.5.1 Setup of the sieve

We can factor an integer n, or a small multiple of n, of the form $r^c - s$. Here, r and $|s|$ are small positive integers, $r > 1$, and c large. We choose a small positive integer d, in a manner to be discussed later, and let k be the smallest positive integer so that $kd \geq c$. Then, let $t = sr^{kd-c}$, and define the polynomial f to be $f(x) = x^d - t$. Let $m = r^k$. This gives us

$$\begin{aligned} f(m) &= m^d - t \\ &= (r^k)^d - t \\ &= r^{kd} - sr^{kd-c} \\ &= r^{kd-c}(r^c - s) \\ &\equiv 0 \pmod{n}. \end{aligned}$$

The last line is true because n divides $r^c - s$ by assumption.

Note that in this particular case, we have that f is reducible if and only if there is a prime p dividing d such that t is a pth power, or 4 divides d and $-4t$ is a 4th power, per Theorem 9.1 in chapter VI of [10].

We then define our number field to be $K = \mathbb{Q}(\alpha)$, with $f(\alpha) = 0$. We can check whether or not f is irreducible by the above fact. If it isn't, its non-trivial factor often corresponds to a non-trivial factor of n, in which case we are done, since our goal is to factor n. Else, we can replace f with one of its irreducible factors and continue.

Recall that f being irreducible means that the degree of our number field K is d, the degree of f. Further, we get that elements in K may be described uniquely as follows:

$$\sum_{i=0}^{d-1} q_i \alpha^i, \quad q_i \in \mathbb{Q}.$$

Similarly, elements of the subring $\mathbb{Z}[\alpha]$ of K are described uniquely in a similar fashion, as

$$\sum_{i=0}^{d-1} z_i \alpha^i, z_i \in \mathbb{Z}.$$

We then define the ring homomorphism $\varphi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$ given by

$$\varphi(\alpha) = (m \bmod n).$$

Here, we are using $m \pmod n$ for $m + n\mathbb{Z}$. For $z_i \in \mathbb{Z}$, this can be seen more clearly as

$$\varphi\left(\sum_{i=0}^{d-1} z_i \alpha^i\right) = \left(\sum_{i=0}^{d-1} z_i m^i \bmod n\right).$$

In order to present the algorithm without a long detour, we will assume going forwards that $\mathbb{Z}[\alpha]$ is a principal ideal domain. In particular, this means that $\mathbb{Z}[\alpha]$ is equal to $\mathcal{O}_K$. This is a strong assumption that is certainly not always true. The consequences for this assumption not being made will be discussed later.

2.5.2 A number field interpretation of smoothness

Recall what it means for an integer $z \in \mathbb{Z}$ to be B-smooth. We adapt this definition for our number field. An algebraic integer is B-smooth if every prime integer dividing its norm is at most B.

The norm of $(a + b\alpha)$ is $\mathbf{N}(a + b\alpha) = a^d - t(-b)^d$, so $a + b\alpha$ is B-smooth if

and only if $|a^d - t(-b)^d|$ is a product of primes at most B. To see why this is the norm, recall that the number field K is a finite $\mathbb{Q}$ vector space, with basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{d-1}\}$. Then, multiplication by $(a + b\alpha)$ is a linear transformation, and taking the determinant of the matrix of the transformation, for example via cofactor expansion, yields this norm.

The norm of a nontrivial ideal $\mathfrak{a}$ in $\mathbb{Z}[\alpha]$ is $\mathfrak{N}\mathfrak{a} = \#(\mathbb{Z}[\alpha]/\mathfrak{a})$. We know that a prime ideal of $\mathbb{Z}[\alpha]$ has prime power norm. A *first degree prime ideal* of $\mathbb{Z}[\alpha]$ is a nontrivial ideal $\mathfrak{p}$ of prime norm p. So, any first degree prime ideal is a prime ideal. This can also be seen by noticing that

$$\mathbb{Z}[\alpha]/\mathfrak{p} \cong \mathbb{Z}/p\mathbb{Z},$$

is a field, and so $\mathfrak{p}$ is a prime ideal.

Proposition.

The set of first degree prime ideals $\mathfrak{p}$ is in bijection with the set of pairs $(p, u \mod p)$, where p is a prime number and u satisfies $f(u) \equiv 0 \ (\mod p)$.

This follows from Theorem 27 of [13].

We use a map $\lambda : \mathbb{Z}[\alpha] \to \mathbb{Z}[\alpha]/\mathfrak{p} \cong \mathbb{Z}/p\mathbb{Z}$ that sends α to $u \pmod p$. This map λ can be used to check whether or not an arbitrary element of $\mathbb{Z}[\alpha]$ is contained in $\mathfrak{p}$. That is,

$$\sum_i s_i \alpha^i \in \mathfrak{p}$$

if and only if, for p, u as above,

$$\sum_i s_i u^i \equiv 0 \bmod p.$$

Suppose a and b are relatively prime integers. We have just seen that $a + b\alpha$ is in a prime ideal corresponding to the pair $(p, u \bmod p)$ if and only if $a + bu \equiv 0 \bmod p$. Further, we know that all prime ideals of $\mathbb{Z}[\alpha]$ that contain $a + b\alpha$ are first degree prime ideals [11].

These two facts imply that the prime factorization of $a + b\alpha$ corresponds to the prime factorization of its norm, $\mathbf{N}(a + b\alpha) = a^d - t(-b)^d$, as follows.

Suppose p divides $a^d - t(-b)^d$ exactly k times, for k greater than zero. This means that $a \equiv -bu \bmod p$ for a unique $u \bmod p$, namely $u \equiv \frac{a}{-b} \pmod p$, that satisfies $f(u) \equiv 0 \bmod p$. By the cited fact from [11], we know that any ideal $\mathfrak{p}$ containing $a + b\alpha$ is a first degree prime ideal, with norm p. Then, the first degree prime ideal $\mathfrak{p}$ that is in bijective correspondence with $(p, u \pmod p)$ contains $a + b\alpha$, and more importantly $\mathfrak{p}$ divides $a + b\alpha$ to the kth power.

Thus, a single ideal of norm p accounts for the full exponent of p in $\mathbf{N}(a+b\alpha) = a^d - t(-b)^d$.

We write the generator of $\mathfrak{p}$ in $\mathbb{Z}[\alpha]$ as $\pi_\mathfrak{p}$. This generator is guaranteed to exist, as $\mathbb{Z}[\alpha]$ is a principal ideal domain by assumption, and the generator is thus unique up to multiplication by a unit. Then, we are able to pass from the prime ideal factorization of $a + b\alpha$ to its prime factorization by replacing prime ideal factors $\mathfrak{p}$ by $\pi_\mathfrak{p}$ and multiplying the result by an appropriate unit.

2.5.3 The factor base

We are now at a point where we have established the lead up to the number field sieve. This allows us to move into constructing the factor base. Choose two bounds, B_1 and B_2. B_1 is a smoothness bound for integers $a + bm$, and B_2 a smoothness bound for the algebraic integers $a + b\alpha$. Lenstra et.al [12] note that these B_1 and B_2 are best determined empirically, and give some guidelines and examples of choices.

Let $I = P \cup U \cup G$, where $P = \{p \in \mathbb{Z} : p \,\text{prime}, p < B_1\}$, U is the set of generators of the group of units of $\mathbb{Z}[\alpha]$, and finally G is made up of the set of generators $\pi_{\mathfrak{p}} \in \mathbb{Z}[\alpha]$, where $\mathfrak{p}$ is any member of the set of first degree prime ideals of $\mathbb{Z}[\alpha]$ with $\mathfrak{N}\mathfrak{p} \leq B_2$. Then, the factor base for the number field sieve is formed by elements the $a_i = \varphi(i) \in \mathbb{Z}/n\mathbb{Z}$.

Note that we assume $\gcd(a_i, n) = 1$ for all $i \in I$, as if this is not the case n can be factored trivially, and so we are done.

We postpone the discussion of finding the sets U and G until later.

2.5.4 Building and using the relations

Next, we discuss how to find relations in the number field sieve. That is, we aim to find vectors $v = (v_i)_{i \in I} \in \mathbb{Z}^I$ such that

$$\prod_{i \in I} a_i^{v_i} = 1.$$

To find dependencies mod 2 as discussed in previous sections, we require slightly more relations than the size of the finite set I.

We choose two more bounds, B_3 and B_4, once more chosen empirically, as discussed in [12]. To find relations among the a_i, we search for ordered pairs of integers (a, b) with $b > 0$ so that the following conditions are met:

(*i*) $\gcd(a, b) = 1$,

(*ii*) $|a + bm|$ is B_1 - smooth, with the exception of at most one additional prime factor p_1 satisfying $B_1 < p_1 < B_3$,

(*iii*) $a + b\alpha$ is B_2 - smooth, with the exception of at most one additional prime ideal factor $\mathfrak{p}_2$ satisfying $B_2 < p_2 < B_4$, where $p_2 = \mathfrak{N}\mathfrak{p}_2$.

We assume $a + bm > 0$. If in fact $a + bm < 0$, replace the ordered pair (a, b) with its negative.

For each pair (a, b), we call the prime p_1 the large prime, and the prime ideal $\mathfrak{p}_2$ the large prime ideal. Recall from the bijective correspondence earlier that $\mathfrak{p}_2$ corresponds to a unique ordered pair, $(p_2, u \bmod p_2)$, where u satisfies $a \equiv -bu \bmod p_2$. This allows us to differentiate between prime ideals of the same norm. If either the exceptional large prime or large prime ideal do not occur, we simply write $p_1 = 1$, or $\mathfrak{p}_2 = 1$ and $p_2 = 1$. When $p_1 = p_2 = 1$, we call the relation a full relation, as opposed to the other partial relations.

In order to find ordered pairs (a, b) satisfying the three conditions above, we make use of two sieves, the rational sieve and the algebraic sieve. The discussion of these sieves can be found in a later subsection. We now discuss how the pairs result in relations among the a_i's. Suppose (a, b) is a full relation. Then, we have

$$a + bm = \prod_{p \in P} p^{e(p)},$$

with $e(p) \in \mathbb{Z}$ and $e(p) \geq 0$, since P as before is the set of primes less than B_1, and since (a, b) is full we know that there is no exceptional prime factor that is greater than B_1.

By the above conditions and our discussion of smoothness in the number field, we can write $a + b\alpha$ as a product of elements $\pi_{\mathfrak{p}} \in G$ to certain powers and a unit of $\mathbb{Z}[\alpha]$. Since U generates the units of $\mathbb{Z}[\alpha]$, the units in the prime factorization of $a + b\alpha$ may be written as a product of elements from U, as discussed in [12]. Thus, we get

$$a + b\alpha = \prod_{z \in U} z^{e(z)} \cdot \prod_{g \in G} g^{e(g)},$$

with $e(z) \in \mathbb{Z}$ and $e(g) \in \mathbb{Z}$ and $e(g) \geq 0$. However, we know that $a + bm$ and $a + b\alpha$ have the same image under our ring homomorphism, $\varphi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$, and so we get

$$\prod_{p \in P} \varphi(p)^{e(p} = \prod_{z \in U} \varphi(z)^{e(z)} \cdot \prod_{g \in G} \varphi(g)^{e(g)}.$$

This gives a relation $v = (v_i)_{i \in I} \in \mathbb{Z}^I$ between the a_i, obtained by setting $v_i = e(i)$ for $i \in P$ and $v_i = -e(i)$ for $i \notin P$. Then, $a_i^{v_i}$ exists for $i \notin P$ as well, since $\gcd(a_i, n) = 1$ for $i \in I$.

Note that partial and also free relations can be used to build more relations between the a_i's, however this discussion is left to [12].

2.5.5 The sets U and G

We now revisit the sets U and G introduced earlier in this section. Recall that an element belongs to the set of units of $\mathbb{Z}[\alpha]$ if and only if it has norm ± 1. The polynomial f is $f(x) = x^d - t$. We say that f has r_1 real roots and $2r_2$ non-real complex roots, and so $r_2 = \dfrac{d - r_1}{2}$. Then, if d is odd, we get that $r_1 = 1$, and if d is even, we have $r_1 = 0$ if $t < 0$, else $r_1 = 2$ when $t > 0$. Set $l = r_1 + r_2 - 1$. Then, units of $\mathbb{Z}[\alpha]$ are generated by an appropriate root of unity u_0, in addition to l independent units of infinite order. We then get that $U = \{u_0, u_1, \ldots, u_l\}$.

Due to the bijective correspondence discussed previously, a list of all first degree prime ideals with norm $\leq B_2$ is equivalent to a list of pairs $(p, u \bmod p)$, where $p < B_2$ is prime and $u \in \mathbb{Z}$ satisfies $f(u) \equiv 0 \bmod p$. By [9], a probabilistic root finder for polynomials over finite fields can be applied to find these pairs. The number of pairs found, which is the cardinality of G, is expected to be equal to the number of primes less than B_2. Then, an element of $\mathbb{Z}[\alpha]$ generates $\mathfrak{p}$ if and only if it is in fact a member of $\mathfrak{p}$, and if its norm is $\pm p$. Then, finding G amounts to finding one such generator for each pair $(p, u \bmod p)$.

2.5.6 The sieve without assumption of PID

In our presentation of the number field sieve, we forced the assumption that $\mathbb{Z}[\alpha]$ is a principal ideal domain, which implies that $\mathbb{Z}[\alpha]$ is the ring of integers of K, which means that $u \equiv 0$ or $-1 \bmod d$.

Fortunately, we can adapt without the above assumption. In this section, we

will summarize the methods for resolving the issues. However, repairing the sieve without the assumption of a PID is very difficult, and so we will leave most of the details to section 3.4-3.8 of [12]. The first step is that instead of working in $\mathbb{Z}[\alpha]$, we replace it with the ring A of algebraic integers of K. The number field sieve requires a few modifications to work with A instead of $\mathbb{Z}[\alpha]$. Most obviously, we must extend the ring homomorphism $\varphi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$ to A. If we have that $\gcd(drs, n) = 1$, then any $\gamma \in A$ has the form $\gamma = \beta/\delta$, with $\delta \in \mathbb{Z}$ a product of primes that divide drs. Then, $\varphi(\delta)$ has an inverse in $\mathbb{Z}/n\mathbb{Z}$, and the extension of φ is just $\bar{\varphi} : A \to \mathbb{Z}/n\mathbb{Z}$ given by $\bar{\varphi}(\gamma) = \varphi(\beta)\varphi(\delta)^{-1}$.

Further, a prime ideal $\mathfrak{p} \subseteq A$ that divides an $a + b\alpha$ for a, b relatively prime integers is not necessarily a first degree prime ideal. That is, it may not be the case that $\#A/\mathfrak{p}$ is prime. Instead of just first degree prime ideals, there are also a new type of ideals which we henceforth refer to as exception ideals. *Exceptional ideals* are prime ideals $\mathfrak{p}$ of A that have a prime number p dividing $[A : \mathbb{Z}[\alpha]]$. Also, these ideals' intersection with $\mathbb{Z}[\alpha]$ is a first degree prime ideal of $\mathbb{Z}[\alpha]$, and the p divides drs. Thus, to construct prime ideal factorizations of the $a + b\alpha$'s, the exceptional prime ideals $\mathfrak{p}$ must be constructed as well.

The remaining work to repair the sieve when $\mathbb{Z}[\alpha]$ is not a PID is left for [12]. It involves adapting the sets U and G from the previous discussion to work in the new environment.

2.5.7 Sieving for pairs (a, b)

Recall that from the section on building relations, we are searching for pairs $a, b \in \mathbb{Z}$ where a, b are coprime so that $a + bm$ is B_1-smooth, except for at most one prime factor p_1 less than B_3, and so that $a + b\alpha$ is B_2-smooth, except for

at most one prime ideal $\mathfrak{p}_2$ with norm less than B_4. We lay out a way to find pairs (a, b) as above for a fixed b, and a in an interval $[a_{\min}, a_{\max})$. This method is then applied to all b in $[1, b_{\max}]$. Note that the values of $a_{\min}$ and $a_{\max}$ are best determined empirically, discussed more in Sections 6 and 8 of [12]. On the other hand, a $b_{\max}$ need not be chosen, as more pairs can be checked until the number of relations is larger than the size of the factor base I.

Fix a $b \geq 1$. To find numbers $a+bm$ for $a \in [a_{\min}, a_{\max})$ that are B_1-smooth, we can use a sieve over a, as in the quadratic sieve, as p divides $a + bm$ whenever $a \equiv -bm \pmod{p}$. After sieving for $p \leq B_1$, pairs (a, b) with a chance at satisfying the second condition from the relations building section are identified, and checked to see if the accompanying $a+b\alpha$'s are B_2-smooth. This is done usually with a second sieve over the interval $[a_{\min}, a_{\max})$, though if there are only a few possibilities of (a, b)'s, one may use trial division instead. This second sieve uses the fact that the first degree prime ideal that corresponds to a pair $(p, u \pmod{p})$ contains $a + b\alpha$ for all $a \equiv -bu \pmod{p}$.

Once pairs (a, b) have been found for which $a + bm$ and $a + b\alpha$ are good candidates to satisfy their respective smoothness criteria, those particular pairs are tested to see if they result in full or partial relations.

2.5.8 Run time analysis

A noted problem in the analysis of this particular factorization algorithm, as well as a variety of others, lies in a problematic assumption. In many factorization algorithms, a sequence of numbers is found of which the only ones of interest are the B-smooth ones, for some B. The expected number of B-smooth numbers in the sequence has a sizable impact on the run time analysis. In the case that

the integers are drawn independently in uniform fashion from the interval $[1, B]$, an accurate estimate may be reached. In contrast, a heuristic analysis assumes that the expected number of B-smooth numbers in the sequence is the same as in the aforementioned special case. The number field sieve (and general number field sieve) are not algorithms in which this special case has been proven to occur, so we do not know if there is a uniform distribution of smooth numbers in the interval $[1, B]$.

The heuristic analysis that follows relies on $N = r^c - s$, rather than n, a factor of N. The L function discussed previously is used here. Then, let $C \subseteq \mathbb{R}^4$ be a compact set such that for all $(\lambda, \mu, w, v) \in C$, we have $\lambda > 0, \mu > 0$, and $0 < w < v \leq 1$. Then, the chance that a positive integer $m \leq L_x[v, \lambda]$ is $L_x[w, \mu]$-smooth is $L_x[v - w, -\lambda(v - w/\mu + o(1)]$ for $x \to \infty$, uniformly for $(\lambda, \mu, w, v) \in C$ [12].

Using this probability, the authors found in [3] that the optimal values for $a_{\max}, b_{\max}, B_1$, and B_2 are equal to

$$\exp((\frac{1}{2} + o(1))(d \log d + \sqrt{(d \log d)^2 + 2 \log(N^{1/d}) \log\log(N^{1/d)}})),$$

with $o(1)$ for $c \to \infty$, with bounded r and s, and d from $1 < d^{2d^2} < N$. Further, take $B_3 = B_1$ and $B_4 = B_2$. In this way, only full relations are considered. Then, the size of the factor base and the expected number of full relations are given by

$$\exp((\frac{1}{2} + o(1))(d^2 \log d + 2 \log(N^{1/d} + d\sqrt{(d \log d)^2 + 2 \log(N^{1/d}) \log\log(N^{1/d})})).$$

Further, the run time for the sieving in the step of collecting relations, and the

solution of the linear system while finding dependencies is

$$\exp(1+o(1))(d\log d+\sqrt{(d\log d)^2+2\log(N^{1/d})\log\log(N^{1/d})}).$$

The other parts of the algorithm are negligible in comparison, with the potential exception of finding the sets U and G.

Finally, the optimal choice for the degree of the function d as a function of N is

$$d=(\frac{(3+o(1))\log N}{2\log\log N})^{1/3},$$

for $c\to\infty$. Using this choice of d, the values of $a_{\max}, b_{\max}, B_1, B_2, B_3$, and B_4 are $L_N[\frac{1}{3},(2/3)^{2/3}]$. The typical size of $|a+bm|$ and $|\mathbf{N}(a+b\alpha)|$ is $L_N[\frac{2}{3},(2/3)^(1/3)]$, and so the numbers one wants to be smooth are roughly $L_N[\frac{2}{3},(16/3)^{1/3}]$. This gives us an expected run time of the whole special number field sieve algorithm, with possible exception of the search while selecting a factor base from the number field, is

$$L_N[\frac{1}{3},(32/9)^{1/3}].$$

All told, this means that the number field sieve is the first factorization algorithm with run time better than $L_n[\frac{1}{2},\lambda]$ for some $\lambda>0$.

2.6 Choosing an algorithm

Recall that Pollard's algorithm requires an integer L so that, for $N=pq$, $p-1$ divides L and $q-1$ does not divide L. The key insight that Pollard made was that if $p-1$ is the product of small primes, we can use that to our advantage and

factor N. Thus, the $p-1$ algorithm is the best factorization when $N = pq$, and $p-1$ is B-smooth, for some suitably small B.

Often, this is not the case. This is especially true since people are aware of the $p-1$ factorization algorithm, and so when choosing prime factors p, q in RSA, for example, they will avoid that situation entirely. Our discussion then shifts to the question of when we should prefer the quadratic sieve, versus when the number field sieve will serve us better, and in particular the general number field sieve.

There is a general consensus, as Pomerance describes in his enjoyable article, "A tale of Two Sieves" [17], that the general number field sieve outperforms the quadratic sieve for numbers 130 digits and longer, while the quadratic sieve is superior for numbers 100 digits or less. However, getting anything more precise than that is difficult. A large part of that is that the efficacy of the algorithms varies based on implementation, and in the particular case of the general number field sieve, the amount of memory the computers running it have available.

What is precise are factorization records for each algorithm. Though not the most useful measure, they do help illustrate the ability of the number field sieve to deal with large numbers in an unprecedented fashion. For example, the largest RSA number first factored with the quadratic sieve is known as RSA-129. This is a 129 digit number that used 600 volunteers and around 1600 computers, as described in [1]. On the other hand, the record for the general number field sieve is known as RSA-768, which is a number with two-hundred and thirty two digits. This factorization took over two years to complete, finishing in 2009, and nearly two-thousand years of computing time [7].

Chapter 3

Collision Algorithms

3.1 Introduction

The well documented birthday problem serves as an introduction to the area of Collision Algorithms in cryptography. The birthday problem is stated as follows: In a random grouping of 40 people, what is the probability that two of them share a birthday? Let X be the event that two people share a birthday. Then,

$$\begin{aligned} Pr(X) &= 1 - Pr(\neg X) \\ &= 1 - \prod_{i=1}^{40} \frac{365 - (i-1)}{365} \\ &= 1 - (\frac{365}{365})(\frac{364}{365})(\frac{363}{365}) \ldots (\frac{326}{365}) \\ &\approx .891. \end{aligned}$$

Contrast this with the following scenario. Suppose John's birthday is March 5th. The probability that one of the other 39 people shares a birthday with John is only $\approx 10.4\%$. The relevant implication is that it is easier for any two objects

to be the same than it is for any object to match another particular one.

Collision algorithms, whose goal is, like in the birthday problem, to find any two objects that are equal, have great relevance in cryptography. In the introduction, we discussed the Diffie-Hellman key exchange, and mentioned that for Eve to be able to break the encryption, she must be able to solve either the Diffie Hellman problem, or a particular case of the discrete logarithm problem itself. Some of the collision algorithms discussed below, including the first one, may be used towards that exact goal.

One such relevant form of the DLP is known as the interval DLP. Suppose a and b are integers with $0 \leq a < b$, and $x \in [a, b]$. Then, the interval DLP on $[a, b]$ is, given elements $g, h \in G$, to find x so that $g^x = h$. Note that if $a = 1$ and b is the order of g, this is just the classic DLP. In cryptogaphy, it is often the case that $a = 0$ and b is much smaller than the order of g.

In this chapter, we will first discuss cycle detection algorithms. Often, as is the case in Pollard's ρ method, discussed later in section 3.3 of this document, cycle detection algorithms are a pivotal step in actually detecting the titular collision.

3.2 Cycle detection algorithms

Let $F : X \to X$ be an arbitrary function on a finite set X. Choose some x_0 from X randomly, with respect to the uniform distribution. Then, as X is finite, the sequence $(x_k)_{k\in\mathbb{N}}$ given by $x_{k+1} = F(x_k)$ is cyclic. That is, there exist integers $t \geq 1$ and $c \geq 0$ such that $x_0, \ldots, x_{t+c-1}$ are pairwise distinct, with $x_i = x_{i+c}$ for $i \geq t$. The question of cycle detection is then the question of finding when and where such a match occurs. Here, t is the length of the tail, which means that t

is the greatest integer such that x_{t-1} appears only once in the sequence (x_k), and c is the length of the cycle itself.

3.2.1 Floyd and Brent's algorithms

Two common algorithms for cycle detection are given by Floyd and Brent. Floyd's algorithm is based on taking two variables, y and z so that one advances twice as fast as the other. That is, $y_i = f^i(x)$ and $z_i = f^{2i}(x)$. This will eventually result in $y_j = f^j(x) = f^{2j}(x) = z_j$, since we know there is a cycle, where j is the smallest positive multiple of c that is at least t.

Brent's algorithm proceeds similarly to Floyd's [2]. The major difference is that at powers of 2, instead of letting the slower sequence lag further and further behind, we set $y = z$, and then advance them as in Floyd's algorithm. This means that at the start, the faster sequence only advances two steps before the slower sequence is set equal to it, with the next time four steps, and so on.

Brent's algorithm has three advantages over Floyd's. One is that it recovers the value of c directly, while Floyd's requires you to find c in a later stage, and further it requires just one evaluation of the function rather than three. Finally, Brent himself showed in his paper that his algorithm is always better than Floyd's, both on average and in worst case performance. In particular, Floyd's algorithm is expected to find a match after $\cong 3.0924$ function evaluations, whereas Brent's algorithm is is expected to take only $\cong 1.9828$ function evaluations.

3.2.2 Nivasch's algorithm

Nivasch gives another cycle detection algorithm, making use of a stack. Note that a stack is an abstract data type, which is a collection of elements together with two major operations. The first operation is adding an element to the stack, and the second is removing the most recently added element still present in the stack. Further, Nivasch's algorithm requires that our function $f : X \to X$ operates on a set X with a total ordering. In our context, Nivasch's algorithm is applied to finite sets of integers, so this holds.

The algorithm proceeds as follows. Record a stack of pairs $(f^i(x), i)$ where both the $f^i(x)$'s and i's form strictly increasing sequences at all times. This stack is initially empty, and for each step j, we remove from the stack all entries $(f^i(x), i)$ where $f^i(x) > f^j(x)$. If $f^i(x) = f^j(x)$ is found, we are done, and we have recovered the cycle length, in that $c = j - i$. Otherwise, move $(f^j(x), j)$ to the top of the stack, and perform the next step.

We claim that Nivasch's algorithm always halts on the smallest value of the sequence's cycle that repeats. To see this, let $f^{i'}(x)$ be the minimal value of the cycle. When it is added to the stack on the first loop through the cycle, it is never removed. Thus, the algorithm will halt when it encounters $f^{i'}(x)$ on the second loop through the cycle. Else, for any other value $f^j(x)$, is necessarily greater than $f^{i'}(x)$, and so it will be removed from the stack by $f^{i'}(x)$ before it can appear again.

In order to perform a run time analysis on this algorithm, we assume that the sequence of $f^i(x)$'s are independently random values. Note that this is constrained by the necessarily period nature of an iterating function over a finite set. Then, since the minimal value $f^{i'}(x)$ appears at a random spot in the cycle, for a fixed

tail length t and cycle length c, the average run time is $t + \frac{3}{2}c$.

Nivasch presents another similar stack algorithm, which makes use of multiple stacks. We will distinguish these algorithms by calling the previous one the single stack algorithm, as opposed to the following multi-stack algorithm. The goal of the multi-stack algorithm is to decrease the halting time of the single stack algorithm, with negligible increases to memory and no change to running time on per-step basis.

Let $k \in \mathbb{Z}$, and partition the finite, totally ordered set X into k disjoint classes. One possibility is to consider the remainder modulo k of each value $f^i(x)$. For each class, the algorithm uses a separate stack. At each step j, determine which class $f^j(x)$ belongs to, and then perform the step from the single stack algorithm on the stack corresponding to $f^j(x)$'s particular stack. This algorithm always works, because a given value is always sent to the same stack.

To examine the run time of the multi-stack algorithm, for each class j with $0 \leq j < k$, j contains its own cycle minimum, $f^{i'_j}(x)$. The algorithm finishes when it hits the first over all the minima for the second time. Since we are using the same assumptions as before, the $f^{i'_j}(x)$'s are distributed uniformly and independently at random in the cycle. Thus, the run time of the multi-stack algorithm has gone down from the single stack version to just

$$t + c(1 + \frac{1}{k+1}).$$

Since any algorithm must evaluate f at least $t + c$ times, this algorithm is very powerful.

3.2.3 Sedgewick, Szymanski, and Yao's algorithm

Another algorithm, due to Sedgewick, Szymanski, and Yao, detects the cycle and recovers the values of both t and c, with the caveat that it is memory intensive. The goal of Sedgewick et al. was to develop an algorithm that had the best worst-case run time among cycle detection algorithms. In fact, they showed that their worst case performance is asymptotically optimal. This algorithm is presented in [19], though we will summarize it here.

Let M be a free parameter, and T a table of size M. Set d to be 1 initially, and at each step i, check whether $i \pmod{gd} < d$, where g is also a free parameter. If $i \pmod{gd} < d$, then we search for $f^i(x)$ in T. If i is a multiple of d, store $(f^i(x), i)$ in T. If at any step the table becomes too full, d is doubled and all entries $(f^j(x), j)$ where j is no longer a multiple of d are removed from T. Since we are attempting to optimize the worst case scenario, T is implemented using something to reduce worst case search time, such as a balanced tree. Then, if t_s is the time needed to perform a search of T and t_f is the time needed to evaluate f once, Sedgewick et. al found that g can be chosen so that the worst case run time of their algorithm is

$$t_f(t+c)(1+\Theta(\sqrt{t_s/Mt_f})).$$

3.3 Pollard's ρ Method

The ρ method to detect collisions was presented by J. Pollard [15], to compute discrete logarithms in $(\mathbb{Z}/p\mathbb{Z})^*$. Let X be a finite set. Recall that, for $g, h \in (\mathbb{Z}/p\mathbb{Z})^*$, we are trying to find an x so that $g^x = h$.

Pollard presented the method using a particular iterating function, $F : (\mathbb{Z}/p\mathbb{Z})^* \rightarrow$

$(\mathbb{Z}/p\mathbb{Z})^*$.

$$F(y) \equiv \begin{cases} qy & 0 < y < \frac{1}{3}p \\ y^2 & \frac{1}{3}p < y < \frac{2}{3}p \\ ry & \frac{2}{3}p < y < p \end{cases}$$

where r is a primitive root of a prime p, $q \in \mathbb{Z}$, and y is in $0 < y < p$, and the above function is viewed mod p.

Then, we define (y_n) by $y_0 = 1$ and $y_{n+1} \equiv F(y_n) \pmod{p}$. We also define sequences (α_n) and (β_n) so that $y_n = g^{\alpha_n} \cdot h^{\beta_n}$ for $k \in \mathbb{N} \cup \{0\}$. We define $\alpha_0 = 0, \alpha_{n+1} \equiv \alpha_n + 1, 2\alpha_n$, or $\alpha_n \bmod (p-1)$, and $\beta_0 = 0, \beta_{n+1} \equiv \beta_k, 2\beta_k$, or $\beta_k + 1 \bmod (p-1)$, split up according to the same rules as the function F. Then, we have a solution to the DLP when there are terms y_k and y_l with $y_k = y_l$ and $k \neq l$, which gives us $\alpha_k + x\beta_k \cong \alpha_l + x\beta_l \bmod (p-1)$. This allows us to find x using the extended Euclidean algorithm, provided $\gcd(p-1, \beta_k - \beta_l) = 1$.

In order to find the solution to the DLP using the above information, one can apply either Floyd or Brent's cycle detection algorithms, as we saw in Section 3.1 of this document. To find the k for which $y_k = y_l$ using Brent's algorithm, finding the match which solves the DLP is expected to take $1.97\sqrt{n}$ iterations of the function F, where n is the order of g in $\mathbb{Z}/p\mathbb{Z}$. [21]. This calculation is done under the assumption that F is a random mapping, though in fact Pollard's iterating function appears to not behave quite randomly.

Further work has been done on optimizing Pollard's ρ method, including the development of more desirable iterating functions that more closely mimic the performance we would expect from one of the $|X|^{|X|}$ random mappings [20].

3.4 Pollard's Lambda Method

Another collision algorithm presented by J. Pollard in [15] is known as either the Lambda method or the Kangaroo method. We present it here as a method to solve the $[a, b]$ interval DLP. Before we begin, we note that the Lambda method is really about two sequences of group elements. However, Pollard found that discussing it using the metaphor of tame and wild kangaroos aided in the conceptualization of the algorithm. Since we agree, we discuss it here in its full marsupial glory.

Let T be a tame kangaroo that starts at $t_0 = g^b$, and W a wild kangaroo with starting point $w_0 = h$. Recall that in the interval DLP, we are given elements $g, h \in G$ and have integers a, b and x such that $0 \leq a < b$ and $x \in [a, b]$. Our goal to to find the value of x so that $g^x = h$.

Let $N = \{g^{n_1}, \ldots, g^{n_r}\}$ with $n_i > 0$ of size $O(\sqrt{b-a})$ and $r = O(\log(b-a))$ be a set of jumps. The exponent n_i's are thought of as travel distances. Finally, let $v : G \to \{1, \ldots, r\}$ be a hash function.

The tame kangaroo follows the following recursive path as it travels the group.

$$t_{k+1} = t_k \star g^{n_{v(t_k)}}, \qquad k \in \mathbb{N}.$$

While tracing the path followed by our tame kangaroo T, we record the path it has traveled by its distance, $D_k(T)$. Let $D_0(T) = 0$, and

$$D_{k+1}(T) = D_k(T) + n_{v(t_j)}, \qquad k \in \mathbb{N}.$$

Thus, we have that $t_j = x^{j+D_j(T)}$ for every $j \in \mathbb{N}$. After some number of jumps,

the tame kangaroo stops and sets a trap for the wild kangaroo at spot t_Z. The wild kangaroo W is then set loose, following the path

$$w_{j+1} = w_j \star g^{n_v(w_j)}, \qquad j \in \mathbb{N}.$$

Again, we have that distances $D_j(W)$ are given by $D_0(T) = 0$, and

$$D_{j+1}(W) = D_j(W) + s_{n(w_j)}, \qquad j \in \mathbb{N}.$$

Thus, we have $w_j = y \star g^{D_J(W)} (j \in \mathbb{N})$.

Note that at each jump, we know the exact position of the tame kangaroo T. In contrast, we do not know the position of W due to its unknown starting point. After each jump of W, it is checked whether or not W has fallen into the trap laid at t_Z. If this has occurred, say after M jumps, then we have $t_Z = w_M$. This gives us a solution to the discrete logarithm problem $g^x = h$. That is, $x = D_Z(T) - D_M(W)$.

Clearly however, it is possible that after a certain number of jumps, W has not fallen into the trap laid by T. If that is the case, W is halted and a new wild kangaroo is released with starting point $w_0 = h \star g^c$ and initial distance $D_0(W) = c$ for c small.

Now, if W falls into T's trap, then it must be the case that the two kangaroos had met earlier in the group, and had identical paths ever since. One can imagine the lambda referred to in the name of the method as the figure traced out by the

paths of the two kangaroos, if W starts at the bottom left and T the bottom right, noting that once W's path intersects T's, they must travel in lock step from then on.

Pollard found in [15] that his kangaroo method has minimal run time when the exponent n_i's have mean value $\sqrt{b-1}/2$, and the tame kangaroo jumps $0.7(\sqrt{b-a})$ times before dropping the trap. When this occurs, the kangaroo method is expected to take about $3.33\sqrt{b-a}$ group operations to complete. We note that a variety of improvements have been made to the kangaroo method, including ones that use more than two kangaroos.

3.4.1 Choosing a cycle detection algorithm

To discuss choosing an algorithm, we restrict our attention to only the cycle detection algorithms. The reason for this is that, as presented, Pollard's two algorithms are solving slightly different problems, in that the lambda method is solving the $[i, j]$ interval DLP, as opposed to the ρ method solving the DLP in $\mathbb{Z}/p\mathbb{Z}$. Thus, the algorithms we aim to compare now are Floyd and Brent, Nivasch, and Sedgewick, Szymanski, and Yao. Though run time is the primary concern, we will make some mention of the trade off of run time vs. memory.

First, Nivasch finds that his single stack algorithm is roughly 20% faster than Brent's algorithm on average. Previously, we mentioned that Brent's algorithm is always faster than Floyd's algorithm, so on average performance the single stack algorithm beats both necessarily. However, Floyd and Brent's algorithms require much less memory, as they only need keep track of two sequence values as opposed to an entire stack of values of the function f.

Sedgewick, Szymanski, and Yao set out to find an algorithm that had the best worst-case performance, while using bounded memory. Nivasch finds that the multi-stack algorithm's average performance, as a function of memory used, is asymptotically better than the worst case performance of Sedgewick, et al.'s. However, the multi-stack algorithm's worst case performance is far worse.

Finally, a major factor influencing the best choice of algorithm in different situations is how our function f behaves. Nivasch finds that if f is close to random, the multi-stack algorithm is a strong choice, although there are algorithms not discussed in this document that compete with it while using less memory. On the other hand, if f is known to not behave randomly, such as f often yields short cycles, then the multi-stack algorithm will perform the best. Conversely, if the goal is to minimize the worst case performance, such as when we know very little about the function at all, the optimal choice is Sedgewick et al.'s algorithm.

Bibliography

[1] D. Atkins, M. Graff, A.K. Lenstra, and P. C. Leyland. The magic words are squeamish ossifrage. In *Proceedings of Asiacrypt '94*, pages 261–277. Springer-verlag, 1994.

[2] Richard P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20:176–184, 1980.

[3] J.P. Buhler, H.W. Lenstra Jr., and C. Pomerance. *Factoring integers with the number field sieve*, pages 50–94. Springer-verlag, 1991. Lecture notes in mathematics.

[4] E.R. Canfield, P. Erdős, and C. Pomerance. On a problem of oppenheim concerning "factorisatio numerorum". *J. Number Theory*, 17:1–28, 1983.

[5] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[6] Jeffrey Hoffstein, Jill Pipher, and J.H. Silverman. *Undergraduate Texts in Mathematics: An Introduction to Mathematical Cryptography.* Springer, 2008.

[7] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery,

Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010. http://eprint.iacr.org/2010/006.

[8] Donald E. Knuth. *The art of computer programming: fundamental algorithms*, volume 1. Addison-wesley longman publishing co, inc., 3 edition, 1997.

[9] Donald E. Knuth. *The art of computer programming: seminumerical algorithms*, volume 2. Addison-wesley longman publishing co, inc., 3 edition, 1997.

[10] S. Lang. *Algebra*. Addison-Wesley, third edition, 1993.

[11] A.K. Lenstra, H.W. Jr. Lenstra, M.S. Manasse, and J.M. Pollard. The factorization of the ninthe fermat number. *Mathematics of Computation*, 61:319–349, 1993.

[12] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, and J.M. Pollard. *The number field sieve*, pages 11–42. Springer-verlag, 1991. Lecture notes in mathematics.

[13] Daniel A. Marcus. *Number Fields*. Springer-verlag, 1977.

[14] J.M. Pollard. Theorems on factorization and primality testing. *Mathematical Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[15] J.M. Pollard. Monte carlo methods for index computation (mod p). *Mathematics of Computation*, 32 (143):918–924, 1978.

[16] Carl Pomerance. Analysis and comparison of some integer factoring algorithms. In *Computational Methods in Number Theory: Part 1, Lenstra, H.W. and Tijdeman, R.*, pages 89–139, 1982.

[17] Carl Pomerance. A tale of two sieves. *Notices Amer. Math. Soc*, 43:1473–1485, 1996.

[18] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[19] Robert Sedgewick, Thomas G. Szymanski, and Andrew C. Yao. The complexityh of finding cycles in periodic functions. *Society for Industrial and Applied Mathematics Journal on Computing*, 11(2):376–390, 1982.

[20] Edlyn Teske. Speeding up pollard's rho method for computing discrete logarithms. In *Algorithmic Number Theory*, pages 541–554. Springer, 1998.

[21] Edlyn Teske. Square-root algorithms for the discrete logarithm problem (a survey). In *In Public Key Cryptography and Computational Number Theory, Walter de Gruyter*, pages 283–301, 2001.