The Index Calculus Algorithm for Discrete Logarithms

Jason S. Howell

March 31, 1998

A paper submitted to the
Department of Mathematical Sciences
of Clemson University
in partial fulfillment of the
requirements for the Masters Degree

Approved by: _____
Committee Chairman

Abstract

The intractability of the discrete logarithm problem provides a basis for the security of many public-key cryptosystems. We provide a survey of various algorithms to attack the discrete logarithm problem in finite fields. In particular, we study the effectiveness of the index calculus methods for finding discrete logarithms and different techniques that improve their performance. We discuss the use of a sieve method for selecting smooth polynomials from a large set. Gordon and McCurley (1992) developed a sieve for Coppersmith polynomials using a Gray code. Our new contribution is a generalization of their approach to any subspace of polynomials, along with implementation techniques that improve the efficiency of the sieve. We also present applications of this sieve to Coppersmith and Semaev polynomials.

Contents

List of Tables

1 Cryptography and the Discrete Logarithm Problem

For centuries it was thought that the purest discipline of mathematics, number theory, would not be of much use for any practical purposes aside from recreation and sport for mathematicians. The following passage, by a mathematician, reflects this sentiment.

> ...both Gauss and lesser mathematicians may be justified in rejoicing that there is one science [number theory] at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean.
>
> –G. H. Hardy, *A Mathematician's Apology*, 1940

A mere fifty-seven years have passed since Hardy's remark, and things have changed dramatically. Number theory has become a focal point for research in cryptography and computation. As we move into an era dominated by computers, the internet, and electronic communication, cryptography (and thus number theory) becomes an even more important part of our society.

1.1 Introduction

Cryptography, the art and science of secure information transfer, is rooted deeply in mathematics. From the earliest shift (permutation) ciphers to the current cutting-edge elliptic curve cryptosystems, we can find discrete mathematics and probability in each type of system. The main purpose of cryptography (from Greek $\kappa\rho\upsilon\pi\tau\acute{o}\sigma$, meaning hidden, and $\gamma\rho\alpha\phi\acute{\iota}\alpha$, meaning writing) is to enable two individuals to communicate privately across an insecure channel. References to cryptography have been found in the Bible, stories of ancient Greece, and through countless wars and battles fought all over the globe in man's history [54]. Present applications of crytpography extend far beyond national security, as the use of computers and electronic information transfer become more and more a part of everyday life.

Figure 1 is (a portion of) the official internet homepage for RSA Data Security, Inc, and the official logo of RSA can be seen here. At present, this logo can also be found on many other world wide web pages, as many individuals and businesses employ RSA to secure their communications, from private e-mail to transferring account numbers over the internet. The RSA cryptosystem is one of the most widely used cryptosystems today in practice.

The underlying theme in all cryptographic research is essentially to first design and create seemingly secure cryptosystems, and then to use every tool possible to attack them. This may seem undesirable, for why would you want to dismantle something you have just created? The answer is simple: you can only assume that an opponent knows just as much about the cryptosystem being used as you do and has the same abilities. This is often referred to as *Kerckhoff's principle* [55]. Thus we begin our study of the mathematical aspects of cryptography and cryptosystems.

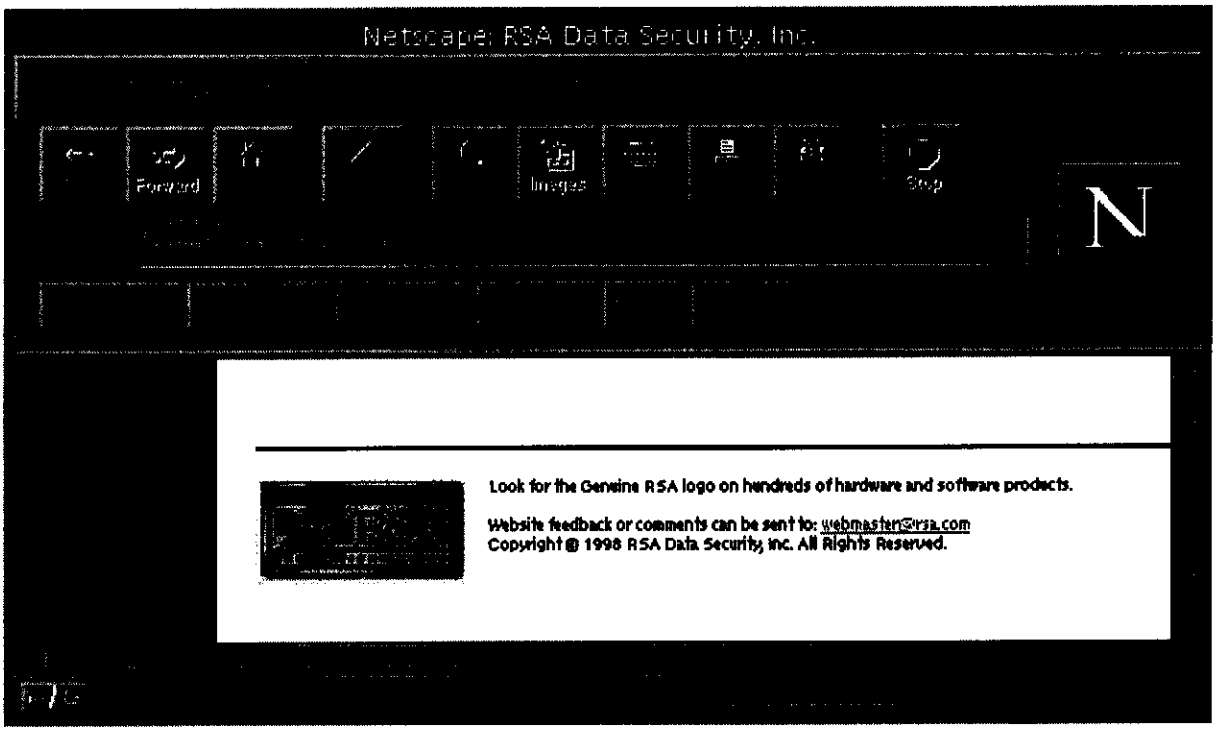

Figure 1: RSA Data Security, Inc. logo

1.2 Cryptosystems

There are an abundance of different types of crytposystems that have been created and implemented in the past and present, each one specific to the needs of those who use it. In the United States, for example, the National Bureau of Standards regulates the most widely used cryptosystem in the world, known as the Data Encryption Standard (DES). This is the current standard for electronic funds transfer used by financial institutions. Another widely used cryptosytem is the aforementioned RSA cryptosystem, which is employed by the internet browser manufacturer Netscape to promote secure commerce across the world wide web. We now give a precise definition of what we mean by cryptosystem.

Definition: [55] A *cryptosystem* is a 5-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ where the following are satisfied:
1. $\mathcal{P}$ is a finite set of possible plaintexts (letters, words, messages).
2. $\mathcal{C}$ is a finite set of possible ciphertexts (the encrypted message).
3. $\mathcal{K}$ is a finite set of possible keys (called the keyspace).
4. $\forall k \in \mathcal{K}$, $\exists$ an encryption rule $e_k \in \mathcal{E}$ and a decryption rule $d_k \in \mathcal{D}$ such that $e_k : \mathcal{P} \longrightarrow \mathcal{C}$, $d_k : \mathcal{C} \longrightarrow \mathcal{P}$ and $\forall x \in \mathcal{P}$, $d_k(e_k(x)) = x$. ■

The sets $\mathcal{P}$ and $\mathcal{C}$ are commonly derived from alphabets, which could be a variety of things, such as English letters, Greek letters, Chinese characters, or even integers. This definition allows us to characterize what exactly is needed for a two individuals, most commonly referred to as Alice and Bob, to communicate over an insecure line, possibly under the

observation of another individual (usually dubbed Oscar). Suppose Alice would like to send the plaintext message $x \in \mathcal{P}$ to Bob. Under the specifics of their mutual cryptosystem, Alice would choose a key $k \in \mathcal{K}$ and encrypt x by using e_k. Then she would send the encrypted message $e_k(x) = m \in \mathcal{C}$ to Bob. Upon receipt of m, Bob would then compute $d_k(m) = d_k(e_k(x)) = x$ to retrieve the original plaintext. During the transfer Oscar has access to m, and thus the integrity of the cryptosystem depends on how difficult it would be for Oscar to determine x from m, i.e. how difficult it is for Oscar to find d_k. Oscar can attack the system in other ways, such as a *known plaintext* attack, in which he sends Alice a selected plaintext, and then observes the ciphertext in hopes to determine the key. A complete description of the potential attacks and security needs of a cryptosystem can be found in [55].

Most ciphers developed in the past (up to about twenty years ago) are known as *private-key cryptosystems*. Examples of these types of cryptosystems include the Vigenère Cipher, the Hill Cipher, and the Affine Cipher. Descriptions and analyses of these can found in [55]. In a private-key system, the sender and recipient must both have prior knowledge of the key k for any secure communication to take place. Thus either there must be a secure channel to communicate the key or some other means of mutual agreement on the key. If a key is set and then used for a long period of time, there are many opportunities for Oscar to find the key and compromise the system. And if there is a secure channel for key communication, do you even need to encrypt your messages? The answer, of course, is yes, for the secure channel may be expensive or inconvenient to use.

Suppose we had a cryptosystem such that it would be computationally infeasible or impossible to compute the decryption rule d_k from e_k. Then each user could actually publish their respective e_k in some sort of file or directory so that someone else could send them a message using this e_k. The recipient would be the only one who knows d_k, and thus could be the only one with the ability to decrypt the message. This is the general idea behind a *public-key cryptosystem*. This idea was first introduced in 1976 by Diffie and Hellman [15], and has become widely used in many cryptosystems at present.

Although it may be computationally infeasible to determine d_k from e_k, a public-key system is not unconditionally secure. For most practical purposes though, the integrity of a public-key system is regarded as being equivalent to the infeasibility of finding d_k. Thus, in order for a public-key system to be considered appropriate for use, e_k must be defined in a way such that the mapping e_k itself is easy to compute, but its inverse d_k is difficult to find. In other words, e_k should be a 'one-way' function. However, it must not be entirely one-way, as the recipient must have a way to compute $d_k(x)$. Hence the idea of a 'trapdoor' is needed for this one-way function, i.e., some extra information that only the recipient knows that will help in decrypting the ciphertext.

There are a few different types of one-way functions that are used in current cryptosystems. One such function is large integer multiplication, as the factorization of large integers is, in general, a hard problem. This is the basis for the security of the RSA cryptosystem, invented in 1977 by Rivest, Shamir, and Adleman [50]. Some other public-key cryptosystems include

the Merkle-Hellman Knapsack, the McEliece cryptosystem, the ElGamal cryptosystem, the Digital Signature Standard (DSS), the Chor-Rivest cryptosystem, and the Elliptic Curve cryptosystem, each of which exploit the properties of a supposed one-way function. One particular function that is believed to be one-way is exponentiation in certain cyclic groups. The next section will describe this function, and present an example of a cryptosystem that uses it.

1.3 The Discrete Logarithm Problem

Before we begin to describe the discrete logarithm problem, we must first give a few definitions and results relevant to the algebraic setting. Standard references in group theory include [23], [7], and [17]. Let G be a finite cyclic group with identity e, and let $g \in G$. The *order* of g in G is the smallest positive integer m such that $g^m = e$. We say that g is *primitive* in G if $m = |G|$ (here $|G|$ denotes the cardinality of G). Note that g is also called a *generator*. We now define the mapping $\log_g : G \longrightarrow \mathbb{Z}_m$ (the integers modulo m) by, for $a = g^x \in G$,

$$\log_g : a \longmapsto x. \tag{1.1}$$

We call $\log_g a$ the *logarithm of a with base g* . Note that the discrete logarithm function $\log_g$ is one-to-one and onto if and only if g is a generator. This is analogous to the logarithm function over the real numbers. The discrete logarithm function shares some of the same properties as its real counterpart, for example we have that $\log_g g^x = x$, $\log_g ab = \log_g a + \log_g b \mod m$, and $\log_g ab^{-1} = \log_g a - \log_g b \mod m$. It is also important to note that our choice of generator does not matter, for if $g, h \in G$ are both generators for G, then we have

$$\log_g a \equiv \log_g h \cdot \log_h a \mod m \tag{1.2}$$

just as we do in the real case. The *discrete logarithm problem* is as follows: given a finite cyclic group G with generator g, and $a \in G$, find $\log_g a$.

Note that the description given here is for a group whose operation is multiplicative in nature, a similar formulation for an additive group would be: find the unique integer x such that $x \cdot g = a$. This problem is, in general, not so difficult for the additive case, except in the case of elliptic curve groups, which we will mention later. A good general description of the discrete logarithm problem as well as many applications of it can be found in [36] (Chapter 6). Often we are interested in the case where G is the group of nonzero elements in a finite field. Let $q = p^n$ be a prime power, and let $\mathbb{F}_q$ denote the field of q elements. Note that if $n = 1$, then $\mathbb{F}_q \cong \mathbb{Z}_p$ (the integers modulo p).

1.4 The Diffie-Hellman Key Exchange System

As we mentioned before, in their seminal paper of 1976, Diffie and Hellman [15] introduced the notion of public-key cryptography. They also introduced a key exchange system that

enabled Alice and Bob to agree (publicly) on a key. For a mutually agreed upon group G and generator g, Alice chooses a random integer a and computes g^a. At the same time, Bob also chooses a random integer b and computes g^b. Then Alice sends g^a to Bob, and he sends g^b to Alice. Each of them now can compute g^{ab} privately (as Alice knows a and Bob knows b), which they will use as a key. The security of such a key-exchange system is based on the *Diffie-Hellman assumption* [27].

Definition: (The Diffie-Hellman Assumption) It is computationally infeasible to compute g^{ab} knowing only g^a and g^b. ■

One can easily see that the Diffie-Hellman assumption implies that the computation of discrete logarithms in the group is infeasible (i.e., if one can feasibly compute discrete logarithms, then the Diffie-Hellman assumption fails). The converse implication is still an unresolved issue, as there may exist a way to determine g^{ab} from g^a and g^b without needing to know a or b.

1.5 The ElGamal Cryptosystem

In 1985 T. ElGamal [16] presented a public-key cryptosystem that bears his name based on the intractibility of the discrete logarithm problem in $\mathbb{Z}_p^*$ (here $\mathbb{Z}_p^*$ denotes the nonzero elements of $\mathbb{Z}_p$, p a prime). The cryptosystem is defined as follows:

Definition: The ElGamal Cryptosystem: Let p be a prime and let α be primitive in $\mathbb{Z}_p^*$. Let $\mathcal{P} = \mathbb{Z}_p^*$, $\mathcal{C} = \mathbb{Z}_p^* \times \mathbb{Z}_p^*$, and define

$$\mathcal{K} = \{(p, \alpha, a, \beta) : \beta = \alpha^a \mod p\}. \tag{1.3}$$

The component a of a key in $\mathcal{K}$ is to be kept secret and all other components of the key are public. For a key $k = (p, \alpha, a, \beta)$, and for a random $r \in \mathbb{Z}_{p-1}$, define

$$e_k(x, r) = (y_1, y_2), \tag{1.4}$$

where $y_1 = \alpha^r \mod p$ and $y_2 = x\beta^r \mod p$ for $x \in \mathcal{P}$. Then we define $d_k(y_1, y_2) = y_2(y_1^a)^{-1} \mod p$. ■

Now the most important component of the key, a, is chosen by the recipient (Bob) and is unknown to everyone else. In Bob's public file, he will publish p, α, and β, so anyone wishing to send him a message can use this information to do so. In essence, when Alice wants to send x to Bob, the message is camouflaged by multiplying it by β^k for the random k she has chosen. Bob can compute x by first computing β^k from α^k, and then multiplying y_2 by $(\beta^k)^{-1}$. Oscar only sees (y_1, y_2), and, if the Diffie-Hellman assumption holds, cannot determine the plaintext unless he knows the secret exponent a.

We illustrate the use of the ElGamal cryptosystem with a small example. In no way is this cryptosystem secure, for the field $\mathbb{Z}_p$ is far too small.

EXAMPLE 1.1: Suppose that Alice wants to send the message "`It's not enough`" to Bob. First, Alice condenses the message and breaks it up into five-letter blocks (the block size could be any predetermined length, we use five for ease of illustration), appending extra characters to meet the length specification. Alice gets the plaintext message `ITSNO TENOU GHXXX`. Then, using a rule to enumerate the English alphabet given by

$$\mathrm{A} = 00,\ \ \mathrm{B} = 01,\ \ \mathrm{C} = 02, \ldots, \mathrm{Y} = 24,\ \ \mathrm{Z} = 25, \tag{1.5}$$

she now wishes to encrypt the following numbers:

$$\begin{aligned} x_1 &= 0819181314 \\ x_2 &= 1904131420 \\ x_3 &= 0607232323 \end{aligned} \tag{1.6}$$

Alice looks up Bob's public file and finds the information necessary to encrypt x_i, $i = 1, 2, 3$ using Bob's public key.

Bob's Public File
$p = 3000000019$
$\alpha = 3$
$\beta = 1351124895$

Note that Bob's public file does not display his secret exponent a. In this case $a = 13007$, as $3^{13007} \equiv 1351124895 \mod p$. Using the ElGamal protocol, she chooses random r_i, $i = 1, 2, 3$, and encrypts the x_i as

$$\begin{aligned} r_1 = 42734110 \implies y_{11} &= \alpha^{r_1} \mod p \equiv 1314668642 \\ y_{12} &= x_1 \cdot \beta^{r_1} \mod p \equiv 1573841406 \\ r_2 = 991431 \implies y_{21} &= \alpha^{r_2} \mod p \equiv 1956031587 \\ y_{22} &= x_2 \cdot \beta^{r_2} \mod p \equiv 1371497105 \\ r_3 = 2271409788 \implies y_{31} &= \alpha^{r_3} \mod p \equiv 1731734287 \\ y_{32} &= x_3 \cdot \beta^{r_3} \mod p \equiv 1784069351 \end{aligned}$$

Thus Alice sends the three ordered pairs, (y_{11}, y_{12}), (y_{21}, y_{22}), and (y_{31}, y_{32}) to Bob. Upon receipt of the ordered pairs, Bob can decipher the message by calculating $x_i = y_{i2} \cdot (y_{i1}^a)^{-1} \mod p$. He does so and computes

$$\begin{aligned} x_1 &= 1573841406 \cdot (1314668642^{13007})^{-1} \mod p \equiv 819181314 \\ x_2 &= 1371497105 \cdot (1956031587^{13007})^{-1} \mod p \equiv 1904131420 \\ x_3 &= 1784069351 \cdot (1731734287^{13007})^{-1} \mod p \equiv 607232323. \end{aligned}$$

Bob, noticing that x_1 and x_3 are only nine digits in length, prepends a 0 to each and the translates the plaintext back to English text, recovering the original message. ▲

In the above example, it is important to note that Alice did not need to know Bob's secret exponent a to encrypt, and that Bob did not need to know any of the random r_i generated by Alice to decipher. Oscar, with only knowledge of Bob's public file and the ciphertext ordered pairs, is not at any particular advantage here, as the r_i and a are unknown to him. This provides a measurable amount of security against Oscar as long as it is difficult to determine a and/or the k_i, i.e. as long as the discrete logarithm problem in $\mathbb{Z}_p^*$ is computationally infeasible.

1.6 Attacking the ElGamal Cryptosystem

We see from the definition and example of the ElGamal cryptosystem that its security can be compromised if Oscar can determine Bob's key (the secret exponent $2 \leq a \leq p-2$) from the given generator α and the field element β, or recover the plaintext (find the k_i) from the ciphertext. Perhaps the simplest way to approach this would be to just try each possible exponent. There are only a finite number of them, so as long as time is not a consideration, this method is guaranteed to work. In other words, this method, which we will dub 'brute force,' is *deterministic*.

It is recommended that (as of 1995 in [55]), for using the ElGamal cryptosystem, our prime p be at least 150 decimal digits in length. Thus if one had computing resources that could perform around a billion (10^9) operations per second, it would take around 10^{141} seconds to find a on average, which translates to about 3×10^{133} years. Not exactly realistic. Even if one had a million such machines, each of which searched different ranges of 2 to $p-2$ for the exponent, it would still take on the order of 10^{127} years to find the correct a. Hence the search begins for faster and more efficient algorithms than brute force, which requires $O(p)$ time to determine a single discrete logarithm in $\mathbb{Z}_p^*$.

2 Algorithms for the Discrete Logarithm Problem

2.1 Different Approaches

Obviously we cannot feasibly attack the discrete logarithm problem by brute force in large groups. Hence we must use a little mathematical finesse to have any chance of finding logarithms in large fields in our lifetime. Many different types of algorithms to attack the discrete logarithm problem have been formulated over the last few years, all of which fall into three classes:

- Algorithms which work for arbitrary groups, regardless of structure.

- Algorithms for finite groups in which the order of the group has no significantly large prime factors.
- The index calculus methods.

We will look at an example of each of the first two types, and then study the index calculus methods in detail, as they are the most powerful algorithms at present.

2.2 Shanks' Algorithm

This algorithm, due to Shanks [53], gives a basic improvement on the brute force method of computing logarithms via a time-memory tradeoff. Let G be a finite cyclic group with order (cardinality) n, and let α be a generator for G. Then for every $\beta \in G$, we have that $\log_\alpha \beta$ is an element of $\mathbb{Z}_n$, and thus can be written in the form $mj+i \mod n$, where $m = \lceil \sqrt{n} \rceil$ and $0 \leq i \leq m-1$ and $0 \leq j \leq m-1$. Thus to find $\log_\alpha \beta$, we start by computing a separate list L_1 where

$$L_1 = \{(j, \alpha^{mj}) : 0 \leq j \leq m\}.$$

Then for any given $\beta \in G$, we would compute a single $(i, \beta\alpha^{-i})$ and then scan the list L_1 (using a binary search) for a matching second entry. If we do not find one, then we would go on to the next i and do the same. When we do find one, we can then solve for $\log_\alpha \beta$. If $(j, y) \in L_1$ and we find a pair (i, y) we have

$$\alpha^{mj} = y = \beta\alpha^{-i}$$

or

$$\alpha^{mj+i} = \beta. \tag{2.1}$$

This simple routine can be implemented in $O(\sqrt{n} \log n)$ time and $O(\sqrt{n})$ memory [34], giving a significant gain over the brute force run time of $O(n)$. In Table 1 we present Shanks' algorithm for use in the field $\mathbb{Z}_p$:

Table 1: Shanks' algorithm for discrete logarithms in $\mathbb{Z}_p$

1	Let $m = \lceil \sqrt{p-1} \rceil$
2	Compute $\alpha^{mj} \mod p$, $0 \leq j \leq m-1$
3	Sort the m ordered pairs $(j, \alpha^{mj} \mod p)$ with respect to their second coordinates, obtaining a list L_1
4	Set $i = 0$
5	While $\beta\alpha^{-i} \neq \alpha^{mj} \mod p$ for some j, $0 \leq j \leq m-1$
6	$\quad i = i + 1$
7	Define $\log_\alpha \beta = mj + i \bmod (p-1)$.

We see how the algorithm functions in the following (small) example.

EXAMPLE 2.1: Let $p = 541$, and let $\beta = 370$. Now $\alpha = 2$ is a generator for $\mathbb{Z}_p$, so our goal is to find $x = \log_\alpha \beta$. Thus $m = \lceil \sqrt{540} \rceil = 24$ and we have

$$\alpha^m = \alpha^{24} \equiv 265 \mod 541.$$

We now compute the ordered pairs $(j, \alpha^{mj}) = (j, 265^j \mod p)$ for $0 \leq j \leq 23$. We get the following list:

Table 2: Ordered pairs for the example

(j, α^{mj})	(j, α^{mj})
(0,1)	(12,241)
(13,27)	(1,265)
(6,115)	(3,307)
(14,122)	(23,309)
(18,124)	(10,312)
(17,125)	(8,368)
(9,140)	(19,400)
(16,174)	(15,411)
(7,179)	(2,436)
(21,198)	(11,448)
(4,205)	(20,505)
(5,225)	(22,534)

We begin computing the pairs $(i, \beta\alpha^{-i})$ for each i until we find a second coordinate that matches a second coordinate of the first list. Eventually we arrive at the ordered pair (12,115), and 115 is also the second coordinate of the list pair when $j = 6$. Thus we set

$$\log_\alpha \beta = mj + i = 23 \cdot 6 + 12 \mod p - 1 \equiv 156.$$

A quick check determines that indeed $2^{156} \equiv 370 \mod 541$. ▲

Shanks' algorithm can be easily implemented, given p, α, and β using the number theoretical computation package GP/Pari with the following code (due to Locke [22]):

```
m=ceil(sqrt(p-1));
z=matrix(m,3,j,k,0);
for(j=1,m,z[j,1]=j-1;z[j,2]=mod(alpha^(m*(j-1)),p);\
z[j,3]=beta*mod(alpha^(1-j),p));\
for(j=1,m,for(k=1,m,if(z[k,3]==z[j,2],j1=j-1;k1=k-1,)));\
x=lift(mod(m*(j1)+(k1),p-1));
```

The output x is the logarithm of β in base α. Note that this code does not do the sorting recommended by the algorithm, instead it just runs through the lists and compares each element. This could be optimized easily, our purposes do not require it, as we will not compute any logarithms in a large group using this code.

Shanks' algorithm is generally studied only for theoretical purposes as it too is not practical for very large fields. There is a similar algorithm that was developed by Pollard [48] that actually has the same running time with no storage requirements. A discussion of this algorithm can be found in [34].

2.3 The Silver-Pohlig-Hellman Algorithm

We now move to finding discrete logarithms in groups that have smooth orders.

Definition: A *smooth* integer is an integer that has only small prime factors. We say that an integer is m-smooth if all of its prime factors are less than or equal to m. ■

When G has a smooth order, it may be easier to compute the logarithm of an element modulo the prime power factors of the order, and then use the Chinese Remainder Theorem to compute the logarithm modulo the order (see [44] for a discussion of the Chinese Remainder Theorem). This is the approach in the algorithm developed by Pohlig and Hellman [47] and independently by R. Silver.

We begin the description of the algorithm for the multiplicative group of the field $\mathbb{Z}_p$ (the algorithm is applicable to any group, we use $\mathbb{Z}_p$ for clarity), with generator α and field element $\alpha^a = \beta$. Suppose

$$\prod_{i=1}^{k} p_i^{e_i} \tag{2.2}$$

is the distinct prime factorization of $p-1$, and let one of the prime factors and its exponent be denoted by ρ^e. We wish to find an integer x such that

$$x = a \mod \rho^e. \tag{2.3}$$

We can write x in a base ρ representation, i.e.,

$$x = \sum_{j=0}^{e-1} b_j \rho^j, \text{ where } 0 \le b_j \le \rho - 1, \quad j = 1, \ldots, e-1. \tag{2.4}$$

Now we can write $a = x + \rho^e t$ for some integer t, hence we have

$$\begin{aligned} \beta^{(p-1)/\rho} &\equiv \alpha^{(p-1)x/\rho + (p-1)\rho^{e-1}t} \\ &\equiv \alpha^{(p-1)\sum_{j=0}^{e-1} b_j \rho^{j-1}} \\ &\equiv \alpha^{(p-1)b_0/\rho} \mod p. \end{aligned} \tag{2.5}$$

We compute b_0 by calculating $\beta^{(p-1)/\rho}$ and $\gamma = \alpha^{(p-1)/\rho}$. Then we compute γ^i for $i = 0, 1, \ldots$ until we find $\gamma^i = \beta^{(p-1)/\rho}$ and then $b_0 = i$. To do this, we can use Shanks' algorithm. Next, provided $e \geq 2$, we can find b_1 as follows: let $h = \alpha^{-1} = \alpha^p$, and set $\beta_1 = \beta h^{b_0}$. We note that

$$\begin{aligned} \beta_1^{(p-1)/\rho^2} &\equiv \alpha^{(p-1)\sum_{j=1}^{e-1} b_j \rho^{j-2}} \\ &\equiv \gamma^{b_1} \mod p. \end{aligned} \tag{2.6}$$

Thus we search through the powers of γ until we find the value of b_1. Continuing on in this fashion we can find each b_i, $i = 0, 1, \ldots, e-1$, finally giving us the base ρ representation of x. We now present this algorithm in a shortened manner.

Table 3: The Silver-Pohlig-Hellman algorithm for $\mathbb{Z}_p$

1	Compute $\gamma = \alpha^{(p-1)/\rho} \mod p$
2	Set $j = 0$ and $\beta_i = \beta$
3	While $j \leq e - 1$ do
4	Compute $\delta \equiv \beta_i^{(p-1)/\rho^{i+1}} \mod p$
5	Find $0 < j < \rho$ such that $\delta = \gamma^j$ (by Shanks' algorithm)
6	$b_i = j$
7	$\beta_{i+1} \equiv \beta_i \alpha^{-b_i \rho^i} \mod p$
8	$i = i + 1$

As stated earlier, the method described only computes $\log_\alpha \beta \mod p_i^{e_i}$. Then the Chinese Remainder Theorem can be applied to find the correct exponent a.

EXAMPLE 2.2: Let $p = 1009$, and let $\beta = 891$. We have that $\alpha = 11$ is primitive, and we see that $p - 1 = 1008 = 2^4 3^2 7$. So, proceeding with the first step of the SPH algorithm, if $x = \log_\alpha \beta$, then we wish to find $x \mod 2^4$ (let $\rho^e = 2^4$). We begin by computing $\gamma_0 = 1$ and

$$\gamma_1 = 11^{\frac{1008}{2}} \mod p \equiv 1008.$$

Then we compute $\delta = 891^{\frac{1008}{2}} \mod p \equiv 1008$, and thus we see that $a_0 = 1$ (as $\gamma_1 = 1008$). We increment our index, compute $\beta_1 = 891 \cdot 11^{-1} \mod p \equiv 81$ and now compute δ again. We now have

$$\delta = 81^{\frac{1008}{4}} \mod p \equiv 1.$$

Thus $a_1 = 0$, and now $\beta_2 = \beta_1 \cdot (\alpha^{-0})^2 = \beta_1 = 81$. Our new δ is $\delta = 81^{\frac{1008}{8}} \mod p \equiv 1$, so again we have $a_2 = 0$. Then $\beta_3 = 81$, so $\delta = 81^{\frac{1008}{16}} \mod p \equiv 1008$, thus $a_3 = 1$. Thus we now have

$$x = \sum_{i=0}^{3} a_i 2^i = 1 + 1 \cdot 2^3 \equiv 9 \mod 16. \tag{2.7}$$

Now we switch modes and compute $x \mod 3^2$. We compute $\gamma_0 = 1$, $\gamma_1 = 11^{\frac{1008}{3}} \equiv 374 \mod p$, and $\gamma_2 = 11^{\frac{1008 \cdot 2}{3}} \equiv 634 \mod p$. Now we have

$$\delta = 891^{\frac{1008}{3}} = 374 \mod p$$

so $a_0 = 1$. Then $\beta_1 = 891 \cdot 11^{\frac{1008}{9}} \equiv 81 \mod p$, and then we have

$$\delta = 81^{\frac{1008}{9}} = 374 \mod p$$

and we have $a_1 = 1$ as well. Thus

$$x = \sum_{i=0}^{1} a_i 3^i = 1 + 1 \cdot 3^1 \equiv 4 \mod 9. \tag{2.8}$$

Moving to the final prime factor of $p - 1$, we wish to find $x \mod 7$. We know $\gamma_0 = 1$, and we must compute 6 different γ_i, they are

$$\begin{aligned}
\gamma_1 &= 11^{\frac{1008 \cdot 1}{7}} \equiv 935 \mod p \\
\gamma_2 &= 11^{\frac{1008 \cdot 2}{7}} \equiv 431 \mod p \\
\gamma_3 &= 11^{\frac{1008 \cdot 3}{7}} \equiv 394 \mod p \\
\gamma_4 &= 11^{\frac{1008 \cdot 4}{7}} \equiv 105 \mod p \\
\gamma_5 &= 11^{\frac{1008 \cdot 5}{7}} \equiv 302 \mod p \\
\gamma_6 &= 11^{\frac{1008 \cdot 6}{7}} \equiv 859 \mod p.
\end{aligned}$$

We only have one δ to compute, and it is $\delta = 891^{\frac{1008}{7}} \equiv 394 \mod p$, so $a_0 = 3$. Thus we have

$$x = \sum_{i=0}^{0} a_i 7^i \equiv 3 \mod 7. \tag{2.9}$$

Now, using the Chinese Remainder Theorem, we wish to find a unique solution to the congruences (2.7), (2.8), and (2.9) modulo $p - 1$. Computing this, we have

$$x \equiv 409 \mod 1008. \tag{2.10}$$

A check shows us that $11^{409} \equiv 891 \mod 1009$. ▲

These calculations seem very cumbersome when carried out by hand, but actually can be computed rather quickly if $p - 1$ has only small factors. As with Shanks' algorithm, we can implement the Silver-Pohlig-Hellman algorithm using GP/Pari, with following code taken from [22] (Lemmond).

```
pohlig(p,al,b)=y=p-1;\
               v=factor(y);\
               w=v;\
               for(j=1,omega(y),\
                   c=v[j,2];\
                   q=v[j,1];\
                   z=b;\
                   k=0;\
                   x=0;\
                   while(k-c,gam=1;\
                       e1=(p-1)/q^(k+1);\
                       del=mod(lift(z),p)^(e1);\
                       m=0;\
                       while(gam-del,gam=mod(lift(gam),p)*\
                            mod(lift(al),p)^((p-1)/q);m=m+1);\
                       x=x+m*q^k;\
                       q1=-m*q^k;\
                       z=mod(lift(z),p)*mod(lift(al),p)^(q1);\
                            k=k+1);\
                   Print("a congruent to ",x," mod ",q^c);\
                   w[j,1]=x;\
                   w[j,2]=q^c);\
               ans=mod(w[1,1],w[1,2]);\
               for(l=1,omega(y),\
                   ans=chinese(ans,mod(w[l,1],w[l,2])));\
               Print(ans);\
               Print("The log of ",lift(b)," to the base ",al,\
                   " is ",lift(ans)," in GF(",p,")");
```

An in-depth analysis of the Silver-Pohlig-Hellman algorithm can be found in [55] and [42]. This algorithm has a running time of

$$O\left(\sum_{i=1}^{k} e_i \left(\log(p-1) + p_i^{1/2} \log p_i\right)\right) \tag{2.11}$$

which we can see depends on the size of the largest prime factor of $p-1$.

2.4 Other Algorithms

There are other algorithms that can be classified in one of the first and second types listed at the beginning of this section, including those that attempt to perform a bit-wise computation of a discrete logarithm [45]. We will not go into those here, the reader should consult [55], [42], or [34].

3 The Basic Index Calculus Method

3.1 Introduction

We now begin our discussion of the strongest family of algorithms for finding discrete logarithms in a cyclic group. The ideas behind the index calculus method, originally thought to be due to Western and Miller [57], actually can be attributed to Kraitchik [28], [29] and Cunningham (see [56]). The ideas were implemented into algorithms for the discrete logarithm problem independently by Adleman [1], Merkle [37], and Pollard [48]. The theme is this: if we can find the discrete logarithms of some *small* and *independent* elements, then we should be able to determine logarithms of almost any element in the group, as most elements we can express in terms of the small independent elements whose logs are known. A reader familiar with linear algebra and vector spaces might begin to relate these small, independent elements as a sort of basis for such a space or subspace, and essentially it is the same sort of idea. However, we do not intend to consider these elements as a basis, and we will not be working in the vector space setting nor have any notion of dimension. But we will say that we intend to be able to express the logarithm of a group element as a linear combination of the logarithms of the elements in our *factor base*, which will consist of the small, independent elements we spoke of earlier.

The index calculus method, in any form, has three basic stages:

1. Generation of smooth relations involving the elements in the factor base.
2. Solving the corresponding linear system of equations to find the logarithms of the factor base.
3. Using the logarithms of the factor base to determine the discrete logarithm of any given group element.

It is important to point out one thing here: the first two stages of the index calculus method do not in any way depend on the element whose logarithm we are trying to find. In the setting where the field is $\mathbb{F}_q$ with generator α, if we are trying to find $\log_\alpha \beta$ for $\beta \in \mathbb{F}_q$, we are not concerned what β is until we get to the third stage. For this reason, the first two stages are often referred to as *precomputation* stages, as they can be done at any time as soon as the field in question is known (recall that the generator we use is not so important).

We now begin the description of the basic form of the index calculus method. For our group G with generator α, we choose a small number of 'prime' elements to place in our factor base $\mathcal{B}$ along with α. When we say prime, we essentially mean that the element does not factor into a product of elements 'smaller' than itself, for instance, if our group is a prime field (a field with a prime number of elements, like $\mathbb{Z}_p$), then we would include elements in our factor base that are small integer primes. The number of elements chosen for $\mathcal{B}$ is minute compared to the size of the group itself, for if $|\mathcal{B}|$ were large, the computation of the logarithms of the

factor base would be difficult. Then we try to find powers of the generator α that factor completely amongst the elements in $\mathcal{B}$. If we are successful, then we have a smooth relation, i.e., a congruence in the group that relates (linearly) logarithms of the elements of the factor base. Once we have obtained sufficiently many relations (enough to ensure that we will be able to solve for the logarithms of the elements in $\mathcal{B}$) we move on to Stage 2. In this stage we set up the corresponding linear system of equations and solve for the logs of the factor base. Note that we have to solve this system modulo $q-1$, which itself may be composite. If so, it may be necessary to factor $q-1$ and solve the linear system modulo the factors of $q-1$ and then employ the Chinese Remainder Theorem to obtain the final solution. Once the solution is obtained, we move to stage three, and attempt to find the logarithm of a field element. We multiply the field element by a random power of the generator in hopes that this product will factor over the factor base. If it does, we take the logarithm of the congruence and solve the linear equation for $\log_\alpha \beta$. If not, we discard the product and try again until we do succeed. To make the method a little clearer, in Table 4 we present the outline of the index calculus method.

Table 4: The index calculus method for finding $\log_\alpha \beta = a$

Stage 1:	Finding logarithms of the factor base
a.	Create the factor base $\mathcal{B} = \{p_1, p_2, \ldots, p_m\}$
b.	Compute the relations $\alpha^{x_j} \equiv p_1^{a_{1j}} p_2^{a_{2j}} \cdots p_m^{a_{mj}}$ for $1 \le j \le t$
Stage 2:	Solving for the logarithms of the factor base
a.	Set $x_j \equiv a_{1j} \log_\alpha p_1 + a_{2j} \log_\alpha p_2 + \ldots + a_{mj} \log_\alpha p_m$
b.	Solve the linear system $\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1t} & a_{2t} & \cdots & a_{mt} \end{bmatrix} \begin{bmatrix} \log_\alpha p_1 \\ \log_\alpha p_2 \\ \vdots \\ \log_\alpha p_m \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_t \end{bmatrix}$ to obtain the logarithms of the factor base
Stage 3:	Compute $\log_\alpha \beta = a$
a.	Choose a random s, $1 \le s \le q-2$, and compute $\gamma \equiv \beta \alpha^s$
b.	Then if $\gamma = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$ We have $\log_\alpha \beta \equiv c_1 \log_\alpha p_1 + c_2 \log_\alpha p_2 + \ldots + c_m \log_\alpha p_m - s$

One important note is that, in practice, we will only store the nonzero entries of the linear system to be solved in Stage 2. We will discuss this in more detail later when we talk about techniques for solving linear systems over finite fields. The number of smooth relations is often chosen to be $2m$ (twice the cardinality of the factor base). Usually this will guarantee us m linearly independent relations. Now as long as we have m independent relations, we should obtain a unique solution modulo $q-1$. Again we iterate that we need not know β until the third stage.

3.2 The Basic Index Calculus Method in Prime Fields

We now present an example that will display the ease and efficiency of the index calculus method in prime fields. It is particularly effective in these fields, as determining whether or not an integer is smooth is easy in many cases (the well-known difficulties in integer factorization are usually only found in integers with an extremely large number of digits that one is trying to factor completely).

EXAMPLE 3.1: Let $q = 14087$, and let $\alpha = 5$. We wish to find $\log_5 5872$, (i.e. $\beta = 5872$). We begin by choosing our factor base $\mathcal{B} = \{2, 3, 5, 7, 11, 13\}$, and we begin to search for smooth relations. After some 'lucky' choices for exponents, we have the following relations:

$$\begin{aligned}
5^{346} &\equiv 6776 \equiv 2^3 \cdot 7 \cdot 11^2 \mod q \\
5^{171} &\equiv 9152 \equiv 2^6 \cdot 11 \cdot 13 \mod q \\
5^{153} &\equiv 2457 \equiv 3^3 \cdot 7 \cdot 13 \mod q \\
5^{442} &\equiv 567 \equiv 3^4 \cdot 7 \mod q \\
5^{458} &\equiv 13608 \equiv 2^3 \cdot 3^5 \cdot 7 \mod q
\end{aligned}$$

so Stage 1 is complete. Taking the logarithm in base 5 of both sides and letting $L_2 = \log_5 2, L_3 = \log_5 3$, etc., we have the following system of equations modulo 14086:

$$\begin{aligned}
3L_2 + L_7 + 2L_{11} &= 346 \\
6L_2 + L_{11} + L_{13} &= 171 \\
3L_3 + L_7 + L_{13} &= 153 \\
4L_3 + L_7 &= 442 \\
3L_2 + 5L_3 + L_7 &= 458
\end{aligned}$$

Letting $\mathbf{A}$ be the coefficient matrix and $\mathbf{b}$ the right-hand side vector we now must solve the matrix equation $\mathbf{AL} = \mathbf{b}$ or

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 0 \\ 6 & 0 & 0 & 1 & 1 \\ 0 & 3 & 1 & 0 & 1 \\ 0 & 4 & 1 & 0 & 0 \\ 3 & 5 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} L_2 \\ L_3 \\ L_7 \\ L_{11} \\ L_{13} \end{bmatrix} = \begin{bmatrix} 346 \\ 171 \\ 153 \\ 442 \\ 458 \end{bmatrix} \mod 14086.$$

However, since the ring $\mathbb{Z}_{14086} \cong \mathbb{Z}_2 \times \mathbb{Z}_{7043}$ contains zero divisors, we run into a difficult problem when performing Gaussian elimination on the augmented matrix. In fact, the element 2 has no inverse modulo 14086. Thus, we solve the linear system $\mathbf{AL} = \mathbf{b}$ twice,

first modulo 2, then modulo 7043. We get the solutions

$$L = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \mod 2 \qquad \text{and} \qquad L = \begin{bmatrix} 3028 \\ 5018 \\ 1499 \\ 5446 \\ 4729 \end{bmatrix} \mod 7043.$$

Then applying the Chinese Remainder Theorem yields the logarithms

$$\begin{aligned} L_2 &= 3028 \\ L_3 &= 5018 \\ L_7 &= 8542 \\ L_{11} &= 5446 \\ L_{13} &= 4729 \end{aligned}$$

which completes Stage 2.

Stage three begins with choosing some random exponents s, calculating $\beta\alpha^s$, i.e. $5872 \cdot 5^s \mod q$, and factoring the result. After a few tries, we find that

$$5872 \cdot 5^{145} \equiv 1404 \equiv 2^2 \cdot 3^3 \cdot 13 \mod 14087$$

and thus taking the $\log_5$ of the equivalence gives

$$\begin{aligned} \log_5 5872 &= 2L_2 + 3L_3 + L_{13} - 145 \\ &\equiv 2(3028) + 3(5018) + 4729 - 145 \equiv 11608 \mod 14086 \end{aligned}$$

A quick check determines that indeed $5^{11608} \equiv 5872 \mod 14087$. ▲

The above example seems to require an excessive number of computations in comparison to the size of the field. This is not true of the index calculus method in general, and a small example like the one above does not adequately illustrate the improvement that index calculus brings. As we move into much larger fields, the computations needed for index calculus pale in comparison to those required by Shanks' algorithm and the Silver-Pohlig-Hellman algorithm. McCurley [34] describes how the running time of the index calculus algorithm is found. Define the function

$$L(q) = \exp(\sqrt{\log q \log\log q}) \tag{3.1}$$

for a prime q. Pomerance [49] shows that the running time for stages one and two of the index calculus method in $\mathbb{F}_q$ is given by

$$L(q)^{2+o(1)} \tag{3.2}$$

and the third stage complexity is $L(q)^{3/2+o(1)}$. There are variants of the basic index calculus method due to Pomerance [49] as well as Coppersmith, Odlyzko, and Schroeppel [13], the latter including a method that uses an isomorphism between $\mathbb{F}_q$ and the ring of Gaussian integers modulo a maximal ideal. One can consult [51] or [42] for resources on these and other types of algorithms not discussed in detail here.

3.3 The Basic Index Calculus Method in Nonprime Finite Fields

Although we have previously discussed a few different algorithms for finding discrete logarithms in a finite field, we now digress a little and cover some more involved fundamentals of finite field theory. The seminal reference of Lidl and Niederreiter [32] is recommended for the reader who wishes to see more background than we present here, and [36] provides elaboration and many applications of finite fields.

3.3.1 Fields of order p^n, p prime, $n > 1$

It is well known that for every prime number p and every positive integer n, there exists a unique field of p^n elements [32]. In fact, the elements of the field are the roots of the polynomial $x^{p^n} - x$ in the algebraic closure of $\mathbb{F}_p$.

Now $\mathbb{F}_{p^n} \cong \mathbb{F}_p[x]/(f(x))$, where $\mathbb{F}_p[x]$ is the ring of polynomials with coefficients in $\mathbb{F}_p$ and $(f(x))$ is the ideal generated by the irreducible polynomial $f(x)$ (irreducible in $\mathbb{F}_p[x]$). Let $\omega \in \mathbb{F}_{p^n}$ be a root of $f(x)$ (i.e., $\omega = x \mod f(x)$). Then all elements of $\mathbb{F}_{p^n}$ are of the form $\sum_{i=0}^{n-1} a_i \omega^i$, where $a_i \in \mathbb{F}_p$. We can represent these elements as polynomials of degree less than n with coefficients in $\mathbb{F}_p$. We now illustrate the elements of a nonprime finite field with the following example.

EXAMPLE 3.2: Consider the field $\mathbb{F}_9 = \mathbb{F}_{3^2}$ where $\mathbb{F}_9 \cong \mathbb{F}_3[x]/(f(x))$ and $f(x) = x^2 + 2x + 2$. Then all of the elements of this field are represented by polynomials of degrees less than 2 with coefficients in $\mathbb{F}_3$. Let ω be the residue class corresponding to $x \mod f(x)$. Then these elements are 0, 1, 2, ω, 2ω, $\omega + 1$, $2\omega + 1$, $\omega + 2$, and $2\omega + 2$.

Using the fact that $\omega^2 \equiv \omega + 1$, we can find any product of elements in this field. For example, we have that

$$(2\omega + 1)(\omega + 2) = 2\omega^2 + 2\omega + 2 \equiv 2(\omega + 1) + 2\omega + 2 = \omega + 1.$$

Other products and inverses can be found similarly. ▲

For the remainder of this paper, we will consider elements of nonprime finite fields as polynomials in $\mathbb{F}_p[x]$ of degree less than n. Doing so allows for easy representation and manipulation

of these elements.

There are other ways to represent a finite field. One way is to find an algebraic number field K of degree n over $\mathbb{Q}$ such that p remains a prime in the ring $\mathcal{O}$ of integers in K. Then $\mathcal{O}/(p) \cong \mathbb{F}_{p^n}$. Also, there have been algorithms to find discrete logarithms designed specifically for use in alternate field representations ([3] and [33]). We will not discuss those here, as they are generally more difficult to understand and analyze.

3.3.2 Computational notes

Up to this point in this paper, any of the computations involved in computing discrete logarithms could be done by using a computation package that included routines for modular arithmetic, such as GP/Pari or Maple. As we move into computations involving polynomials over finite fields, a more efficient method of computation may be necessary, as well as more predefined routines related to polynomials over finite fields. For this we will employ a C++ class library by the name of NTL (Number Theory Library). This package was developed especially for computations involving finite fields, polynomials, and arbitrary length integers. It includes various routines and structures such as polynomial factorization routines, polynomial modular arithmetic, and more. NTL is developed by Victor Shoup, and can be found at the internet address `http://www.cs.wisc.edu/~shoup/ntl/`.

Also it is important to point out that the main focus of our study of the index calculus method will be in fields of order 2^n. These fields are commonly used in practice, for their ease of computer representation and arithmetic. For example, an element of $\mathbb{F}_{2^n}$ can be represented as a binary string of length n, as the element is actually a polynomial with coefficients in $\mathbb{F}_2$ of degree less than n. NTL has a special class structure for these polynomials by the name of `BB`. The appendix contains many of the routines used in computations relevant to our work. Another class that was used quite often is the class `ZZ`, and variables of this type are arbitrary-length integers. This is quite useful when an integer you are trying to represent has more binary digits than the length of your particular computer's word size.

Throughout the rest of this paper we will present many computational results, and some of these results will have an actual computation time associated with it. The times are presented merely for comparison purposes, for more reasons than one. First, the computations were done on a variety of machines, including Sun SparcStation 5s, Sun Ultras, and even an IBM PC running Linux. Also, the resources available were adequate at best when compared to the high-performance computing power available today to those who wish (and have the means) to have it. We should also point out that we do not claim to present the most efficient code possible for these routines, it is almost certain that their performance could be improved in some way.

3.3.3 Primitive elements in $\mathbb{F}_{p^n}$

Just as in the case where the cardinality of a finite field is prime, using the representation described in the previous section we would like to determine conditions on an element in $\mathbb{F}_{p^n}$ being primitive. In particular, for use in the index calculus method for discrete logarithms, we wish to find out when the field element x is primitive. We have that x generates the field if and only if the monic irreducible polynomial $f(x)$ is a *primitive polynomial*. In the example field in the section 5.1, it turns out that $f(x) = x^2 + 2x + 2 \in \mathbb{F}_3[x]$ is primitive, and thus x is a generator for the group of nonzero elements of the field.

In each field we work in, we want to be able to determine whether or not an irreducible polynomial is primitive. If it is, and x is a generator for the field, then the order t of x is $t = p^n - 1$. A theorem of Lagrange gives way to the fact that any element of a finite group has order that divides the order of the group. When we are working in a finite field of order p^n, the nonzero elements of the field form a cyclic group of order $p^n - 1$. So if the order of x is $p^n - 1$, then $x^t \neq 1$ for any $t < p^n - 1$. Thus we test primitivity of $f(x)$ by raising x to a distinct prime factor of $p^n - 1$, reduce it modulo f, and then check to see if it is 1. If it is not for all primes dividing $p^n - 1$, then $f(x)$ is primitive. In Appendix A we present a routine (called `primpoly.c`) for computing the 'smallest' primitive polynomial for fields of order 2^n. Into this subroutine, we must input the exponent of 2 as well as the distinct prime factors of $2^n - 1$ in each implementation. This factorization is not always easy to come by. However, we have access to many precomputed factorizations of the form $2^n - 1$ for n odd, $2^n + 1$ for n odd, and other forms involving even exponents that are not divisible by 4. We only need those such factorizations, for if we wish to factor $2^n - 1$ for even n, then we factor this as $2^{2k} - 1 = (2^k - 1)(2^k + 1)$. These tables can be found at the internet address `ftp://www.cs.purdue.edu/pub/ssw/`.

We should clarify what we mean by 'smallest.' Consider the coefficient vector of the polynomial as the binary representation of an integer. To illustrate, we see that $x^6 + x^3 + x^2 + x$ corresponds to $1001110_2 = 78$. We will use this correspondence between polynomials and integers often in work to come, especially when we deal with output and vectors of polynomials. The 'smallest' primitive polynomial of degree n is the polynomial with the smallest coefficient vector (when viewed as an integer) such that the polynomial is primitive. Table 5 presents a brief list of the smallest polynomials $f_1(x) \in \mathbb{F}_2[x]$ such that $f(x) = x^n + f_1(x)$ is primitive, along with the computation time needed to find $f(x)$. It has been argued (but not yet proven) that, for any degree n, there exists a primitive polynomial $f(x) = x^n + f_1(x)$ where the degree of f_1 is smaller than $\log n + c$ for some constant c. Now using the smallest possible primitive polynomial to define the field will result in faster finite field arithmetic, especially exponentiation and multiplication. Thus it is to our advantage to use the smallest primitive polynomial in our computations. We are at liberty to use this particular primitive, as it does not matter much which polynomial defines our field. For example, suppose we can compute discrete logarithms in a field we define by the smallest primitive polynomial. Then if the finite field being used in a cryptosystem we wish to attack is defined by a different polynomial, we can construct an isomorphism between our field and the field being used,

Table 5: Primitive polynomials in $\mathbb{F}_2[x]$ of selected degrees

Degree	$f_1(x)$	Time (sec.)
10	x^3+1	0.002621
15	$x+1$	0.001669
20	x^3+1	0.003324
25	x^3+1	0.002874
30	$x+1$	0.002507
35	x^2+1	0.001128
40	$x^5+x^4+x^3+1$	0.006378
45	x^4+x^3+x+1	0.003739
50	$x^4+x^3+x^2+1$	0.003908
60	$x+1$	0.001539
70	x^5+x^3+x+1	0.00799
80	$x^7+x^5+x^3+x^2+x+1$	0.027109
90	$x^5+x^3+x^2+1$	0.009555
100	$x^6+x^5+x^2+1$	0.018834
120	x^4+x^3+x+1	0.013652
140	x^6+x^4+x+1	0.020329
160	$x^5+x^3+x^2+1$	0.02042
180	x^3+1	0.01039
200	$x^5+x^3+x^2+1$	0.022712
225	$x^8+x^6+x^5+x^3+x^2+1$	0.122697
250	$x^6+x^5+x^3+x^2+x+1$	0.040233
300	x^5+1	0.037401
350	$x^6+x^5+x^2+1$	0.075309
400	$x^5+x^3+x^2+1$	0.069271
500	$x^8+x^6+x^5+x^2+x+1$	0.93002
600	$x^7+x^6+x^5+x^4+x^2+1$	1.13752

which will allow us to compute discrete logarithms in that field as well [42].

3.4 Factoring Polynomials over $\mathbb{F}_p[x]$

As we see from example 4.1, the most time-consuming portions of the index calculus method are a) generating smooth relations and b) solving the system of congruences. We will speak later on the solution of linear systems over finite fields, at present we focus on Stage 1, the acquisition of smooth relations. In the case of a nonprime finite field, our smooth relations are polynomial equivalences, and we must factor these polynomials in order to proceed on to Stage 2. Thus we must be able to factor polynomials over a finite field efficiently.

There are (at present) many well-known algorithms for polynomial factorization over a fi-

nite field, the most famous of which is due to Berlekamp [6]. Other algorithms have been discovered by Niederreiter [39], [40], Cantor and Zassenhaus [9], von zur Gathen and Shoup [19], Kaltofen and Shoup, and others. New ones are constantly being developed, [24] and [25] are survey sources for polynomial factorization. Many of these algorithms involve a 'linearization' idea, as we attempt to express the polynomial to be factored in terms of smaller irreducibles via the Chinese Remainder Theorem for Polynomials. [32] provides a good explanation of how Berlekamp's algorithm and others use this approach to factoring.

The computation package NTL provides the routine `CanZass`, based on the Cantor and Zassenhaus approach, for factoring polynomials in $\mathbb{F}_2[x]$. Table 6 gives factorization time statistics for polynomials in $\mathbb{F}_2[x]$. For each degree, we select 1000 random (pseudorandom, as they are generated by the computer) and record the time necessary for `CanZass` to factor each one. This routine performs well for polynomials of relatively low degrees, but as n tends

Table 6: `CanZass` factorization times

Degree	factorization time (sec)		
	Mean	Min	Max
100	.003	.0024	.0224
200	.092	.0058	.0289
300	.0214	.012	.0363
400	.0425	.0218	.071
500	.0701	.0327	.1181
600	.1146	.0534	.2086
700	.1761	.0736	.3087
800	.3421	.1347	.7629
900	.371	.1536	.6857
1000	.4899	.1616	.8081
1500	1.604	.5878	7.3323
2000	2.9162	1.2151	4.8602
2500	5.8771	2.3788	11.0257
3000	9.459	3.8595	20.5516
4000	17.5136	6.7097	28.5565
5000	34.7737	65.3569	13.6171

to infinity, the mean computation time increases rapidly. Thus, it would be computationally infeasible (at present) to factor a large number of polynomials of fairly large degree, say $n > 10000$ with this routine. However, other algorithms may be able to do such factorizations in reasonable time.

3.5 Implementation of Basic Index Calculus in $\mathbb{F}_{2^n}$

We are now ready to apply the index calculus method to a field of cardinality 2^n. We must first determine our factor base $\mathcal{B}$. It is recommended that $\mathcal{B}$ should consist of all irreducible

polynomials in $\mathbb{F}_2[x]$ of degree up to some specified number B. It is known [35] that the formula

$$I_n = \frac{1}{n}\sum_{d|n}\mu(d)q^{n/d} \tag{3.3}$$

gives the number of irreducible polynomials in $\mathbb{F}_q[x]$ of degree n. Here $\mu(d)$ is the *Möebius μ-function*. Thus, for a selected degree B, we have that our factor base size will be

$$|\mathcal{B}| = \sum_{j\leq B} I_j = \sum_{j\leq B}\frac{1}{j}\sum_{d|j}\mu(d)q^{j/d}. \tag{3.4}$$

Upon examination of this quantity for selected B, we see that the base size grows rapidly as n gets larger (see Table 7).

For even larger B, we can estimate the quantity (3.4) by

$$I_n \approx \frac{q^n}{n} \tag{3.5}$$

as the dominant term in (3.4) occurs when $d = 1$. Thus we can estimate $|\mathcal{B}|$ by

$$|\mathcal{B}| \approx \sum_{j\leq B}\frac{q^j}{j}. \tag{3.6}$$

This leads us to an important question: How should we choose B to minimize the overall running time of the index calculus algorithm? On one hand, a smaller B will require less computation time for Stage 2 (as the linear system will have $|\mathcal{B}|$ columns and usually around $2|\mathcal{B}|$ rows) but it may be more difficult to find B-smooth relations. On the other hand, a larger B will make stage one much quicker (as smooth relations will be easier to come by), but the corresponding linear system becomes more difficult to solve. A first impression tells us to make B smaller, as it seems that solving a larger linear system will be more time and space requiring than finding relations that are one degree smoother, as one more element in $|\mathcal{B}|$ means one extra column, which will require two more rows, thus adding a total of around $4|\mathcal{B}|$ entries. This is discussed in detail in [42], and for the basic index calculus method it is recommended that we choose B to be

$$B \sim c(n\log n)^{1/2} \tag{3.7}$$

where $c = (4\log 2)^{-1/2}$. The best way to illustrate how the method works is by examining a (large but small) example.

Table 7: Factor base size for selected B

B	$\lvert\mathcal{B}\rvert$	B	$\lvert\mathcal{B}\rvert$
3	5	31	143522117
4	8	32	277737797
5	14	33	538038783
6	23	34	1043325198
7	41	35	2025032004
8	71	36	3933898964
9	127	37	7648465274
10	226	38	14882080607
11	412	39	28978383317
12	747	40	56466147791
13	1377	41	110100861341
14	2538	42	214816204142
15	4720	43	419376506984
16	8800	44	819198821759
17	16510	45	1601073756327
18	31042	46	3130828882176
19	58636	47	6125243528034
20	111013	48	11989305191954
21	210871	49	23478079751570
22	401428	50	45996077217314
23	766150	51	90149014737984
24	1465020	52	176756698589169
25	2807196	53	346703854338999
26	5387991	54	680303824246455
27	10358999	55	1335372860954853
28	19945394	56	2622115606838643
29	38458184	57	5150452239739197
30	74248451	58	10119941474477832

EXAMPLE 3.3: Let $q = 2^{17}$, and let $\mathbb{F}_{2^{17}} \equiv \mathbb{F}_2[x]/(f(x))$ where $f(x)$ is the primitive polynomial $x^{17} + x^3 + 1$ in $\mathbb{F}_2[x]$ (generated by `primpoly.c`). We see that $q - 1 = 131071$ is prime and we let x be our chosen generator for the field. Our goal is to find $\log_x \beta$ where

$$\beta = x^{16} + x^{15} + x^{12} + x^{11} + x^{10} + x^9 + x^5 + x^4 + x^2. \tag{3.8}$$

We begin by choosing our factor base to be all irreducible polynomials in $\mathbb{F}_2[x]$ of degree less than 5, so we have $|\mathcal{B}| = 14$ and

$$\begin{aligned} \mathcal{B} \;=\; & \{x, x+1, x^2+x+1, x^3+x+1, x^3+x^2+1, x^4+x+1, x^4+x^3+1, \\ & x^4+x^3+x^2+x+1, x^5+x^2+1, x^5+x^3+1, x^5+x^3+x^2+x+1, \\ & x^5+x^4+x^2+x+1, x^5+x^4+x^3+x+1, x^5+x^4+x^3+x^2+1\}. \end{aligned} \tag{3.9}$$

For Stage 1, we are interested in determining the logarithms in base x of the elements in $\mathcal{B}$. Let $L_1, L_2, \ldots, L_{|\mathcal{B}|}$ be the logarithms in base x of the elements in the factor base (in the above order). Note that $L_1 = \log_x x = 1$, so we need only find the other 13 logarithms. We begin generating smooth relations by raising x to random powers $0 \leq r \leq 131070$. After doing so and checking for smoothness (actually 0.340847 seconds computation time), we arrive at the following 35 5-smooth relations:

$$
\begin{aligned}
x^{69750} &\equiv (x^4+x^3+1)(x^5+x^4+x^3+x^2+1)(x+1)^7 \mod f(x)\\
x^{4606} &\equiv (x^2+x+1)(x^4+x^3+x^2+x+1)(x+1)^2x^4 \mod f(x)\\
x^{7341} &\equiv (x^4+x^3+x^2+x+1)(x^5+x^3+1)(x^5+x^4+x^3+x+1)x^2 \mod f(x)\\
x^{38581} &\equiv (x^3+x^2+1)(x^5+x^3+1)x^4 \mod f(x)\\
x^{36862} &\equiv (x^4+x+1)x^2(x^5+x^3+1)^2 \mod f(x)\\
x^{35533} &\equiv (x^5+x^3+1)(x^3+x+1)^2 \mod f(x)\\
x^{731} &\equiv (x^3+x+1)(x^4+x^3+1)(x^5+x^4+x^2+x+1)x^2 \mod f(x)\\
x^{121864} &\equiv (x^2+x+1)(x^5+x^3+1)x^3 \mod f(x)\\
x^{70506} &\equiv (x^3+x^2+1)(x^4+x+1)(x^4+x^3+x^2+x+1)x^3 \mod f(x)\\
x^{17769} &\equiv (x^3+x+1)x^5(x+1)^2(x^2+x+1)^2 \mod f(x)\\
x^{1254} &\equiv (x^4+x+1)x^3(x+1)^2(x^3+x^2+1)^2 \mod f(x)\\
x^{127026} &\equiv x(x^4+x^3+x^2+x+1)(x^5+x^3+x^2+x+1)(x+1)^2(x^2+x+1)^2 \mod f(x)\\
x^{33530} &\equiv x^2(x^5+x^4+x^3+x^2+1)^2(x+1)^4 \mod f(x)\\
x^{37870} &\equiv x(x^5+x^3+x^2+x+1)(x^2+x+1)^2(x+1)^6 \mod f(x)\\
x^{87318} &\equiv x(x+1)(x^2+x+1)(x^5+x^3+x^2+x+1)(x^5+x^2+1) \mod f(x)\\
x^{43705} &\equiv (x^2+x+1)(x^4+x^3+1)(x+1)^5x^2 \mod f(x)\\
x^{73531} &\equiv x(x^3+x+1)(x+1)^6 \mod f(x)\\
x^{114462} &\equiv x(x^3+x+1)(x^5+x^3+1)(x^5+x^4+x^3+x+1)(x+1)^2 \mod f(x)\\
x^{12391} &\equiv (x^3+x+1)(x^4+x+1)(x^5+x^4+x^2+x+1)(x^2+x+1)^2 \mod f(x)\\
x^{73399} &\equiv (x+1)(x^2+x+1)(x^3+x^2+1)(x^4+x+1)(x^5+x^4+x^3+x^2+1) \mod f(x)\\
x^{118239} &\equiv (x^3+x^2+1)(x^5+x^4+x^2+x+1)(x^2+x+1)^3 \mod f(x)\\
x^{107594} &\equiv (x^5+x^4+x^2+x+1)x^{10} \mod f(x)\\
x^{58817} &\equiv x(x^5+x^2+1)(x^2+x+1)^3 \mod f(x)\\
x^{64781} &\equiv x(x^2+x+1)(x^5+x^3+x^2+x+1)(x^4+x+1)^2 \mod f(x)\\
x^{130655} &\equiv (x^4+x+1)x^2(x^2+x+1)^4 \mod f(x)\\
x^{130045} &\equiv (x^2+x+1)(x^4+x^3+x^2+x+1)(x^4+x^3+1)^2 \mod f(x)\\
x^{81705} &\equiv x(x^3+x^2+1)(x^4+x^3+1)(x^4+x+1) \mod f(x)\\
x^{19103} &\equiv x(x^3+x^2+1)(x^3+x+1)(x^2+x+1)^4 \mod f(x)\\
x^{43208} &\equiv (x^4+x+1)(x^4+x^3+1)x^8 \mod f(x)\\
x^{79147} &\equiv x(x^4+x^3+1)(x^3+x+1)^2(x+1)^4 \mod f(x)\\
x^{73868} &\equiv (x^3+x^2+1)(x^5+x^4+x^3+x^2+1)(x+1)^4 \mod f(x)\\
x^{29093} &\equiv (x+1)(x^2+x+1)(x^3+x^2+1)x^2(x^4+x^3+x^2+x+1)^2 \mod f(x)\\
x^{121840} &\equiv x(x+1)^3(x^2+x+1)^4 \mod f(x)\\
x^{75705} &\equiv x(x^3+x^2+1)(x+1)^4 \mod f(x)\\
x^{8869} &\equiv (x^4+x+1)^1(x^2+x+1)^3x^4 \mod f(x)
\end{aligned}
$$

Moving on to Stage 2, we now wish to take the logarithm in base x of each relation. For

example when we take $\log_x$ of the first relation, we get the congruence

$$69750 \log_x x \equiv 7 \log_x(x+1) + \log_x(x^4+x^3+1) + \log_x(x^5+x^3+x^2+x+1) \mod q-1. \tag{3.10}$$

Then, using the correspondence we mentioned earlier and the fact that $\log_x x = 1$, we now have

$$7L_2 + L_7 + L_{14} \equiv 69750 \mod 131071. \tag{3.11}$$

Doing this for each of the 35 relations we can set up the corresponding linear system $\mathbf{AL} \equiv \mathbf{b}$ mod 131071. Taking a look at the augmented coefficient matrix we have

$$[\mathbf{A} \mid \mathbf{b}] = \begin{bmatrix}
7 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 69750 \\
2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 4602 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 73416 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 38577 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 36860 \\
0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 35533 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 729 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 121861 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 70503 \\
2 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 17764 \\
2 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1251 \\
2 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 127025 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 33528 \\
6 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 37869 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 87317 \\
5 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 43703 \\
6 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 73530 \\
2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 114461 \\
0 & 2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 12391 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 73399 \\
0 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 118239 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 107584 \\
0 & 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 58816 \\
0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 64780 \\
0 & 4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 130653 \\
0 & 1 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 130045 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 81704 \\
0 & 4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 19102 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 43200 \\
4 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 79146 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 73868 \\
1 & 1 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 29091 \\
3 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 121839 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 75704 \\
0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8865
\end{bmatrix} \mod 131071$$

We need only perform Gaussian elimination once, as $q-1$ is prime. Doing so with the software package Maple, after about one second of computation time we arrive at the following solutions.

$$\begin{bmatrix} L_2 \\ L_3 \\ L_4 \\ L_5 \\ L_6 \\ L_7 \\ L_8 \\ L_9 \\ L_{10} \\ L_{11} \\ L_{12} \\ L_{13} \\ L_{14} \end{bmatrix} = \begin{bmatrix} 9300 \\ 121788 \\ 17730 \\ 38504 \\ 36714 \\ 6486 \\ 126356 \\ 86665 \\ 73 \\ 635 \\ 107584 \\ 78508 \\ 129235 \end{bmatrix}$$

We then verify that we do indeed have the correct solutions by computing $x^{L_i} \mod f(x)$ for $2 \leq i \leq 14$, and we are done with Stage 2.

Recall that our goal was to find $\log_\alpha \beta$ where β is given in (5.6), so we will choose a random exponent $0 \leq r \leq 131070$ and compute the product $x^r\beta \mod f(x)$. If the product is B-smooth, then we can take the logarithm of the congruence to get a linear equation of $\log_x \beta$ in terms of the L_i. After a few different tries (13 to be exact) we arrive at the following relation.

$$x^{195}\beta \equiv (x^3 + x + 1)(x + 1)^5(x^2 + x + 1)^2x^4 \mod f(x) \tag{3.12}$$

Then we take the logarithm of both sides of the relation modulo $q - 1$ and we have

$$195 + \log_x \beta \equiv L_4 + 5L_2 + 2L_3 + 4 \mod 131071 \tag{3.13}$$

or

$$\log_x \beta = 17730 + 5 \cdot 9300 + 2 \cdot 121788 - 191 \equiv 45473 \mod 131071. \tag{3.14}$$

We check to ensure that $x^{45473} \equiv \beta \mod f(x)$. ▲

The generation of smooth relations in the above example and placing them into matrix form was accomplished by the routine `indcal.c`, and can be found in the appendix. The routine incorporates many subroutines, as well as many built-in functions of the library NTL. This routine is also used in comparisons, which we will discuss later.

For this field of over 100,000 elements, it took almost no computation time at all to determine the logarithms of the factor base $\mathcal{B}$. This illustrates the power of the index calculus method somewhat better than example 4.1 does. We also see a decided advantage over previous algorithms: once the first two stages are complete, the entire field can essentially be deemed insecure for crytpographic purposes. This is because the running time of the third stage is small compared to the first two stages and the first two stages in no way depend on the element β. Thus, we can attempt to increase the performance of the index calculus method even further by improving stages one and two. For the remainder of this paper, we will look at various improvements that affect Stages 1 and 2 of the algorithm.

4 Improving the Index Calculus Method

4.1 The Search for Smooth Polynomials

Our quest for improving Stage 1 of the Index Calculus method begins with finding ways to get B-smooth relations more efficiently. First, we will present some results regarding the probability that a random polynomial in $\mathbb{F}_2[x]$ of degree k is B-smooth. Let $p(k, B)$ denote this probability. From [42], we have that

$$p(k, B) = \frac{N(k, B)}{N(k, k)} = \frac{N(k, B)}{2^k} \tag{4.1}$$

where $N(k, B)$ denotes the number of polynomials in $\mathbb{F}_2[x]$ of degree k that are B-smooth. Trivially we have $N(k, k) = 2^k$, as there are 2^k polynomials of degree k, and they are all k-smooth. In the general index calculus method, we are essentially generating random polynomials of degrees less than n and hoping that they are smooth. The probability that a random polynomial of degree less than n is B-smooth is given by

$$\frac{\sum_{k<n} N(k, B)}{\sum_{k<n} N(k, k)} = \frac{\sum_{k<n} N(k, B)}{2^n - 1}. \tag{4.2}$$

Odlyzko [42] derives recurrences that $N(k, B)$ satisfies, and determines the estimate for the probability (6.2) as

$$\exp\left((1 + o(1))\frac{k}{B}\log\frac{B}{k}\right) \tag{4.3}$$

where $k^{1/100} \leq B \leq k^{99/100}$. Tables of numerical estimates of these probabilities can be found appended to [42]. These probabilities are, in general, not very large, for example, we have $p(100, 10) = 1.71395 \times 10^{-10}$. Thus we would expect to have to factor around six billion polynomials (in $\mathbb{F}_2[x]$) of degree 100 before we would find one that factored completely over a factor base that contained all irreducibles of degrees less than or equal to 10. Hence care must be taken in deciding on the size of the factor base, as well as the types of polynomials to test for smoothness. [5] and [43] provide further discussions on smooth polynomials and their distributions.

4.2 Coppersmith's Method

In 1984 D. Coppersmith [11] developed a variant of the index calculus method that improved the asymptotic running time of the algorithm in $\mathbb{F}_{2^n}$. Instead of raising x to random power

which will be congruent to a polynomial of degree less than n (which about half of the time will be a polynomial of degree $n-1$), Coppersmith sought a different way to generate equivalences that would force the congruent polynomials to be of smaller degrees, thus increasing their respective probabilities of smoothness. Working with ideas of generating 'systematic equations' that were introduced by Blake, et al. [8], and the property that squaring is a linear operator in $\mathbb{F}_2$ ($(a+b)^2 = a^2 + b^2$), Coppersmith developed this method for finding smooth relations. Assume that our field $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f(x))$ where $f(x) = x^n + f_1(x)$ is primitive and $\deg(f_1)$ is small. Let $u_1(x), u_2(x)$ be two random polynomials in $\mathbb{F}_2[x]$ of degrees less some specified degree d (we will determine this parameter later) such that $\gcd(u_1(x), u_2(x)) = 1$ (u_1 and u_2 are relatively prime). This condition can be checked easily by applying the Euclidean algorithm for polynomials. Then we define, for some integer h, the polynomial

$$w_1(x) = u_1(x)x^h + u_2(x). \tag{4.4}$$

Then we have, for any integer 2^k,

$$\begin{aligned}
(w_1(x))^{2^k} &\equiv (u_1(x)x^h + u_2(x))^{2^k} \mod f(x) \\
&\equiv u_1(x)^{2^k} x^{h2^k} + u_2(x)^{2^k} \mod f(x) \\
&\equiv u_1(x)^{2^k} f_1(x) x^{h2^k - n} + u_2(x)^{2^k} \mod f(x)
\end{aligned} \tag{4.5}$$

and we will let $w_2(x)$ denote the right-hand side of (4.5). Thus we have the congruence

$$w_2(x) \equiv w_1(x)^{2^k} \mod f(x). \tag{4.6}$$

The coprime condition imposed on the $u_1(x)$ and $u_2(x)$ polynomials prevents us from arriving at redundant relations, for if $\gcd(u_1(x), u_2(x)) = e(x)$ for some nontrivial $e(x)$, then $e(x)^{2^k}$ is a nontrivial factor of both $w_1(x)^{2^k}$ and $w_2(x)$, and thus we would get the same relation as we would by choosing $u_1(x)/e(x)$ and $u_2(x)/e(x)$ as our random seeds.

The parameters $h, 2^k, d$, and B are chosen so that the degrees of w_1 and w_2 are on the order of $n^{2/3}$ and the degree of our factor base is on the order of $n^{1/3}$. Derivation of these parameters can be found in [42] and [14], and they are defined as follows:

$$\begin{aligned}
d &\approx n^{\frac{1}{3}} (\log n)^{\frac{2}{3}} \\
2^k &\approx \left(\frac{n}{\log n} \right)^{\frac{1}{3}} \\
h &= \left\lfloor \frac{n}{2^k} \right\rfloor + 1 \\
B &\approx n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}.
\end{aligned}$$

Extensive tables of these parameters can be found in [14]. We will now examine briefly what these polynomials look like with a specific case.

EXAMPLE 4.1: Let $q = 2^{25}$, and let $\mathbb{F}_q \cong \mathbb{F}_2[x]/(f(x))$ where $f(x) = x^{25} + x^3 + 1$. Then $B = d = 7$, $h = 13$, and $2^k = 2$. We start by picking random $u_1(x)$ and $u_2(x)$ and checking to see if $\gcd(u_1, u_2) = 1$. We choose $u_1(x) = x^5 + x^4 + x^3 + x + 1$ and $u_2(x) = x^6 + x^3 + 1$. Applying the Euclidean algorithm shows us that $\gcd(x^5 + x^4 + x^3 + x + 1, x^6 + x^3 + 1) = 1$, so we can use these to define our $w_1(x)$ and $w_2(x)$. Now we construct

$$\begin{aligned} w_1(x) &= u_1(x)x^h + u_2(x) \\ &= (x^5 + x^4 + x^3 + x + 1)x^{13} + (x^6 + x^3 + 1) \\ &= x^{18} + x^{17} + x^{16} + x^{14} + x^{13} + x^6 + x^3 + 1 \end{aligned}$$

and since we will want $w_1(x)^{2^k} \mod f(x)$, we have

$$\begin{aligned} w_1(x)^{2^k} &= x^{36} + x^{34} + x^{32} + x^{28} + x^{26} + x^{12} + x^6 + 1 \\ &\equiv x^{14} + x^{11} + x^{10} + x^9 + x^7 + x^4 + x^3 + x + 1 \mod f(x). \end{aligned}$$

Then we create the $w_2(x)$ polynomial, and we see that

$$\begin{aligned} w_2(x) &= u_1(x)^{2^k} f_1(x) x^{h2^k - n} + u_2(x)^{2^k} \\ &= (x^5 + x^4 + x^3 + x + 1)^2 x^{2 \cdot 13 - 25}(x^3 + 1) + (x^6 + x^3 + 1)^2 \\ &= (x^{11} + x^9 + x^7 + x^3 + x)(x^3 + 1) + (x^{12} + x^6 + 1) \\ &= x^{14} + x^{11} + x^{10} + x^9 + x^7 + x^4 + x^3 + x + 1 \\ &\equiv w_1(x)^{2^k} \mod f(x). \end{aligned}$$

Hence we have a relation of polynomials in $\mathbb{F}_2[x]$ modulo $f(x)$. ▲

With the choices of parameters defined above, Coppersmith's variant has a heuristic (not rigorously proven) running time of

$$\exp\left((1 + o(1))n^{1/3} \log^{2/3} n\right) \tag{4.7}$$

which we see is a significant improvement over the time of $L(q)^{2+o(1)}$ for the general method. We will illustrate how Coppersmith's method works with an example, this one for a field of size a little larger than the previous example of index calculus in a nonprime field, and we will only complete Stages 1 and 2.

EXAMPLE 4.2: Let $q = 2^{19}$, and let $\mathbb{F}_q \cong \mathbb{F}_2[x]/(f(x))$ where $f(x)$ is the primitive polynomial $x^{19}+x^5+x^2+x+1$. By the definition of the Coppersmith parameters, we have that $B = d = 6$, $h = 10$, and $2^k = 2$. So our factor base $\mathcal{B}$ consists of all irreducibles in $\mathbb{F}_2[x]$ of degrees up to 6. We have already (in example 5.2) noted the irreducibles of degrees less than 6, the ones of degree 6 are $x^6 + x + 1, x^6 + x^3 + 1, x^6 + x^4 + x^2 + x + 1, x^6 + x^4 + x^3 + x + 1, x^6 + x^5 +$

$1, x^6+x^5+x^2+x+1, x^6+x^5+x^3+x^2+1, x^6+x^5+x^4+x+1$, and $x^6+x^5+x^4+x^2+1$. We begin finding smooth relations by choosing random polynomials of degrees less than or equal to d to be our $u_1(x)$ and $u_2(x)$. The C++ routine `copper.c` (see appendix) was used to accomplish this, and after about six seconds of computation time we arrive at the following 55 6-smooth relations (we omit the $\mod f(x)$ for space):

$$
\begin{aligned}
x(x^5+x^3+x^2+x+1)(x^6+x+1)(x+1)^2 &\equiv x^2(x^3+x^2+1)^2(x^4+x^3+x^2+x+1)^2(x^6+x^4+x^3+x+1)^2\\
x^2(x^3+x+1)^2(x^5+x^3+1)^2(x+1)^6 &\equiv x(x^4+x^3+x^2+x+1)(x^5+x^4+x^3+x+1)\\
x^{20} &\equiv x(x^3+x+1)(x+1)^2\\
x^2(x^6+x+1)^2(x^6+x^4+x^3+x+1)^2 &\equiv x(x^6+x^5+x^3+x^2+1)(x+1)^2\\
(x^5+x^4+x^3+x^2+1)^2(x^6+x^5+x^2+x+1)^2 &\equiv (x^3+x+1)(x^6+x^5+x^4+x^2+1)(x+1)^3\\
(x^5+x^4+x^3+x^2+1)^2(x^6+x^5+x^2+x+1)^2 &\equiv (x^3+x+1)(x^6+x^5+x^4+x^2+1)(x+1)^3\\
(x^4+x^3+1)^2(x^4+x^3+x^2+x+1)^2x^4 &\equiv x(x^2+x+1)(x^6+x^4+x^3+x+1)(x+1)^3\\
(x^5+x^4+x^2+x+1)^2(x^5+x^4+x^3+x+1)^2 &\equiv (x^2+x+1)(x^3+x^2+1)(x^4+x^3+x^2+x+1)(x+1)^3\\
(x^3+x+1)^2(x^5+x^3+x^2+x+1)^2x^4 &\equiv x(x^6+x^5+1)(x+1)^3\\
(x+1)^2(x^2+x+1)^2(x^4+x^3+1)^2x^{12} &\equiv x(x^4+x+1)(x^5+x^4+x^3+x^2+1)\\
x^2(x^5+x^2+1)^2(x^5+x^4+x^2+x+1)^2 &\equiv x(x^2+x+1)(x^3+x+1)\\
(x^5+x^3+x^2+x+1)^2(x^6+x^5+x^4+x^2+1)^2 &\equiv (x^3+x^2+1)(x^4+x+1)(x^5+x^4+x^3+x^2+1)\\
(x+1)^2(x^5+x^2+1)^2(x^6+x^5+x^4+x+1)^2 &\equiv (x^2+x+1)(x^5+x^3+x^2+x+1)(x^5+x^4+x^3+x^2+1)\\
(x^3+x^2+1)^2(x^4+x+1)^2(x+1)^6 &\equiv (x^2+x+1)(x^5+x^2+1)(x^5+x^4+x^3+x^2+1)\\
x^2(x^2+x+1)^2(x^5+x^4+x^3+x+1)^2(x^5+x^2+1)^2 &\equiv x(x^3+x^2+1)(x^6+x^5+1)(x+1)^2\\
(x^3+x+1)^8 &\equiv (x^2+x+1)(x^6+x^5+x^2+x+1)(x+1)^2\\
(x^3+x^2+1)^2(x^4+x^3+1)^2(x^5+x^3+x^2+x+1)^2 &\equiv (x^2+x+1)(x^5+x^3+1)(x+1)^3\\
(x^3+x+1)^2(x^4+x+1)^2(x^5+x^4+x^2+x+1)^2 &\equiv (x^5+x^3+x^2+x+1)(x+1)^2\\
(x^2+x+1)^2(x^4+x+1)^2(x^5+x^2+1)^2(x+1)^4 &\equiv (x^5+x^4+x^2+x+1)(x^5+x^3+1)\\
(x^4+x^3+x^2+x+1)^2(x^5+x^4+x^3+x^2+1)^2x^6 &\equiv x(x^2+x+1)(x^3+x+1)(x^6+x^5+x^2+x+1)\\
x^2(x^3+x^2+1)^2(x^6+x^5+x^4+x^2+1)^2 &\equiv x(x^4+x^3+x^2+x+1)(x+1)^3\\
x^2(x^3+x^2+1)^2(x^6+x^5+x^4+x+1)^2 &\equiv x(x^2+x+1)(x^3+x+1)(x^4+x^3+1)(x+1)^2\\
x^2(x^3+x+1)^2(x^3+x^2+1)^2(x^4+x^3+x^2+x+1)^2(x+1)^8 &\equiv x(x^4+x+1)(x^5+x^4+x^3+x^2+1)(x^6+x^5+x^2+x+1)\\
(x+1)^4(x^4+x^3+1)^4 &\equiv (x^3+x^2+1)(x^3+x+1)(x^6+x^5+1)\\
x^4(x+1)^4(x^6+x^5+x^3+x^2+1)^4 &\equiv x(x^3+x^2+1)(x^4+x+1)(x^5+x^2+1)(x^5+x^4+x^3+x^2+1)\\
(x^4+x+1)^2(x^5+x^4+x^2+x+1)^2x^4(x^2+x+1)^4 &\equiv x(x^3+x+1)(x^4+x^3+x^2+x+1)(x^5+x^3+1)(x+1)^3\\
x^4(x+1)^4(x^3+x^2+1)^4 &\equiv x(x^5+x^4+x^3+x+1)(x^6+x^5+x^3+x^2+1)\\
(x^3+x+1)^2(x^5+x^3+1)^2(x^5+x^3+x^2+x+1)^2 &\equiv (x^2+x+1)(x^3+x^2+1)(x^5+x^4+x^2+x+1)\\
(x^2+x+1)^2(x^3+x+1)^2(x+1)^6x^4 &\equiv x(x^3+x^2+1)(x^4+x^3+1)(x^4+x+1)\\
x^2(x^4+x+1)^2(x^2+x+1)^6 &\equiv x(x^4+x^3+1)\\
(x^2+x+1)^2(x^6+x^4+x^3+x+1)^2x^6 &\equiv x(x^3+x^2+1)(x^4+x^3+x^2+x+1)\\
(x^5+x^4+x^3+x^2+1)^2(x+1)^{10} &\equiv (x^2+x+1)(x^3+x^2+1)(x^5+x^3+1)\\
x^2(x+1)^2(x^2+x+1)^2(x^6+x^3+1)^2 &\equiv x(x^5+x^2+1)\\
(x^4+x^3+1)^8 &\equiv (x^6+x^5+x^4+x^2+1)(x^6+x^5+x^3+x^2+1)(x^6+x^3+1)\\
(x+1)^2(x^6+x^5+1)^2(x^2+x+1)^4 &\equiv (x^6+x^5+x^3+x^2+1)\\
(x^4+x+1)^2(x^6+x^4+x^2+x+1)^2x^6 &\equiv x(x^5+x^4+x^3+x^2+1)(x+1)^3\\
(x^3+x+1)^2(x^4+x^3+1)^2(x^6+x^4+x^2+x+1)^2 &\equiv (x^2+x+1)(x^5+x^4+x^3+x+1)(x+1)^3\\
(x^3+x^2+1)^2(x^4+x^3+1)^2(x^4+x+1)^2x^6 &\equiv x(x^5+x^4+x^3+x+1)(x^6+x^5+1)(x+1)^2\\
x^2(x^3+x+1)^2(x^4+x^3+1)^2(x^6+x^5+x^2+x+1)^2 &\equiv x(x^5+x^4+x^3+x^2+1)(x^5+x^3+x^2+x+1)(x+1)^3\\
(x^4+x+1)^2x^{10}(x^3+x^2+1)^4 &\equiv x(x^2+x+1)(x^4+x^3+1)(x^6+x^5+x^2+x+1)(x+1)^3\\
x^2(x+1)^2(x^3+x^2+1)^2(x^5+x^4+x^2+x+1)^4 &\equiv x(x^2+x+1)(x^4+x^3+x^2+x+1)(x^4+x+1)(x^5+x^3+x^2+x+1)
\end{aligned}
$$

$$
\begin{aligned}
(x^3+x+1)^2(x^4+x+1)^2(x^4+x^3+1)^2x^4 &\equiv x(x^4+x^3+x^2+x+1)(x^5+x^2+1)\\
(x^5+x^4+x^3+x+1)^2x^6(x+1)^8 &\equiv x(x^3+x+1)(x^3+x^2+1)(x^5+x^4+x^3+x^2+1)\\
(x^4+x^3+1)^2(x^6+x+1)^2(x+1)^6 &\equiv (x^2+x+1)(x^3+x^2+1)(x^4+x^3+x^2+x+1)\\
(x+1)^4(x^4+x^3+x^2+x+1)^4 &\equiv (x^2+x+1)(x^4+x^3+1)\\
x^2(x^6+x^5+1)^2(x^3+x^2+1)^4 &\equiv x(x^3+x+1)(x^5+x^3+x^2+x+1)\\
x^2(x^5+x^3+1)^2(x^2+x+1)^4 &\equiv x(x^3+x+1)(x^3+x^2+1)(x+1)^3\\
x^2(x^5+x^3+1)^2(x^2+x+1)^4 &\equiv x(x^3+x^2+1)(x^3+x+1)(x+1)^3\\
(x^4+x^3+x^2+x+1)^2(x+1)^6(x^2+x+1)^4 &\equiv (x^6+x+1)(x^6+x^4+x^2+x+1)\\
(x^4+x^3+x^2+x+1)^2(x^5+x^2+1)^2x^6 &\equiv x(x^6+x^5+x^3+x^2+1)(x+1)^3\\
x^2(x^2+x+1)^2(x^4+x^3+x^2+x+1)^2(x^4+x^3+1)^4 &\equiv x(x^3+x+1)(x^4+x+1)(x^5+x^4+x^3+x+1)(x+1)^3\\
x^2(x^6+x^5+x^2+x+1)^2(x+1)^6 &\equiv x(x^2+x+1)(x^5+x^4+x^2+x+1)\\
x^2(x^2+x+1)^2(x^3+x+1)^2(x^4+x^3+x^2+x+1)^2 &\equiv x(x^5+x^3+1)(x+1)^2\\
(x^3+x+1)^4x^8 &\equiv x(x^4+x^3+x^2+x+1)(x^5+x^4+x^3+x^2+1)(x+1)^2\\
(x^5+x^3+x^2+x+1)^2(x^6+x^5+x^4+x^2+1)^2 &\equiv (x^3+x^2+1)(x^4+x+1)(x^5+x^4+x^3+x^2+1)
\end{aligned}
$$

As `copper.c` executes, once it finds a smooth relation, it 'takes the logarithm' of the relation by recording the exponents and places the proper values in a row vector, with $\log_x x$ on the right-hand

side (last column). We then wish to solve the corresponding linear system modulo 524287.

$$\begin{bmatrix}
-2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -1 \\
6 & 0 & 2 & 0 & 0 & 0 & -1 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
-2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -19 \\
-2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & -1 \\
-3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 & 0 \\
-3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 & 0 \\
-3 & -1 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -3 \\
-3 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-3 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -3 \\
2 & 2 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -11 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\
2 & -1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
6 & -1 & 0 & 2 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2 & 2 & 0 & -1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\
-2 & -1 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
-3 & -1 & 0 & 2 & 0 & 2 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -5 \\
-3 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 \\
-2 & -1 & -1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & -1 \\
8 & 0 & 2 & 2 & -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 \\
4 & 0 & -1 & -1 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & -3 \\
-3 & 4 & -1 & 0 & 2 & 0 & -1 & 0 & -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 \\
4 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -3 \\
0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 2 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 2 & 2 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 \\
0 & 6 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 2 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -5 \\
10 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\
2 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & -1 & 0 & 0 & 0 \\
-3 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\
-3 & -1 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2 & 0 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -5 \\
-3 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & -1 \\
-3 & -1 & 0 & 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -9 \\
2 & -1 & 0 & 2 & -1 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 2 & 0 & 2 & 2 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 \\
8 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\
6 & -1 & 0 & -1 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & -1 & 0 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\
-3 & 4 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
-3 & 4 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
6 & 4 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-3 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -5 \\
-3 & 2 & -1 & 0 & -1 & 4 & 2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
6 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\
-2 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
-2 & 0 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -7 \\
0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0
\end{bmatrix}$$

After about a second and a half of running time in Maple, we get unique solutions for the logarithms of the factor base

$$\begin{aligned}
L_2 &= 264794 \quad L_3 = 209140 \quad L_4 = 519005 \quad L_5 = 107199 \\
L_6 &= 211481 \quad L_7 = 104942 \quad L_8 = 75870 \quad L_9 = 22128 \\
L_{10} &= 29869 \quad L_{11} = 42411 \quad L_{12} = 341944 \quad L_{13} = 513495 \\
L_{14} &= 421995 \quad L_{15} = 83211 \quad L_{16} = 323560 \quad L_{17} = 396705 \\
L_{18} &= 406679 \quad L_{19} = 328453 \quad L_{20} = 267590 \quad L_{21} = 450193 \\
L_{22} &= 49851 \quad L_{23} = 65783
\end{aligned}$$

We perform a check to see that these are the correct logarithms of the elements of $\mathcal{B}$. ▲

Coppersmith also gives an improvement on the third stage that reduces the running time of that particular stage. In brief, he suggests that we compute $\log_x \beta$ by first forming a product that may include one prime factor not in the factor base, and then computing the logarithms of a sequence of such polynomials of decreasing degrees. A description of this approach can be found in [14].

One important assumption in the running time analysis is that the Coppersmith polynomials will behave like random independent polynomials of the same degrees. If this is so, then these Coppersmith polynomials should have a larger probability of being smooth over ones generated by the basic index calculus method as the degrees of the Coppersmith polynomials are generally smaller than those generated by the basic approach. Cuneaz [14] found that, for a majority of $n \leq 300$, there was sufficient statistical evidence to support the conclusion that the proportion of w_1 polynomials that are B-smooth is greater than the proportion of random polynomials of the same degree being B-smooth. However, there was also sufficient statistical evidence to conclude that the proportion of w_2 polynomials that are B-smooth is less than the proportion of random polynomials that are B-smooth. Cuneaz was also able to deduce that there did not exist sufficient statistical evidence to conclude that the Coppersmith polynomials behaved as dependent polynomials, which gives us a significant reason to think that Coppersmith's method is a drastic improvement over the original algorithm.

It is also worthwhile to point out that when Coppersmith first developed this algorithm in 1984, he focused on the case $q = 2^{127}$. This field was actually intended for use in cryptosystems being developed by Hewlett-Packard and Mitre at the time, but Coppersmith's successful computation of the logarithms of the factor base for this field have rendered it totally insecure for cryptographic purposes. In 1985 Odlyzko [42] conjectured that logarithms for fields of order 2^n for n up to 520 would be feasible to compute with a supercomputer. It seems that this bound should be much larger today with twelve years of technological advancements.

4.3 Semaev's Method

In a 1994 I. A. Semaev [52] presented two different ways to arrive at polynomial relations that, heuristically, improved the running time of the first stage of the index calculus method for certain fields, including $\mathbb{F}_{2^n}$ for some specific n. We will present only one of these at present. The other involves linear combinations of Dickson polynomials, and a good description of this version can be found in [18].

For the type of Semaev polynomials we are interested in, we will form congruences, similar to congruences of Coppersmith's method, of the form

$$C(x)^{2^k} \equiv D(x) \mod f(x) \tag{4.8}$$

where $f(x)$ is the monic irreducible polynomial used to define the field. These congruences will be applicable in $\mathbb{F}_{2^n}$ when $2^n - 1$ has a small primitive factor r (i.e., $r \mid (2^n - 1), r \nmid (2^k - 1)$ for $k < n$). Now these conditions are satisfied if $r = n + 1$ is prime and 2 is a primitive

element in $\mathbb{Z}_r$ (note that these conditions are not necessarily equivalent to $2^n - 1$ having a small primitive factor). E. Artin conjectured that there are infinitely many such values of n. The following is a listing of $n < 1000$ that satisfy $r = n + 1$ is prime and 2 is a primitive element in $\mathbb{Z}_r$. For such an n, the polynomial $f(x) = x^n + x^{n-1} + \cdots + x + 1$ is irreducible.

Table 8: Some $n < 1000$ that satisfy Semaev's conditions

2	4	10	12	18	28	36	52	58	60	66	82
100	106	130	138	148	162	172	178	180	196	210	226
268	292	316	346	348	372	378	388	418	420	442	460
466	490	508	522	540	546	556	562	586	612	618	652
658	660	676	700	708	756	772	786	796	820	826	828
852	858	876	882	906	940	946					

Now for a given n that satisfies these conditions, let $u = 2^k \mod r$ be an integer, and let $m \sim n^{2/3}$. Define the ordered sets

$$T_u = \{0 \leq \ell < r : \ell < m, (\ell u \mod r) \leq m\} \tag{4.9}$$

$$T_{i_u} = \{ui \mod r : i \in T_u\}. \tag{4.10}$$

Let $\omega \in \mathbb{F}_{2^n}$ be a primitive r-th root of unity. Now let $C = \sum_{i \in T_u} a_i \omega^i$ for $a_i \in \mathbb{F}_2$, and set $i_u = ui \mod r$. Then, if $D = \sum_{i \in T_u} a_i \omega^{i_u}$, we have

$$\begin{aligned} C^u &= \left(\sum_{i \in T_u} a_i \omega^i \right)^u \\ &= \sum_{i \in T_u} a_i \omega^{iu} \\ &= \sum_{i \in T_u} a_i \omega^{i_u} \\ &= D. \end{aligned} \tag{4.11}$$

Letting $C(x)$ and $D(x)$ be the corresponding polynomials in $\mathbb{F}_2[x]$, we now have the relation

$$C(x)^{2^k} \equiv D(x) \mod f(x) \tag{4.12}$$

where $f(x)$ is the $n+1$-st cyclotomic polynomial in $\mathbb{F}_2[x]$. Now $C(x)$ and $D(x)$ are of degree at most m, and thus will be more likely to be B-smooth than random polynomials of degrees less than n (B is defined as before in the Coppersmith case). Now the number of such pairs we have is $2^{|T_u|}$, as there are two choices for each a_i. Semaev goes on to show that for

Table 9: T_u and T_{i_u} for selected n, u

n	m	u	T_u	T_{i_u}
18	10	2	$\{0,1,2,3,4,5,10\}$	$\{0,2,4,6,8,10,1\}$
18	10	4	$\{0,1,2,5,6,7,10\}$	$\{0,4,8,1,5,9,2\}$
18	10	8	$\{0,1,3,5,6,8,10\}$	$\{0,8,5,2,10,7,4\}$
28	13	2	$\{0,1,2,3,4,5,6\}$	$\{0,2,4,6,8,10,12\}$
28	13	4	$\{0,1,2,3,8,9,10\}$	$\{0,4,8,12,3,7,11\}$
28	13	8	$\{0,1,4,5,8,11,12\}$	$\{0,8,3,11,6,1,9\}$
60	24	2	$\{0,1,2,3,4,5,6,7,8,9,10,11,12\}$	$\{0,2,4,6,8,10,12,14,16,18,20,22,24\}$
60	24	4	$\{0,1,2,3,4,5,6,16,17,18,19,20,21\}$	$\{0,4,8,12,16,20,24,3,7,11,15,19,23\}$
60	24	8	$\{0,1,2,3,8,9,10,16,17,18,23,24\}$	$\{0,8,16,24,3,11,19,6,14,22,1,9\}$
100	35	2	$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17\}$	$\{0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34\}$
100	35	4	$\{0,1,2,3,4,5,6,7,8,26,27,28,29,30,31,32,33,34\}$	$\{0,4,8,12,16,20,24,28,32,3,7,11,15,19,23,27,31,35\}$
100	35	8	$\{0,1,2,3,4,13,14,15,16,17,26,27,28,29\}$	$\{0,8,16,24,32,3,11,19,27,35,6,14,22,30\}$
100	35	16	$\{0,1,2,7,8,13,14,19,20,21,26,27,32,33\}$	$\{0,16,32,11,27,6,22,1,17,33,12,28,7,23\}$

almost all choices of u, $|T_u| = m^2/r(1+o(1))$. Now if these polynomials behave like random independent polynomials, then we should obtain a sufficient number of smooth relations for Stage 1. Table 9 is a brief list of the ordered sets T_u and $T_{iu} = T_{i_u}$ for selected n and u.

However, some things have changed. Note that x is a primitive r-th root of unity, and thus we have $x^{n+1} - 1 = 0$. Hence our x is no longer a primitive element in the field. So we must find a different element of our factor base to serve as our logarithmic base. We can do so by applying the primitive element test to $x+1$, x^2+x+1, and so on until we do find one that survives the test.

Implementation of Semaev's method in Stage 1 of the index calculus method would proceed as follows:

1. Determine the size of the factor base and a primitive element in the factor base.
2. Select an integer $u = 2^k$ such that T_u and T_{i_u} are large.
3. For all choices of the a_i, construct and factor $C(x)$ and $D(x)$, check to see if both are B-smooth, and add the relation to the collection if so. Stop when $2|\mathcal{B}|$ relations are obtained or all choices for the a_i have been exhausted.
4. If there are not $2|\mathcal{B}|$ relations, go to step 2 and select a different k.

An NTL routine that uses this method had not been constructed at the time of writing, and it is yet to be seen whether the use of these relations can improve the performance of stage one. One potential downfall of this approach is that it is applicable only in certain fields. Some future research will focus on the applicability of this approach to other fields, as well as ways to choose u such that T_u is large.

4.4 Other Improvements and Techniques

There are other improvements to the traditional index calculus method, including one in particular due to Adleman [2] that implements a function field sieve. Also, there are many

different techniques [21] that one can implement into Coppersmith's or Semaev's variants. These include forcing a small degree factor into the w_1 and w_2 Coppersmith polynomials to increase the probability of smoothness, using equations that involve a prime factor of degree slightly larger than B, and testing a polynomial for smoothness before performing the actual factorization. There are at least two known methods for applying the latter technique, they are outlined in [42]. One of these, found by Coppersmith, tests a polynomial $h(x)$ for B-smoothness by computing

$$h'(x) \prod_{i=\lceil B/2 \rceil}^{B} (x^{2^i} + x) \mod h(x). \tag{4.13}$$

and checking whether the resulting polynomial is zero or not. This method can fail occasionally (as mentioned in [42]) but is still of use as it can reduce the number of unsuccessful factorizations we must compute.

5 Polynomial Sieving

This leads us to consider the question of being able to 'predict' if a polynomial of a given form will be smooth or not, even before we know the specific polynomial itself. Along the same lines, we would like a method, preferably simpler and faster than factoring, to determine which polynomials are factors of a polynomial of a given form

5.1 The Sieving Process

Before we attempt to answer some of these questions, we will again stop and digress a little to describe the ideas we will implement in this section. A peek in Webster's gives the definition of a *sieve* as

> a device with meshes or perforations through which finer particles of a mixture (as of ashes, flour, or sand) of various sizes are passed to separate them from coarser ones, through which the liquid is drained from liquid-containing material, or through which soft materials are forced for reduction to fine particles.

At first the above passage seems hardly mathematical, one is likely to conjure up thoughts of a prospector panning away for gold in the local creek. But implicit in the definition is an important idea: look at a large number or space of something that you are interested in and figure out a way to easily select the elements you want without having to pick through everything piece by piece. The first (and probably most well-known) application of the sieve idea in mathematics was the *Sieve of Eratosthenes* [44]. The sieve of Eratosthenes was developed to find prime numbers, and it uses a simple procedure to do so. The numbers

from 2 to, say 100, are written down in order. Then one strikes through the number 2, then 4, 6, 8, and all multiples of 2 in the list. We do the same for 3 and its multiples, 5, 7 and so on until we get to a prime that is larger than the square root of the last number in our list. Upon examination of the list now, the numbers that have not been struck through are the primes less than the last number that were not used to sieve with.

We are interested in using a similar approach, but we want different results. We wish to find elements that are smooth (factor completely over a small set), or that have a sufficient number of small factors. Let us illustrate what we are looking for with the following example, which is presented with a computational flavor.

EXAMPLE 5.1: Consider the integers from 41308 to 41349. We wish to determine which of these integers will factor smoothly among primes less than 20. To begin, we initialize an array with 0's for each of these 45 numbers, the first position corresponding to 41308, the second to 41309, and so on. The figure below could be a possible computer representation of this array. Note that the actual numbers (above the lines) would not be stored.

41308	41309	41310	41311	41312	41313	41314	41315	41316	41317	41318	41319	41320	41321
41322	41323	41324	41325	41326	41327	41328	41329	41330	41331	41332	41333	41334	41335
41336	41337	41338	41339	41340	41341	41342	41343	41344	41345	41346	41347	41348	41349

It's pretty easy to see that the first integer in the list is divisible by 2. So now we will mark off every second number in our array by placing a 2 in each number's respective position.

41308	41309	41310	41311	41312	41313	41314	41315	41316	41317	41318	41319	41320	41321
2		2		2		2		2		2		2	
41322	41323	41324	41325	41326	41327	41328	41329	41330	41331	41332	41333	41334	41335
2		2		2		2		2		2		2	
41336	41337	41338	41339	41340	41341	41342	41343	41344	41345	41346	41347	41348	41349
2		2		2		2		2		2		2	

Now, by inspection, we see that 41310 is divisible by 3 (using that old trick of summing the digits) so we mark off the positions that are divisible by three by simply starting at 41310 and pausing at every third position to place a 3. If a position already has another factor, we simply multiply the factors together.

41308	41309	41310	41311	41312	41313	41314	41315	41316	41317	41318	41319	41320	41321
2		2·3		2	3	2		2·3		2	3	2	
41322	41323	41324	41325	41326	41327	41328	41329	41330	41331	41332	41333	41334	41335
2·3		2	3	2		2·3		2	3	2		2·3	
41336	41337	41338	41339	41340	41341	41342	41343	41344	41345	41346	41347	41348	41349
2	3	2		2·3		2	3	2		2·3		2	3

We continue in this fashion for each prime p up to 19. For each p, note that we only need to find one number in the array that is divisible by p, for then we can just step along the array and mark every pth place. As we go along, we will compute the products of the primes placed in our array. Doing so, we arrive at the following array.

41308	41309	41310	41311	41312	41313	41314	41315	41316	41317	41318	41319	41320	41321
2		510		2	3	182	5	66		2	3	10	7
41322	41323	41324	41325	41326	41327	41328	41329	41330	41331	41332	41333	41334	41335
6		2	285	2	2431	42		10	3	2		6	35
41336	41337	41338	41339	41340	41341	41342	41343	41344	41345	41346	41347	41348	41349
2	3	22		390		14	3	646	5	6		2	231

Now, we will select those integers j whose positions contain values greater than $\sqrt{j}$. This value was chosen to 'weed out' the j that do not appear to be 19-smooth. Thus the integers we select to test for 19-smoothness are 41310, 41325, 41327, 41340, 41344, and 41349. We now factor these integers, and we have

$$\begin{aligned} 41310 &= 2 \cdot 3^5 \cdot 5 \cdot 17 \\ 41325 &= 3 \cdot 5^2 \cdot 19 \cdot 29 \\ 41327 &= 11 \cdot 13 \cdot 17^2 \\ 41340 &= 2^2 \cdot 3 \cdot 5 \cdot 13 \cdot 53 \\ 41344 &= 2^7 \cdot 17 \cdot 19 \\ 41349 &= 3 \cdot 7 \cdot 11 \cdot 179 \end{aligned}$$

So we see that the integers 41310, 41327, and 41344 are 19-smooth. ▲

The approach used in the above example is due to Pomerance, and it describes the type of sieving we want to do, only with polynomials instead of integers. This sieve has its good aspects as well as bad. Undesirable is the fact that we selected and factored three integers that were not smooth, and (even though we did find all in this particular range) the possibility of overlooking a 19-smooth integer exists. On the contrary, the benefits are numerous. First we should note that we did find some smooth integers, so our method was fruitful. Also, the number and type of operations necessary to do this were simple machine operations (multiplications and index incrementing), aside from a few divisibility checks. In doing this we avoid the expensive factoring of each integer, and thus our computation time was reduced considerably.

Thus our motivation for using a sieve to find smooth polynomial relations is clear: we want to determine which individual polynomials have a greater probability of being smooth and thus taking part in a smooth relation. Once we can find these polynomials, we will factor those (and only those) in efforts to make Stage 1 of the index calculus method faster.

5.2 Sieving with Coppersmith's Method

5.2.1 Approach

In their paper presented at the Crypto '92 conference, Gordon and McCurley [21] describe a sieving process they used to determine which pairs of relatively prime u_1, u_2 will produce a B-smooth w_1, where the polynomials are the ones described by Coppersmith's method. The main idea was to fix u_1 and set up an array of the u_2 polynomials. Note that we would not actually store these polynomials, but we need a way to know exactly which u_2 corresponds to which position in the array. The seemingly 'natural' way to do this is to map each u_2 polynomial to an integer by the mapping $u_2(x) \longmapsto u_2(2)$. We mentioned this sort of correspondence before, as this is essentially mapping a vector of coefficients, which one can view as the binary representation of an integer, to the integer itself.

Now consider such an array whose ith position corresponds to the u_2 polynomial where $u_2(2) = i$. Then, as u_1 is fixed, we wish to determine which u_2 (coupled with the fixed u_1) produce a B-smooth w_1 polynomial. To do this, we will use an approach similar to the example in the previous section. What we will do is see if the w_1 is divisible by an irreducible polynomial of degree j. If so, then we will add the integer j to the u_2 polynomial's position in the array. Once we do this for all irreducibles of degree $j \leq B$, we will have an array of integers. The value of position i in this array, call it z, gives the total degree of irreducible factors (of degrees up to B) of the w_1 polynomial that is constructed by the fixed u_1 and the u_2 corresponding to i. If this value z is greater than or equal to the degree of w_1 minus B, we know that this w_1 is B-smooth, for if not, then w_1 would have an irreducible factor of degree D larger than B and thus $\deg(w_1) - B > z$. Thus for each u_1, we can sift through all possible u_2 and pick out some particular ones that make w_1 B-smooth. Doing this for a range of u_1 should give us a sufficient number of smooth w_1. Then we will construct and factor the corresponding w_2 polynomial and check to see if it is B-smooth also. If so, we have a relation to add to our stockpile. This process should drastically reduce computation time, as we now are factoring only the w_1 and w_2 that we know w_1 to be smooth, compared with Coppersmith's method of factoring all w_1 and w_2 that are generated.

There are a few loose ends to tie up, the first being the fact that our sieve may not find all u_2 such that $w_1(x) = u_1(x)x^h + u_2(x)$ is B-smooth. One case when this can occur is if w_1 has a repeated factor of some degree, say $d_1 < B$, coupled with another repeated factor of degree $d_2 < B$ such that $d_1 + d_2 > B$. Another problem we have not yet addressed is that, in Coppersmith's method, we require that u_1 and u_2 are relatively prime. We will accommodate this by checking for the coprime condition after we have sieved and before we construct and factor, as gcd's are relatively easy to compute.

5.2.2 Stepping through the array

Now we must consider ways to go about this 'marking off' process. We want to minimize the number of divisibility checks we must perform, so once we find a polynomial that is divisible

by an irreducible, we want to be able to step through the array to the next polynomial that is divisible by the same irreducible, as we did in the earlier example. In their search for such a method, Gordon and McCurley note that there seems to be no obvious way to represent the polynomials so that representatives of a given residue class lie a fixed distance apart, as in the integer case. However, it is not necessary that we step a fixed distance, what matters is that we are able to step through all of the polynomials that are divisible by the fixed irreducible. They also noted that all of the polynomials in $\mathbb{F}_2[x]$ of degree less than some d can be thought of as the the vertices of a d-dimensional hypercube. Thus the idea of using a *Gray code* [41] was introduced. A Gray code is a sequence of bit strings that have some special properties. This sequence gives an efficient way to step through all of these vertices (polynomials), as well as a way to step through all polynomials that are divisible by some fixed irreducible. The way a Gray code does so can be thought of as 'toggling bits.' Let $G_1, G_2, \ldots, G_{2^d}$ be the binary Gray code of dimension d. For any integer z we have that G_z differs from G_{z+1} by $l(z)$, where $l(z)$ is the position of the lowest true bit of z. We give a brief illustration of how this is done with binary strings of 5 digits in Table 10 (note that $0 \leq l(x) \leq 4$).

Table 10: Gray code of dimension 5 on binary strings

z	$l(z)$	G_z	z	$l(z)$	G_z
1	0	00000	17	0	00011
2	1	10000	18	1	10011
3	0	11000	19	0	11011
4	2	01000	20	2	01011
5	0	01100	21	0	01111
6	1	11100	22	1	11111
7	0	10100	23	0	10111
8	3	00100	24	3	00111
9	0	00110	25	0	00101
10	1	10110	26	1	10101
11	0	11110	27	0	11101
12	2	01110	28	2	01101
13	0	01010	29	0	01001
14	1	11010	30	1	11001
15	0	10010	31	0	10001
16	4	00010	32		00001

It is interesting to note the reflected structure of this code. For example, if we look in Table 10, we can see that the first i bits of each word are reflected across the line between 2^i and $2^i + 1$. We reiterate that the Gray code is one of the most efficient ways to step through these bit strings, as we only toggle one bit each step.

5.2.3 The algorithm

We will now present the algorithm developed by Gordon and McCurley to sieve over $u_2(x)$ for a fixed $u_1(x)$. Here h and B are as defined in Coppersmith's method, and the parameter t denotes one more than the largest degree of the u_2 polynomials we will sieve over. Note that u_2 will have dual meaning here, alone as a polynomial in $\mathbb{F}_2[x]$, and in brackets (as an index of an array) as the integer that corresponds to the polynomial $u_2(x)$.

Table 11: The Coppersmith polynomial sieve algorithm

Step 1: Determining which w_1 are divisible by each irreducible in $\mathcal{B}$

```
for i = 0 to 2^t - 1
        s[i] := 0
for d = 1 to B
        dim := max(t - d, 0)
        for each irreducible g of degree d
                u_2 := u_1 x^h  mod g
                if deg(u_2) < t then
                for i = 1 to 2^dim
                        s[u_2] := s[u_2] + d
                        u_2 := u_2 + g x^{l(i)}
```

Step 2: Finding the u_2 that produce B-smooth w_1

```
for i = 0 to 2^t - 1
        if s[i] ≥ (deg(u_1) + h - B) then
                print u_1, u_2
```

There are some things we need to point out here, the first of which being the fact that we must have stored all of the irreducibles in $\mathbb{F}_2[x]$ of up to degree B in some file so they can be accessed for use in the algorithm. This is not too great a task, as for most of the computation that we are currently available to do, we will not need to store irreducibles of degree greater than around 20 (at present, a file containing the all the irreducibles of degrees up to 20 takes up a little less than 5 Mb of storage). Also, we must have some sort of method of converting a polynomial to an integer quickly, and vice versa. To summarize step one, we must find all u_2 such that w_1 is divisible by an irreducible g (for each g). We can do this by finding an initial u_2 and then stepping through the array to find all other u_2. Step two gives us a way of determining which of these u_2 give a smooth w_1, for if the value in the u_2 position in the array is greater than $\deg(w_1) - B = \deg(u_1(x)x^h) - B$, then we know w_1 is B-smooth.

Perhaps the most important piece of the algorithm is the initial computation $u_2 = u_1 x^h \mod g$. What this does is fundamental to our sieving approach: this sets u_2 as the smallest polynomial such that $w_1 = u_1 x^h + u_2$ is divisible by g. This is so because if $u_2 \equiv u_1 x^h \mod g$, then $u_1 x^h + u_2 \equiv 0 \mod g$. After recording the degree of g in the position corresponding to this u_2 in our array, we then step through all of the u_2 that give w_1 divisible by g. These u_2 are just all multiples of g times the initial u_2 (found by $u_2 := u_2 + gx^{l(i)}$) that are of degree less than t. The conditional that tests the degree of the initial u_2 to see if it is less than t is

merely a safeguard to protect against sieving over g that are of degree larger than t. In the selection of the Coppersmith parameters, it turns out that we will select $t = B$, as both are on the order of $n^{1/3}$.

We can demonstrate what the algorithm does by looking at an example.

EXAMPLE 5.2: Let $B = 6$ and $h = 8$. We fix $u_1(x) = x^3 + 1$, and we wish to find all $u_2(x)$ of degrees less than or equal to 6 such that $w_1(x) = u_1(x)x^h + u_2(x)$ is B-smooth. Thus $t = 7$ (the array covers all 2^7 polynomials of degree less than 7), and we begin proceeding through the algorithm. We first set $d = 1$ and thus $dim = 6$, and now wish to sieve with each irreducible of degree 1 (namely x and $x+1$). So we let $g = x$, and thus $u_2 = (x^3+1)x^8 \mod x$, or $u_2 = 0$. Then, since $\deg(u_2) < 6$, we begin to place a 1 (d) in each position of the array that corresponds to a u_2 where w_1 is divisible by x. We do so by going through the loop, beginning with $i = 1$.

$$\begin{aligned}
s[0] &= s[0] + 1 \\
u_2 &= 0 + x \cdot x^{l(1)} = x \cdot x^0 = x \\
s[2] &= s[2] + 1 \\
u_2 &= x + x \cdot x^{l(2)} = x + x \cdot x^1 = x^2 + x \\
s[6] &= s[6] + 1 \\
u_2 &= x^2 + x + x \cdot x^{l(3)} = x^2 + x + x = x^2 \\
s[4] &= s[4] + 1 \\
u_2 &= x^2 + x \cdot x^{l(4)} = x^2 + x \cdot x^2 = x^3 + x^2 \\
s[12] &= s[12] + 1 \\
&\vdots
\end{aligned}$$

And we continue until $i = 2^6 = 64$. Then we set $g(x) = x + 1$, and we begin the loop again. We find

$$\begin{aligned}
u_2 &= u_1 x^h \mod g = x^{11} + x^8 \mod (x+1) = 0 \\
s[0] &= s[0] + 1 \\
u_2 &= 0 + (x+1)x^{l(1)} = x + 1 \\
s[3] &= s[3] + 1 \\
u_2 &= x + 1 + (x+1)x^{l(2)} = x + 1 + x^2 + x = x^2 + 1 \\
s[5] &= s[5] + 1 \\
u_2 &= x^2 + 1 + (x+1)x^{l(3)} = x^2 + 1 + x + 1 = x^2 + x \\
s[6] &= s[6] + 1 \\
&\vdots
\end{aligned}$$

Again we continue until $i = 64$. We have exhausted all of the irreducibles of degree 4, so we

set $d = 2$, and thus $dim = 5$. Now we start with $g = x^2 + x + 1$, and we proceed as

$$\begin{aligned} u_2 &= u_1 x^h \mod g = x^{11} + x^8 \mod (x^2 + x + 1) = 0 \\ s[0] &= s[0] + 2 \\ u_2 &= 0 + (x^2 + x + 1)x^{l(1)} = x^2 + x + 1 \\ s[7] &= s[7] + 2 \\ u_2 &= x^2 + x + 1 + (x^2 + x + 1)x^{l(2)} = x^2 + x + 1 + x^3 + x^2 + x = x^3 + 1 \\ s[9] &= s[9] + 2 \\ &\vdots \end{aligned}$$

We continue until $i = 32$. Then we move on to $d = 3$, $dim = 4$, and $g = x^3 + x + 1$, and continue in the same fashion until we have covered all irreducibles of degree less than 7. We then run through our array, and any entry that is larger than $3 + 8 - 6 = 5$ will correspond to a u_2 that makes w_1 6-smooth. ▲

A first glance at the algorithm and the above example indicates that we should see a dramatic improvement in the performance of Stage 1 of the Index Calculus method, and not only because of our newfound method of selectivity. The operations that we perform in this sieve loop are relatively simple operations, most of which are moving through an array and integer addition. There are a few modular multiplications, all of which are of small degree. Gordon and McCurley note that the actual operation counts come close to the operation counts for the original Coppersmith method, but the fact that many of the operations are much simpler reduces the actual running time. Gordon and McCurley give operation counts for the number of steps to sieve a range of (u_1, u_2) pairs in [21], and conclude that the advantage sieving has over Coppersmith's method is immense as n (and thus B, h) tend to infinity.

EXAMPLE 5.3: Let $q = 2^{25}$ and let $f(x)$ be the primitive polynomial $x^{25} + x^3 + 1$. Then $B = d = 7$, $h = 13$, and $2^k = 2$. We will employ the Gordon and McCurley sieve to determine which (u_1, u_2) pairs give a 7-smooth w_1. Then we will construct and factor the w_2 to check if it is 7-smooth as well. Table 12 is a list of (u_1, u_2) pairs (in integer form) that produced a 7-smooth w_1 and w_2.

Note that the largest u_1 needed to find $2|\mathcal{B}|$ smooth relations was the polynomial corresponding to the integer 11, which is $x^3 + x + 1$. Thus our sieve loop only executed 11 times, and the routine finished in a little less than a second. After these relations were constructed and factored, the linear system was solved modulo $2^{25} - 1 = 31 \cdot 601 \cdot 1801$ by Maple, which required around six seconds of computation time. Table 13 is a list of the logarithms of the 41 elements of the factor base, excluding $\log_x x = 1$. The irreducibles in the factor base are represented by their corresponding integer.

For example, the polynomial corresponding to the integer 117 is the factor base element $x^6 + x^5 + x^4 + x^2 + 1$. From Table 13, we have that $\log_x x^6 + x^5 + x^4 + x^2 + 1 = 19365177$, so $x^{19365177} \mod f \equiv x^6 + x^5 + x^4 + x^2 + 1$. ▲

Table 12: (u_1, u_2) pairs that produce a 7-smooth w_1 and w_2

(1, 6)	(1, 8)	(1, 9)	(1, 10)	(1, 11)	(1, 13)	(1, 15)	(1, 16)
(1, 19)	(1, 21)	(1, 23)	(1, 24)	(1, 28)	(1, 29)	(1, 30)	(1, 36)
(1, 42)	(1, 52)	(1, 56)	(1, 63)	(1, 64)	(1, 78)	(1, 96)	(1, 107)
(1, 109)	(1, 110)	(1, 112)	(1, 115)	(1, 117)	(2, 1)	(2, 3)	(2, 5)
(2, 17)	(2, 21)	(2, 35)	(2, 65)	(2, 81)	(2, 87)	(2, 105)	(2, 107)
(3, 1)	(3, 4)	(3, 7)	(3, 16)	(3, 37)	(3, 56)	(3, 69)	(3, 84)
(3, 122)	(4, 1)	(4, 9)	(4, 21)	(4, 31)	(4, 61)	(4, 125)	(5, 2)
(5, 8)	(5, 19)	(5, 31)	(5, 42)	(5, 64)	(5, 73)	(6, 109)	(7, 6)
(7, 16)	(7, 26)	(7, 31)	(7, 48)	(7, 50)	(7, 61)	(7, 88)	(7, 96)
(7, 111)	(8, 21)	(8, 27)	(8, 35)	(8, 65)	(8, 69)	(8, 121)	(8, 123)
(9, 25)	(9, 32)	(9, 104)	(10, 19)	(10, 31)	(10, 37)	(10, 49)	(11, 8)
(11, 28)	(11, 32)	(11, 45)	(11, 50)	(11, 51)	(11, 102)	(11, 107)	(11, 124)

Table 13: Factor base logarithms for $\mathbb{F}_{2^{25}}$

i	L_i	i	L_i
3	1364812	115	25781272
7	32189644	117	19365177
11	201	131	4202914
13	2471824	137	30694753
19	18937161	143	4413727
25	10398293	145	20687632
31	589486	157	23774069
37	18413215	167	4880112
41	7817197	171	22866545
47	4589504	185	20800206
55	15446844	191	23251895
59	17769309	193	9153935
61	23823675	203	12909424
67	2940776	211	10006282
73	15347348	213	1455443
87	22115493	229	19891866
91	10302649	239	30988832
97	13588658	241	2053121
103	6681576	247	25363113
109	11993222	253	8447931

At first it seems that a sieve to determine if the w_2 polynomials would work similarly. Gordon and McCurley [21] note this, the only difficulty being that, to initialize the u_2 in the loop, it would be necessary to take roots (the 2^kth root to be precise). They also mention that doing so may not necessarily increase the performance, as only a small number of u_1, u_2

pairs survive the original sieve and even fewer survive the coprime test. However, a simple method to sieve over the w_2 polynomials may improve Stage 1 even further, we will discuss this more in section 8.2.

Another thing to note from the above example is that, if we start with the smallest possible u_1 and u_2, our polynomials to be factored will be of relatively lower degree than those generated by random choices of u_1 and u_2. This will, in general, decrease the amount of time necessary to factor each polynomial, thereby reducing the overall running time of the first stage even further.

5.3 Computational Comparisons

With three different ways to approach Stage 1 of the index calculus algorithm, we are ready to do some actual computations with these methods and compare the results. The routines used were

- `indcal.c` - the basic index calculus method, raising the generator x to random powers, and then test for smoothness.
- `copper.c` - select relatively prime u_1, u_2 and compute w_1, w_2, and then test for smoothness.
- `gordon.c` - sieve over all possible u_1, u_2 pairs to determine which will give smooth w_1, check for coprime and then compute w_2 and test for smoothness.

We asked each routine to find $2|\mathcal{B}|$ smooth relations for some selected n. To adequately assess the quickness of each method relative to the others, we stipulate the same parameters $B, n, h, 2^k$ for each routine (`indcal.c` requires only n and B). Table 14 presents a sample of the results of these computations.

We see that the Gordon and McCurley sieve gives us an incredible improvement in the actual computation time of Stage 1 for the selected n. We have every reason to believe that this trend would continue as n goes to infinity. However, our computational resources place a limit on the size of our experiments.

5.4 Improving Gordon and McCurley's Sieve

There are ways to perhaps increase the performance of the sieve even further. One suggestion is to relax our selection criterion for the (u_1, u_2) pairs in hopes of arriving at some valid (u_1, u_2) pairs that did not survive the original sieve. As we noted earlier in our the integer sieve and the discussion of the polynomial sieve, we may unknowingly omit a w_1 that is actually B-smooth if the w_1 has repeated smooth factors whose degrees sum to more than B. By relaxing the selection constraint by 1 or 2, we may come across more smooth w_1 (as

Table 14: Computation times for some selected n

Degree n	B	$2\lvert\mathcal{B}\rvert$	Computation Time (hh:mm:ss) `indcal.c`	`copper.c`	`gordon.c`
15	5	28	0.171	1.411	0.256
18	6	46	0.061	4.205	0.591
28	7	82	5.876	21.119	1.562
36	8	142	45.188	1:28.278	4.042
52	10	452	22:36.97	5:24.70	13.857
58	10	452	2:07:13.02	14:47.36	29.65
60	11	824	1:33:07.46	13:25.09	37.00
66	11	824	1:19:20.11	22:30.04	59.11
82	12	1494	>200h	1:36:50.95	3:28.97
100	13	2754	>200h	8:18:03.70	19:12.72
130	15	9440	>200h	36:45:33	2:12:53.62
148	16	17600	>200h	>200h	5:58:00.9
172	17	33020	>200h	>200h	22:07:34.7
180	17	33020	>200h	>200h	28:58:21
210	19	117272	>200h	>200h	180:58:56

many w_1 will have many factors of x). We implemented this change for some chosen n and present a table of the results below.

Table 15: Comparisons of relaxed selection criterion for `gordon.c`

Degree n	B	$2\lvert\mathcal{B}\rvert$	Time (sec.) for criterion $\deg(u_1)+h-B$	$\deg(u_1)+h-B-1$	$\deg(u_1)+h-B-2$
15	5	28	0.255	0.258	0.243
18	6	46	0.591	0.608	0.634
28	7	82	1.562	1.619	1.579
36	8	142	4.042	4.217	4.181
52	10	452	13.857	14.407	13.925
58	10	452	29.65	31.38	28.95
60	11	824	37.00	37.73	36.42
66	11	824	59.11	60.69	58.73
82	12	1494	208.97	206.08	207.11
100	13	2754	1152.72	1152.14	1136.93
130	15	9440	7973.62	7603.16	7421.29
148	16	17600	21480.9	20910.7	21703.8
172	17	33020	79654.7	78438.6	78682.7
180	17	33020	104301	100260	101109

We see that changing the selection criterion seems to have a small effect on the overall

running time for small n, but our real interests lie as $n \to \infty$. We notice the beginning of a trend as n passes 100, as the routines with the relaxed criterion do seem to run a little faster. Our only real gain will be realized if we can reduce the number of times the sieving loop is executed. Future experiments will likely determine whether or not changing the selection criterion actually reduces the number of loops necessary, as well as the ratio of the number of pairs that actually produce smooth relations to the number of pairs selected.

Another aspect of the sieve that impacts its performance is the choice of $f_1(x)$, where $f_1(x) = f(x) - x^n$. Gordon and McCurley discuss how the selection of this polynomial can affect the smoothness probability for w_1 and w_2 (for a random (u_1, u_2)). For the particular case $n = 593$, they determine the probability that a random (u_1, u_2) pair will produce a smooth w_2 for all possible f_1 of degree less than 11.

It is noted in [21] that the memory-access patterns for the Gordon and McCurley sieve are seemingly random and chaotic. For example, in the 'marking off' process, we may mark an entry in the array, then have to go back a large number of positions for the next entry to mark, then perhaps jump forward to get to the next one, almost like an irregular oscillation of some sorts. Computational studies have not been done to investigate this problem thoroughly, but this type of behavior could affect the performance of the sieve on processors that rely on using memory caches. At present there seems to be no obvious way to 'straighten out' this flaw; more research could possibly determine a better way to step through the array. However, as we will see in the next section, we can construct a different sieve that could step through the array smoothly, in fact always to the right (increasing indices).

6 A General Polynomial Sieve

6.1 Generalizing Gordon and McCurley's Sieve

Coupled with the dramatic increase in performance, the ideas that motivated Gordon and McCurley spark our interest even further. In particular, we wish to develop a sieving method that will work for polynomials in general, not just Coppersmith polynomials.

We begin this discussion by letting $\phi_i(x)$, $i = 1, \ldots, b$ denote fixed polynomials in $\mathbb{F}_q[x]$. Consider the polynomial

$$A(x) = a_1\phi_1(x) + a_2\phi_2(x) + \cdots + a_b\phi_b(x) = \sum_{i=1}^{b} a_i\phi_i(x). \tag{6.1}$$

for some $a_i \in \mathbb{F}_q$, $1 \leq i \leq b$. Our goal is to determine what sort of $A(x)$ are divisible by a fixed monic irreducible $g(x) \in \mathbb{F}_q[x]$ of degree t. Essentially, we wish to determine the conditions on the a_i such that

$$A(x) \equiv 0 \mod g(x) \tag{6.2}$$

or

$$\sum_{i=0}^{b} a_i \phi_i(x) \equiv 0 \mod g(x) \tag{6.3}$$

or

$$a_1\phi_1(x) + a_2\phi_2(x) + \cdots + a_b\phi_b(x) \equiv 0 \mod g(x). \tag{6.4}$$

Now we can reduce each $\phi_i(x)$ modulo $g(x)$. Thus we will let, for each i,

$$\phi_i(x) \equiv \sum_{j=0}^{t-1} g_{ij} x^j \mod g, \tag{6.5}$$

and then

$$a_i\phi_i(x) \equiv a_i \sum_{j=0}^{t-1} g_{ij}x^j \equiv \sum_{j=0}^{t-1} a_i g_{ij} x^j \mod g. \tag{6.6}$$

Then we can write

$$\begin{aligned} A(x) &\equiv a_1 \sum_{j=0}^{t-1} g_{1j}x^j + \cdots + a_b \sum_{j=0}^{t-1} g_{bj}x^j \\ &\equiv \sum_{j=0}^{t-1} a_1 g_{1j} x^j + \cdots + \sum_{j=0}^{t-1} a_b g_{bj} x^j \\ &\equiv \sum_{j=0}^{t-1} \left(\sum_{i=1}^{b} a_i g_{ij} \right) x^j \mod g. \end{aligned} \tag{6.7}$$

Now if $A(x) \equiv 0 \mod g$, then we would have

$$\sum_{i=1}^{b} a_i g_{ij} = 0, \quad 0 \le j \le t-1, \tag{6.8}$$

and since we know each of the g_{ij} (as the ϕ_i are fixed), we can find all $A(x)$ that are divisible by g by solving the linear system given by (6.8) for $0 \le j \le t-1$. Note that this system has t equations and b unknowns. In cases where $t < b$ this system will be underdetermined, so the solutions can be expressed in terms of arbitrary parameters, say $\alpha_1, \ldots, \alpha_r$. Thus there are q^r such polynomials $A(x)$ that are divisible by g. Running through all of the

Table 16: The general polynomial sieve algorithm

Step One: Determine which polynomials are divisible by each irreducible

```
for i = 0 to 2^b − 1
        s[i] := 0
for t = 1 to B
        for each irreducible g of degree t
                for i = 1 to b
                        compute φ_i ≡ Σ_{j=0}^{t−1} g_ij x^j  mod g
                solve Σ_{i=1}^{b} a_i g_ij = 0,  0 ≤ j ≤ t − 1
                for all values of the arbitrary parameters
                        determine A(x)
                        s[A(x)] := s[A(x)] + t
```

choices for $\alpha_1, \ldots, \alpha_r$ will give us these polynomials. Then, as before, we can add t to the position in the array corresponding to these polynomials. This array would be q^b elements long, and positions would correspond to the base q representation of the integer given by $(a_b a_{b-1} \cdots a_2 a_1)_q$.

Table 16 gives the general form of this new polynomial sieve. Some of the steps listed in the table would involve many substeps, such as the **solve** step as well as **determine** $A(x)$. These processes are best illustrated by an example.

EXAMPLE 6.1: Let $\phi_i(x) = x^i$, $0 \le i \le 6$, and let $g \in \mathbb{F}_2[x]$ be the irreducible $x^3 + x + 1$. Essentially we wish to find all polynomials in $\mathbb{F}_2[x]$ of degree up to 6 (inclusive) that are divisible by g. We begin by calculating the powers of x (up to 6) mod g.

$$\begin{aligned} x &= x \\ x^2 &= x^2 \\ x^3 &\equiv x + 1 \\ x^4 &\equiv x^2 + x \\ x^5 &\equiv x^2 + x + 1 \\ x^6 &\equiv x^2 + 1 \end{aligned}$$

Now a polynomial of degree up to 6 is of the form

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 \tag{6.9}$$

and thus

$$\begin{aligned} A(x) &\equiv a_0 + a_1 x + a_2 x^2 + a_3(x+1) + a_4(x^2+x) + a_5(x^2+x+1) + a_6(x^2+1) \\ &\equiv (a_0 + a_3 + a_5 + a_6) + (a_1 + a_3 + a_4 + a_5)x \\ &+ (a_2 + a_4 + a_5 + a_6)x^2 \mod g. \end{aligned} \tag{6.10}$$

We want all solutions to $A(x) \equiv 0 \mod g$, or

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \mod 2 \tag{6.11}$$

This has a solution of the form

$$\begin{aligned} a_0 &= a_3 + a_5 + a_6 \\ a_1 &= a_3 + a_4 + a_5 \\ a_2 &= a_4 + a_5 + a_6. \end{aligned} \tag{6.12}$$

Letting $\alpha_1 = a_3, \alpha_2 = a_4, \alpha_3 = a_5, \alpha_4 = a_6$ be arbitrary parameters, we have that all $A(x) \in \mathbb{F}_2[x]$ of degree up to 6 that are divisible by g are given by

$$(\alpha_1 + \alpha_3 + \alpha_4) + (\alpha_1 + \alpha_2 + \alpha_3)x + (\alpha_2 + \alpha_3 + \alpha_4)x^2 + \alpha_1 x^3 + \alpha_2 x^4 + \alpha_3 x^5 + \alpha_4 x^6. \tag{6.13}$$

These 16 polynomials are the zero polynomial and

$$\begin{array}{ccc} x^3 + x + 1 & x^4 + x^2 + x & x^4 + x^3 + x^2 + 1 \\ x^5 + x^2 + x + 1 & x^5 + x^3 + x^2 & x^5 + x^4 + 1 \\ x^5 + x^4 + x^3 + x & x^6 + x^2 + 1 & x^6 + x^3 + x^2 + x \\ x^6 + x^4 + x + 1 & x^6 + x^4 + x^3 & x^6 + x^5 + x \\ x^6 + x^5 + x^3 + 1 & x^6 + x^5 + x^4 + x^2 & x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \end{array}$$

Thus we would add a 3 to each position in the array corresponding to these polynomials. These positions are 0, 11, 22, 29, 39, 44, 49, 58, 69, 78, 83, 88, 98, 105, 116, and 127. ▲

At first it seems like this sort of sieve may be too costly operation-wise; a closer look suggests the opposite. The reductions modulo g are not too costly, as this is done only once for each g, and g is of relatively low degree. Also, many powers of $x \mod g$ can be computed by a repeated-squaring method. Further, the resulting system of equations will consist of only $\deg(g)$ rows, and will not require many operations to find the solution in the form of arbitrary parameters.

This brings us to an important note about the implementation of this sieve. In example 6.1, we let the a_i with the largest indices be the arbitrary parameters. One might ask why one of a_0, a_1, or a_2 were not allowed to be arbitrary. Recall that these a_i are really bits in the binary representation of a position in the array, with a_0 corresponding to the least significant bit and a_6 the most significant. Letting the arbitrary parameters correspond to

the most significant bits in this integer ensures that our walk through the array is a smooth one. In fact we will always increase the integer value of the position in the array (move to the right). Hence we avoid the somewhat chaotic memory-access patterns characteristic of the Gordon and McCurley sieve. It is useful to examine how this is so, we will employ the previous example to illustrate. We have four arbitrary parameters, so we can run through all possibilities by examining the binary representation of the integers 0 through 15. We will let $a_6a_5a_4a_3$ be this representation. For example, $13 = 1101_2$ would correspond to $a_3 = 1$, $a_4 = 0$, $a_5 = 1$, and $a_6 = 1$. The following table shows how we step through the array by adding 1 to the integer corresponding to the previous position. Recall that $a_0 = a_3 + a_5 + a_6$, $a_1 = a_3 + a_4 + a_5$, and $a_2 = a_4 + a_5 + a_6$. Using the same approach in each application of the

Table 17: Stepping through the array for Example 6.1

integer	$a_6a_5a_4a_3$	$a_6a_5a_4a_3a_2a_1a_0$	position
0	0000	0000000	0
1	0001	0001011	11
2	0010	0010110	22
3	0011	0011101	29
4	0100	0100111	39
5	0101	0101100	44
6	0110	0110001	49
7	0111	0111010	58
8	1000	1000101	69
9	1001	1001110	78
10	1010	1010011	83
11	1011	1011000	88
12	1100	1100010	98
13	1101	1101001	105
14	1110	1110100	116
15	1111	1111111	127

polynomial sieve will minimize the computation time spent on stepping through the array. For example, Table 18 lists the positions to step through the array for $g = x^3 + x^2 + 1$, the next irreducible we would use.

Table 18: Stepping through the array for $g = x^3 + x^2 + 1$.

integer	$a_6a_5a_4a_3$	$a_6a_5a_4a_3a_2a_1a_0$	position
0	0000	0000000	0
1	0001	0001101	13
2	0010	0010111	23
3	0011	0011010	26
4	0100	0100011	35
5	0101	0101110	46
6	0110	0110100	52
7	0111	0111001	57
8	1000	1000110	70
9	1001	1001011	75
10	1010	1010001	81
11	1011	1011100	92
12	1100	1100101	101
13	1101	1101000	104
14	1110	1110010	114
15	1111	1111111	127

Another thing stands out from the previous example that could reduce computation time. When the field we are working in is $\mathbb{F}_{2^n}$, we can use an easy way to arrive at the a_i that are determined by linear combinations of the arbitrary parameters. Note that we can write (6.12) as the matrix equation

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} \tag{6.14}$$

or as

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = a_3 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + a_4 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + a_5 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + a_6 \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}. \tag{6.15}$$

Thus when we are ready to determine the vector $\mathbf{v} = [a_0, a_1, a_2]^{\mathrm{T}}$, we can do so by adding scalar multiples of the vectors in (6.15). Since everything is binary, adding vectors is equivalent to the XOR operation (denote this operation by $\oplus$). The XOR operation, sometimes termed symmetric difference, is basically the bit-wise exclusive or comparison on binary strings. For example, $11011 \oplus 10110 = 01101$. Thus, when we increment the integer that corresponds to our selection of arbitrary parameters, we need only XOR $\mathbf{v}$ with the vectors whose coefficients have changed from the previous integer, i.e., the vectors whose coefficients

are bits that get toggled in the incrementing of the integer. For example, note that if we start at the integer is 5, we have

	$a_6a_5a_4a_3$	bits	$a_2a_1a_0$
5	0101	-	100
6	0110	a_3, a_4	$100 \oplus 011 \oplus 110 = 001$
7	0111	a_3	$001 \oplus 011 = 010$

As the XOR operation is quick and easy to compute, this makes the determination of the next position efficient.

Perhaps the biggest advantage of the application of the general polynomial sieve in Example 6.1 over the one presented by Gordon and McCurley is that we sieve over all polynomials of degree d and below in one fell swoop whereas their sieve loop must be executed for each fixed u_1. Another advantage is that there are basically no restrictions on the forms of the $\phi_i(x)$ polynomials, only that they are fixed.

EXAMPLE 6.2: Let the polynomials $\phi_i(x) \in \mathbb{F}_2[x]$ be given by

$$\begin{aligned}
\phi_1(x) &= x^3 + 1 \\
\phi_2(x) &= x^7 + x^6 + x^4 + x + 1 \\
\phi_3(x) &= x^{22} + x^{19} + x^2 \\
\phi_4(x) &= x^5 \\
\phi_5(x) &= x^{37} + x^{36} + x^{35} + x^{34} \\
\phi_6(x) &= x^9 + x^2 + 1
\end{aligned}$$

For $A(x) = \sum_{i=1}^{6} a_i\phi_i(x)$, we wish to determine which choices for the a_i will produce a polynomial that is divisible by $g(x) = x^4 + x^3 + 1$. We first compute the necessary powers of x

modulo g:

$$\begin{aligned}
x^4 &\equiv x^3+1 \\
x^5 &\equiv x^3+x+1 \\
x^6 &\equiv x^3+x^2+x+1 \\
x^7 &\equiv x^2+x+1 \\
x^9 &\equiv x^2+1 \\
x^{19} &\equiv x^3+1 \\
x^{22} &\equiv x^2+x+1 \\
x^{34} &\equiv x^3+1 \\
x^{35} &\equiv x^3+x+1 \\
x^{36} &\equiv x^3+x^2+x+1 \\
x^{37} &\equiv x^2+x+1.
\end{aligned}$$

Now we can compute $\phi_i(x) \mod g$ for each i by substituting these powers of x into the proper places for the ϕ_i and combining like terms. Thus we have that

$$A(x) \equiv (a_1+a_4)+(a_2+a_3+a_4+a_5)x+0x^2+(a_1+a_3+a_4+a_5)x^3 \mod g \qquad (6.16)$$

and solving for the a_i gives way to the binary linear system

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \qquad (6.17)$$

The row-reduced form of the coefficient matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad (6.18)$$

and thus we have that $a_1 = a_2 = a_4$, $a_3 = a_5$, and a_6 are arbitrary (in $\mathbb{F}_2$). Running through all possibilities for a_4, a_5, and a_6, we add 4 to positions 0, 11, 20, 31, 32, 43, 52, and 63. ▲

6.2 The Polynomial Sieve Applied to Coppersmith Polynomials

A different strategy for Coppersmith polynomials would be to let $u_1(x) = \sum_{i=0}^{d} a_i x^i$ and $u_2 = \sum_{i=d+1}^{2d+1} a_i x^{i-(d+1)}$. Then we would run the polynomial sieve for the w_1 and w_2 polynomials

simultaneously for all irreducibles of degrees up to a given t, marking one array with two entries in each position. After this was completed, we would search through the array looking for positions with both entries large enough to ensure that w_1 and w_2 were smooth. Then we would construct and factor those w_1 and w_2.

Yet another strategy could be developed by constructing a sieve that would determine which u_2 produced a B-smooth w_2 for a fixed u_1. In Section 5.2 we mentioned that Gordon and McCurley did not develop such a sieve as it could be difficult to take 2^kth roots of polynomials. The general polynomial sieve developed in the previous section can be applied to sieve over the w_2 polynomials efficiently, without the need for square roots. We will now describe this approach with a little detail and an example.

Given $f(x) = x^n + f_1(x)$, h, 2^k, and an irreducible $g(x)$, let u_1 be fixed. We wish to determine if the polynomial

$$w_2(x) = u_1(x)^{2^k} f_1(x) x^{h2^k - n} + u_2(x)^{2^k} \tag{6.19}$$

is divisible by $g(x)$. We can do so by first forming the product $u_1(x)^{2^k} f_1(x) x^{h2^k - n}$, and then adding to that product the polynomial

$$u_2(x)^{2^k} = a_0 + a_1 x^{2^k} + a_2 x^{2 \cdot 2^k} + \cdots + a_d x^{d2^k}. \tag{6.20}$$

Then we will have an expression for w_2 in terms of x and the a_i, so we can apply our general sieving technique to this expression. We can best illustrate this with an example.

EXAMPLE 6.3: Let $f(x) = x^{20} + x^3 + 1$, $h = 11$, and $2^k = 2$. We will fix $u_1(x) = x^3 + x + 1$, and we wish to determine all u_2 of degrees up to 6 such that w_2 is divisible by the irreducible $g(x) = x^5 + x^3 + 1$. Now u_2 will be of the form

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 \tag{6.21}$$

and thus we will have

$$u_2(x)^2 = a_0 + a_1 x^2 + a_2 x^4 + a_3 x^6 + a_4 x^8 + a_5 x^{10} + a_6 x^{12}. \tag{6.22}$$

We begin by constructing the product

$$\begin{aligned} u_1(x)^2 f_1(x) x^{2 \cdot 11 - 20} &= (x^6 + x^2 + 1)(x^3 + 1) x^2 \\ &= x^{11} + x^8 + x^7 + x^5 + x^4 + x^2. \end{aligned} \tag{6.23}$$

So a w_2 constructed with this u_1 and a u_2 of degree less than 7 will be of the form

$$a_0 + (a_1 + 1)x^2 + (a_2 + 1)x^4 + x^5 + a_3 x^6 + x^7 + (a_4 + 1)x^8 + a_5 x^{10} + x^{11} + a_6 x^{12}. \tag{6.24}$$

We wish to determine conditions on the a_i such that $w_2 \equiv 0 \mod g$. We see that we need to compute the powers of x up to 12 mod g in order to find the form of $w_2 \mod g$.

$$\begin{aligned} x^5 &\equiv x^3+1 \\ x^6 &\equiv x^4+x \\ x^7 &\equiv x^3+x^2+1 \\ x^8 &\equiv x^4+x^3+x \\ x^9 &\equiv x^4+x^3+x^2+1 \\ x^{10} &\equiv x^4+x+1 \\ x^{11} &\equiv x^3+x^2+x+1 \\ x^{12} &\equiv x^4+x^3+x^2+x \end{aligned}$$

Using these we now arrive at an expression for $w_2 \mod g$.

$$\begin{aligned} w_2 \equiv\ & a_0+(a_1+1)x^2+(a_2+1)x^4+(x^3+1)+a_3(x^4+x)+(x^3+x^2+1) \\ +\ & (a_4+1)(x^4+x^3+x)+a_5(x^4+x+1)+(x^3+x^2+x+1) \\ +\ & a_6(x^4+x^3+x^2+x) \mod g. \end{aligned} \tag{6.25}$$

Collecting like terms and simplifying we have

$$\begin{aligned} w_2 \equiv\ & (a_0+a_5+1)+(a_3+a_4+a_5+a_6)x+(a_1+a_6+1)x^2 \\ +\ & (a_4+a_6)x^3+(a_2+a_3+a_4+a_5+a_6)x^4 \mod g. \end{aligned} \tag{6.26}$$

Recall that we want $w_2 \equiv 0 \mod g$, so we equate each coefficient in (8.21) to 0 and now must find all solutions to the linear system

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \mod 2. \tag{6.27}$$

We solve the system in terms of a_5 and a_6, and let $\alpha_1 = a_6$, $\alpha_2 = a_5$ be arbitrary parameters. Then all u_2 that yield a w_2 that is divisible by g are of the form

$$u_2 = (\alpha_2+1)+(\alpha_1+1)x+\alpha_2x^2+\alpha_1x^4+\alpha_2x^5+\alpha_1x^6 \tag{6.28}$$

where $\alpha_1, \alpha_2 \in \mathbb{F}_2$. These u_2 are the polynomials $1+x, x+x^3+x^5, 1+x^4+x^6$, and $x^3+x^4+x^5+x^6$. Thus we would increase the entry in positions 3, 42, 81, and 120 by 5, and move on to the next irreducible g. ▲

A similar sieve could be developed to fix the u_2 and find all u_1 that give a smooth w_2, a bit more arithmetic may be necessary for that approach. As we mentioned before the example, this sieve could be used in conjunction with the Gordon and McCurley approach. This can be accomplished by setting up two different arrays for each u_1, one that contains the degrees of irreducible factors of the w_1 and the other for w_2. Once both sieves have been executed, simply determine which entries give a B-smooth polynomial in both lists.

As we see from our discussion and examples, there are numerous other configurations of this sieve alone or together with the sieve of Gordon and McCurley that can be developed to find smooth relations of Coppersmith polynomials. The applications of this sieve are in no way limited to Coppersmith polynomials, as we will see in the next section.

6.3 The Polynomial Sieve Applied to Semaev Polynomials

As we have already mentioned, we can apply the general polynomial sieve to polynomials of any given form. One such form is the Semaev polynomials we discussed in section 4.3. For a given field and integer $u = 2^k$, we have that

$$C(x)^{2^k} = D(x) \mod f(x) \tag{6.29}$$

where $C(x) = \sum_{i \in T_u} a_i x^i$, $D(x) = \sum_{i \in T_{i_u}} a_i x^i$, and $f(x) = \sum_{i=0}^{n} x^i$ where 2^n is the cardinality of the field. We can implement this sieve easily for the $C(x)$ polynomials. We illustrate this with the following example.

EXAMPLE 6.4: We consider the case where $n = 28$, $B = 7$, and $u = 8$. Then the corresponding ordered Semaev sets are

$$\begin{aligned} T_u &= \{0, 1, 4, 5, 8, 11, 12\} \\ T_{i_u} &= \{0, 8, 3, 11, 6, 1, 9\}. \end{aligned}$$

We are interested in finding all the polynomials $C(x)$ that are divisible by $g(x) = x^5 + x^2 + 1$. We begin by computing the powers of $x \mod g$. We have

$$\begin{aligned} x^5 &\equiv x^2 + 1 \\ x^6 &\equiv x^3 + x \\ x^7 &\equiv x^4 + x^2 \\ x^8 &\equiv x^5 + x^3 = x^3 + x^2 + 1 \\ x^9 &\equiv x^4 + x^3 + x \\ x^{10} &\equiv (x^5)^2 = x^4 + 1 \\ x^{11} &\equiv x^5 + x = x^2 + x + 1 \\ x^{12} &\equiv x^3 + x^2 + x \end{aligned}$$

Now a polynomial $C(x)$ of this particular form can be written as

$$C(x) = a_0 + a_1x + a_4x^4 + a_5x^5 + a_8x^8 + a_{11}x^{11} + a_{12}x^{12} \tag{6.30}$$

and thus we have $C(x) \mod g$ is given by

$$a_0 + a_1x + a_4x^4 + a_5(x^2+1) + a_8(x^3+x^2+1) + a_{11}(x^2+x+1) + a_{12}(x^3+x^2+x). \tag{6.31}$$

Collecting like terms we see that

$$\begin{aligned} C(x) \equiv\ & (a_0 + a_5 + a_8 + a_{11}) + (a_1 + a_{11} + a_{12})x \\ & + (a_5 + a_8 + a_{11} + a_{12})x^2 + (a_8 + a_{12})x^3 + a_4x^4 \mod g \end{aligned} \tag{6.32}$$

and if we want to find the a_i that give $C(x) \equiv 0 \mod g$ we must solve the system

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_4 \\ a_5 \\ a_8 \\ a_{11} \\ a_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \mod 2. \tag{6.33}$$

The row-reduced form of the coefficient matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{6.34}$$

and thus we let $a_{12} = \alpha_1$ and $a_{11} = \alpha_2$ be arbitrary parameters. Then all polynomials of the form (8.15) that are divisible by g are given by

$$\alpha_1 + (\alpha_1 + \alpha_2)x + \alpha_2x^5 + \alpha_1x^8 + \alpha_2x^{11} + \alpha_1x^{12} \tag{6.35}$$

where $\alpha_i \in \mathbb{F}_2$. These polynomials are the zero polynomial and $x^{11}+x^5+x$, $x^{12}+x^8+x+1$, and $x^{12}+x^{11}+x^8+x^5+1$, and we would continue on to the next irreducible of degree 5. ▲

Note that our array will only need to be $2^{|T_u|}$ elements in length, as we can require that the kth bit of the integer corresponding to the $C(x)$ polynomial is equivalent to the coefficient of the x^{i_k} term in $C(x)$, where i_k is the $k+1$st element in T_u. For instance, if T_u is as in the above example, then we would increase positions $2^1 + 2^3 + 2^5 = 42$, $2^0 + 2^1 + 2^4 + 2^6 = 83$,

and $2^0 + 2^3 + 2^4 + 2^5 + 2^6 = 121$ in our array by 5, as a_0 corresponds to the 0th bit, a_5 to the 3rd bit, and so on.

Just as with Coppersmith polynomials, there are numerous ways to set up this sieve to aid in the acquisition of smooth relations. Perhaps the most powerful and practical way of doing so would be to sieve the $C(x)$ and $D(x)$ simultaneously, using an array with two entries in each position.

We now conclude our discussion of Stage 1 of the index calculus method for finding discrete logarithms in a finite field. We have seen that there are many ways to increase the performance of this stage, and we hope to have presented a method to aid the search for smooth relations even further. More work and computational experiments are needed to determine how these developments will affect the running time of Stage 1.

7 Solving Linear Systems over Finite Fields

7.1 Linear Systems Produced By Index Calculus

We now move on to ways to improve the running time of Stage 2 of the index calculus algorithm. We will neglect the first step of this stage, which involves taking the logarithm in base α of both sides of each congruence and placing these into matrix form, as this can be done in virtually no time at all with minimal computational considerations.. The difficult part of the stage is the obvious: solving the very large (and sparse) system of congruences modulo $q-1$. First we will note that we must be careful in our techniques, as many methods developed for use over the real numbers may cause some problems when we apply them to the modular system. One significant (and possibly often encountered) problem is division by zero: if our modulus is not prime, not every integer will have a multiplicative inverse with respect to the modulus. In the examples we presented here, each of the linear systems were solved either with $q-1$ being prime or modulo each prime factor of $q-1$, then the final solution was obtained by applying the Chinese Remainder Theorem. In practice, we may run into situations where $q-1$ is prime (or so large that primality is unknown) where we will not need to worry about division by zero. However it is probable that $q-1$ will factor into many primes, some of which may be small (if all were small then we could use the Silver-Pohlig-Hellman approach). In this case it is actually not recommended that we solve the system modulo each prime factor, but attempt to solve the system modulo $q-1$. If we are to use Gaussian elimination as our solution method, when we run into an integer that has no inverse, we simply pivot to another row whose leading nonzero term is invertible. This can break down if we happen upon a column that has no invertible elements modulo $q-1$. McCurley [34] suggests a approach to recover from this breakdown using the Euclidean algorithm to introduce zeros into columns under a nonzero element (not necessarily invertible). This approach will require $O(|\mathcal{B}|^3)$ operations for the elimination and around $O(|\mathcal{B}|^2)$ calls to the extended Euclidean algorithm, which is comparable to ordinary Gaussian elimination.

One advantage that the discrete linear solver has above solving real systems is that, as long as our computation allows for integers of arbitrary length, we will always attain an exact solution, there is no such thing as roundoff error. This is the first indicator to one of the puzzling questions regarding solving linear system over finite fields (or rings). These questions include: what sort of methods developed for real methods can we use for discrete systems? What sort of modifications are necessary to these methods? Why do these methods work in discrete settings?

7.2 Solution Methods

7.2.1 Ordinary Gaussian elimination

As we already know, we can use Gaussian elimination in just about any setting, continuous or discrete. However, for use in the index calculus method, this may not be most practical approach to solving these systems. The size of the system is always an important factor. For example, suppose we are trying to compute logarithms in a field of order around 2^{500}. Then an appropriate-sized factor base would consist of all irreducibles of degrees up to 26, which means that $|\mathcal{B}| = 5,387,991$. If we are successful in obtaining $2|\mathcal{B}|$ smooth relations, then we are now faced with the problem of solving a linear system with about 5×10^{13} entries. The ability to store such a system is immediately questioned, for if each entry could be represented by only one byte, then we would need about 5000 gigabytes to do so. Luckily, the linear systems produced by Stage 1 have few nonzero entries (especially in the last few columns) and the storage is no longer infeasible as we need only store those nonzero entries. This can be shown by examining the sparsity plot of a matrix generated by Matlab (figure 2). The plot is dark wherever there are nonzero entries in the matrix. This particular matrix was the one produced by the routine `gordon.c` for $n = 66$. The matrix has 852 rows and 452 columns, giving a total of 351,024 entries. But only 6484 (2%) of these entries are nonzero.

However, applying our traditional Gaussian elimination to these systems will produce what is commonly referred to as *fill-in*, which occurs when many previously zero entries become nonzero, and the space requirements increase. Also, Gaussian elimination requires on the order of $|\mathcal{B}|^3$ operations, and on a hypothetical computer that could perform a billion operations per second, the solution of the system (for $\mathbb{F}_{2^{500}}$) would require only around 35000 years to terminate. A little long to wait for a solution, we think.

7.2.2 Structured Gaussian elimination

As we search for ways to solve the linear system $\mathbf{AL} = \mathbf{b}$, we should not abandon Gaussian elimination altogether as a possible aid in our efforts. The objective of the process called *structured Gaussian elimination* is to take a matrix that has some special sparsity properties and exploit those properties in an effort to reduce the original system to a smaller one. Structured Gaussian elimination takes little time and can be implemented with little space

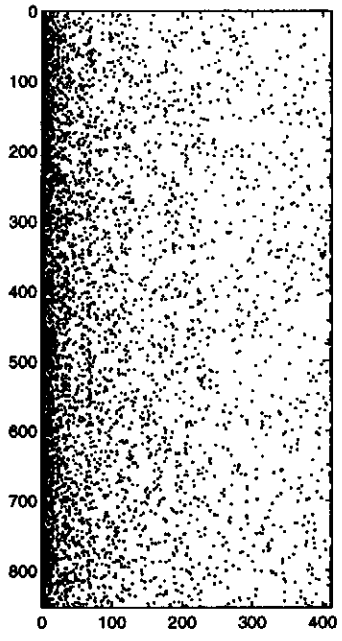

Figure 2: Sparsity plot of the matrix generated by `gordon.c` for $\mathbb{F}_{2^{66}}$

requirements. The main premise is to determine which columns contribute to the sparsity of the matrix, and those that contribute to its density. We will label those columns *light* and *heavy*, respectively. The following (taken from [30]) is a description of the basic steps of structured Gaussian elimination.

1. Delete all columns that have a single non-zero coefficient and the rows in which those columns have non-zero coefficients.
2. Declare some additional light columns to be heavy, choosing the heaviest ones.
3. Delete some of the rows, selecting those which have the largest number of non-zero elements in the light columns.
4. For any row which has only a single non-zero coefficient equal to ± 1 in the light column, subtract appropriate multiples of that row from all other rows that have non-zero coefficients on that column so as to make those coefficients 0.

As long as these steps are followed, the number of non-zero coefficients in the light part of the matrix will never increase. The complete method and its applications are discussed in detail in [42] and [30]. In some instances in [30], structured Gaussian elimination was successful in reducing the size of original systems by as much as 95%, and the resulting system could then be solved by a simple method such as ordinary Gaussian elimination.

7.2.3 Iterative and Krylov subspace methods

Over the years many types of iterative methods to find a solution to large (possibly sparse) linear system over the real numbers have been developed. One such process was developed by Lanczos [31]. The Lanczos method is similar to a family of methods known as *Conjugate Gradient* methods. Thorough explanations and analyses of the conjugate gradient methods and their variants can be found in [4], [20], [26], and [46]. These methods have also been implemented for use in the finite field setting, with much success. However, in most of these methods, scalar multiplication and inner products are required, which can present some problems in finite fields, mainly division by zero. This can occur during an orthogonalization process (such as the Gram-Schmidt process) when we need to divide by an inner product. We could then run into a *self-orthogonal* vector, i.e., a nonzero vector $\mathbf{v}$ such that $\mathbf{v}^T\mathbf{v} = \mathbf{0}$. There is really no such animal as a nonzero self-orthogonal vector over the reals, (aside from machine error), but they occur frequently in finite fields. This leads us to consider the geometry of the solution method, and further ask 'what exactly does a self-orthogonal vector look like?' Most iterative methods take a 'minimization of the residual' approach to solving a system $\mathbf{Ax} = \mathbf{b}$, where the residual is the vector $\mathbf{r}$ given by $\mathbf{r} = \mathbf{ax} - \mathbf{b}$.

Some methods have been developed especially for application to finite fields. Wiedemann [58] presents different algorithms for solving sparse linear systems over finite fields, using the fact that when a square matrix is repeatedly applied to a vector, the computed vector sequence is actually a linear recursive sequence. Niederreiter ([39]) discusses some topics in solving linear systems over finite fields. Other methods and variations of known methods are numerous; [12] and [38] survey some. Active research in this area is plentiful, as solving linear equations over finite fields is directly related to the problem of factoring large integers, factoring polynomials, and of course, the discrete logarithm problem.

7.3 Combined Methods

At present, when one wants to solve a large and partially sparse linear system over a finite field, many of the above methods are integrated to do so. Mostly structured Gaussian elimination is used at first, reducing the system so that an iterative method such as Lanczos or Wiedemann can be applied to arrive at the solution. Many such attempts are done in parallel, but distributed computing is not as attractive as it once was, as slight errors that can occur in the transfer of information from one machine to another can propagate quickly, voiding the integrity of the solution.

With the various improvements made to Stage 1 of the index calculus algorithm over the recent years, Stage 2 remains as our largest obstacle to finding logarithms in finite fields. Coming across a solution method that does not rely heavily on our computational limits seems unlikely, although no one knows what the future holds in store for us.

8 Conclusions

8.1 Relevance to Cryptography

The discrete logarithm problem has been a virtual 'thorn in the side' of many number theorists and cryptanalysts. Many years have gone by without enough progress to consider the problem solved. It is highly unlikely that a mathematician will accidentally trip over a polynomial time algorithm for finding discrete logarithms. So until that watched pot boils, we continue to, piece by piece, develop and refine the methods available to us. Even though the main focus this paper may seem relatively obscure to the study of secure information transfer, it is of significance in the search for cryptosystems impervious to attack. We must continue to search for ways to improve the known algorithms for finding discrete logarithms and try to develop new ones. The applicability of this research is evident, for in order to ensure that Alice and Bob can continue to communicate safely, we must play the role of Oscar. In this paper we have not changed the current state of public-key cryptosystems; more work is necessary to determine what impact the general polynomial sieve will have on the asymptotic running time of the index calculus method. Our computing resources could not begin to compare to some that are available today, as there are machines that can even frustrate world chess champions. However, we hope to have presented an interesting new search direction and/or sparked the reader's interest for ideas in this area with our brief survey.

As for the current ability to compute discrete logarithms in practice, we do have some remarks. As we stated before, Coppersmith successfully computed logarithms of the factor base for $\mathbb{F}_{2^{127}}$ in 1984, and Odlyzko suggested that logarithms in $\mathbb{F}_{2^n}$ were feasible for n up to 520 (1985). Gordon and McCurley worked on several specific cases of $\mathbb{F}_{2^n}$, including $n = 227, 313, 401, 503$, and 593. They were able to complete Stages 1 and 2 of the index calculus method for $n = 227, 313$, and 401, effectively disabling those fields for cryptographic purposes. For $n = 503$, a sufficient number of smooth relations have been collected, but (as of last word) the resulting linear system is yet to be solved. Logarithms in $\mathbb{F}_{2^{593}}$ seem to be out of reach at present, giving some evidence to support Odlyzko's 1985 conjecture. As noted in [34], the field $\mathbb{F}_{2^{593}}$ is of particular interest, for in 1989 it was known that a Canadian company named Newbridge Microsystems was producing a Data Encryption Processor chip that implemented arithmetic in this field, and was intended for use in some cryptographic protocols whose security was based on the infeasibility of the discrete logarithm problem in this field. We also see that, with the Gordon and McCurley sieve, one with quite modest computing resources can complete Stage 1 of the index calculus method in little time.

The 'hot item' in cryptography today is the use of elliptic curve groups over finite fields. An elliptic curve group is additive in structure and the security of many new cryptosytems is based on the elliptic curve logarithm problem. In [36], Menezes, et al., prove that, for certain elliptic curve groups, the elliptic curve logarithm problem can be reduced to the discrete logarithm problem in a finite field. This gives us more fuel for our motivational fire.

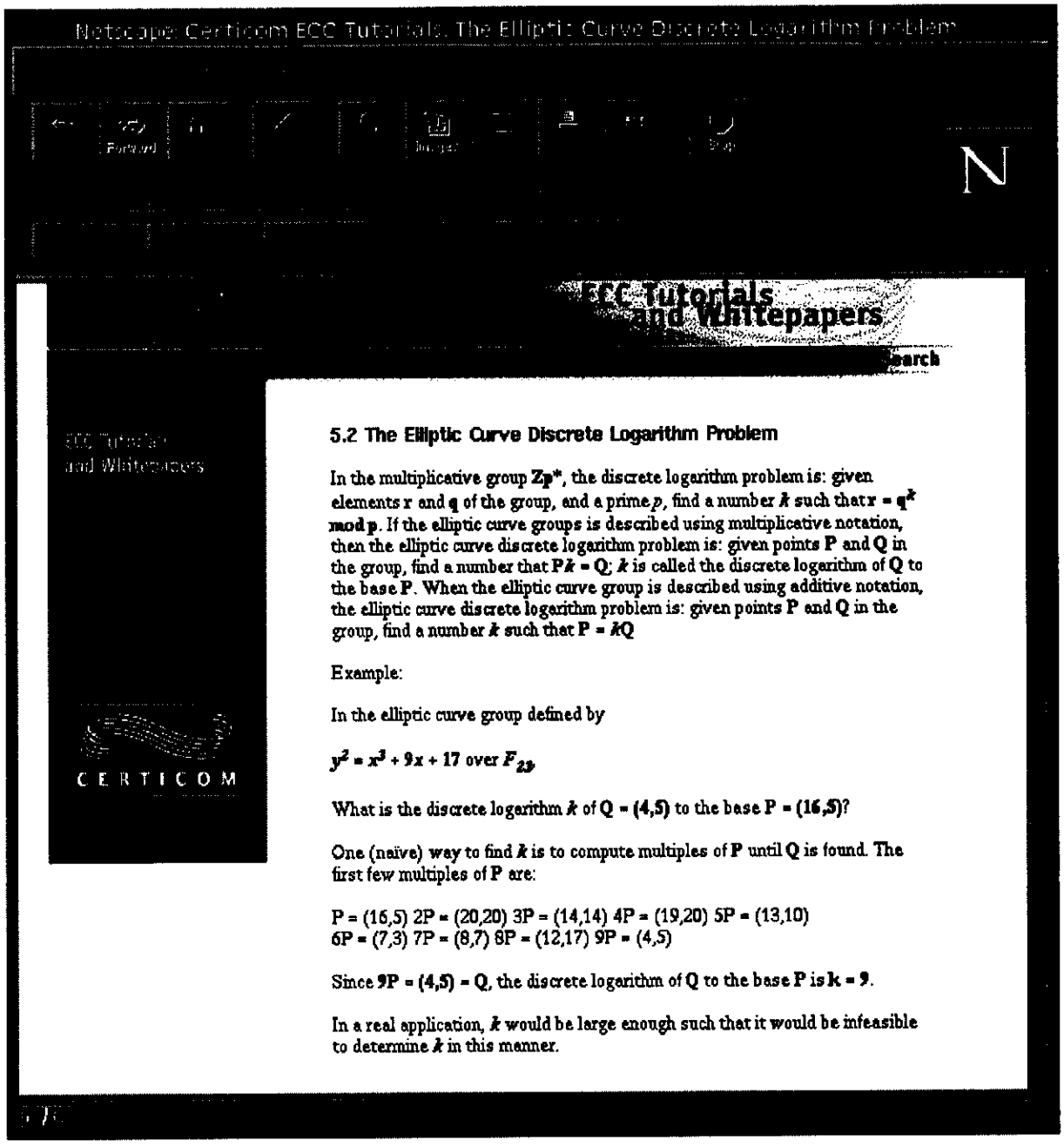

Figure 3: Certicom corp.'s Description of the Discrete Logarithm Problem for Elliptic Curve Groups.

8.2 Open Questions

We now present some interesting open questions to the reader, old and new, to be pondered. Many can be found in other works on the discrete logarithm problem, such as [34] and [51], and are the questions most relevant to our ability to determine discrete logarithms.

- Can one prove the equivalence of the Diffie-Hellman problem and the discrete logarithm problem?
- Can one prove the heuristic running times of the Coppersmith method, as well as for the Gordon and McCurley sieve?
- How can one improve the methods for solving large sparse linear systems over finite fields?
- Why do some methods developed for solving linear systems over the real numbers work for finite fields?
- Do there exist elliptic curve groups where discrete logarithms are easy, (analogous to fields with smooth orders)?
- Can one develop a better, faster way to sieve polynomials?
- Can one find a polynomial-time algorithm for finding discrete logarithms?
- Are there other cases where discrete logs are easy to find (such as smooth orders)?

As with so many things around us, we close just as we began. According to Hardy, Gauss and others should be rolling in their graves, for we have now taken some of the purity of number theory away as we exploit it for our practical purposes. Most often these purposes are relevant to cryptography, combinatorics, coding theory, and computation, and do have significant meaning in our 'human activities.' However, we think Hardy was mistaken in his judgment. Surely Karl and Evariste would be proud.

References

[1] L. M. Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography,* Proc. 20th IEEE Found. Comp. Sci. Symp. (1979), 55-60.

[2] _____, *The function field sieve,* Algorithmic number theory, Lec. Notes in Comp. Sci. **877** 1994, Springer-Verlag, 108-121.

[3] L. M. Adleman and J. DeMarrais, *A subexponential-time algorithm for discrete logarithms over all finite fields,* Math. Comp. **61** (1993), 1-15.

[4] O. Axelsson, *Iterative solution methods,* Cambridge University Press, 1996.

[5] R. L. Bender and C. Pomerance, *Rigorous discrete logarithm computations in finite fields via smooth polynomials,* Computational Perspectives on Number Theory, AMS/IP Stud. Adv. Math. **7** 1998, AMS, Providence, RI, 221-232.

[6] E. R. Berlekamp, *Factoring polynomials over finite fields,* Bell system Tech. J. **46** (1967), 1853-1859.

[7] G. Birkhoff and S. MacLane, *A survey of modern algebra,* Macmillan, New York, 1977.

[8] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, *Computing logarithms in finite fields of characteristic two,* SIAM J. Alg. Disc. Methods, **5** (1984), 276-285.

[9] D. G. Cantor and H. Zassenhaus, *A new algorithm for factoring polynomials over finite fields,* Math. Comp. **36** (1981), 587-592.

[10] F. Chatelin, *Eigenvalues of matrices,* John Wiley & Sons, Chichester, England, 1993.

[11] D. Coppersmith, *Fast evaluation of logarithms in fields of characteristic two,* IEEE Trans. Inform. Theory, IT -30, 587-594, 1984.

[12] _____, *Solving linear equations over GF(2): Block Lanczos algorithm,* Lin. Alg. Appl. **192** (1993), 33-60.

[13] D. Coppersmith, A. Odlyzko, and R. Schroeppel, *Discrete logarithms in GF(p),* Algorithmica **1** (1986), 1-15.

[14] J. M. Cuneaz, *Discrete Logarithms in Finite Fields.* Master's degree project paper. Department of Mathematical Sciences, Clemson University, April 25,1997.

[15] W. Diffie and M. E. Hellman, *New directions in cryptography,* IEEE Trans. Info. Theory, IT-22 (1976), 644-654.

[16] T. ElGamal, *A public key cryptosystem amd a signature scheme based on discrete logarithm,* IEEE Trans. Inform. Theory IT-31 (1985), 469-472.

[17] J. B. Fraleigh, *A first course in abstract algebra,* Addison-Wesley, Reading, Mass., 1994.

[18] S. Gao, *Discrete logarithms and Dickson polynomials,* Unpublished notes, Clemson University, 1996.

[19] J. von zur Gathen and V. Shoup, *Computing Frobenius maps and factoring polynomials,* Comput complexity **2** (1992), 187-224.

[20] G. H. Golub and C. F. van Loan, *Matrix computation,* Johns Hopkins University Press, Baltimore, 1983.

[21] D. M. Gordon and K. S. McCurley, *Massively parallel computation of discrete logarithms,* Advances in Cryptology - Crypto '92, Lec. Notes Comp. Sci. **740** 1993, Springer-Verlag, New York, 312-323.

[22] A. Guest, J. Howell, T. Lemmond, A. Locke, and C. Seawright, *Public-key cryptography and the discrete logarithm problem,* Unpublished notes, 1997.

[23] I. N. Herstein, *Topics in algebra,* Blaisdell, New York, 1964.

[24] E. Kaltofen, *Polynomial factorization 1982-1986,* Computers in Mathematics, Lec. Notes in Pure and Applied Math. **125** 1990, Marcel Dekker, New York, 285-309.

[25] _____, *Polynomial factorization 1987-1991,* Proc. Latin '92, Lec. Notes. Comp. Sci. **583** (São Paulo, Brazil), 1992, 294-313.

[26] C. T. Kelley, *Iterative methods for linear and nonlinear equations,* Frontiers in Applied Mathematics, SIAM, Philadelphia, 1995.

[27] N. Koblitz, *A course in number theory and cryptography,* Springer-Verlag, New York, 1994.

[28] M. Kraitchik, *Théorie des nombres,* vol. 1, Gauther-Villars, Paris, 1922.

[29] _____, *Recherches sur la théorie des nombres,* Gauther-Villars, Paris, 1924.

[30] B. A. LaMacchia and A. M. Odlyzko, *Solving large sparse linear systems over finite fields,* Advances in Cryptology - Crypto '90, Lec. Notes Comp. Sci. **537** 1991, Springer-Verlag, 109-133.

[31] C. Lanczos, *An iterative method for the solution of the eigenvalue problem of linear differential and integral operators,* J. Res. Nat. Bur. Standards Sec. B, **45**, 255-282.

[32] R. Lidl and H. Niederreiter, *Finite fields,* Addison-Wesley, Reading, Mass., 1983.

[33] R. Lovorn, *Rigorous, subexponential algorithms for discrete logaritm algorithms over finite fields,* Ph. D. Thesis, University of Georgia, June 1992.

[34] K. S. McCurley, *The discrete logarithm problem,* Proc. Symposia Applied Mathematics, AMS, 1990.

[35] R. J. McEliece, *Finite fields for computer scientists and engineers,* Kluwer, Boston, 1987.

[36] A. J. Menezes, I. F. Blake, X. H. Gao, R. C. Mullin, S. A. Vanstone, and T. Yaghoobian, *Applications of finite fields* Kluwer, Boston, 1993.

[37] R. Merkle, *Secrecy, authentication, and public-key systems,* Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., 1979.

[38] P. L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2),* Advances in cryptology - proceedings of *Eurocrypt '95,* (Saint Malo, 1995). 106-120, Lecture Notes in Computer Science, Springer, Berlin, 1995.

[39] H. Niederreiter, *Factorization of polynomials and some linear-algebra problems over finite fields,* Lin. Alg. App. **192** (1993), 301-328.

[40] _____, *New deterministic factorization algorithms for polynomials over finite fields,* Finite Fields: Theory, Applications, and Algorithms (G. L. Mullen and P. J.-S. Shiue, eds.), Contemporary Mathematics **168**, Amer. Math. Soc., 1994, 251-268

[41] A. Nijenhuis and H. S. Wilf, *Combinatorial algorithms,* Academic Press, New York, 1978.

[42] A. M. Odylzko, "Discrete Logarithms in Finite Fields and their Cryptographic Significance." *Advances in Cryptology: Proceedings of Eurocrypt '84.* Lecture notes in Computer Science **209** pgs. 224-314.

[43] _____, *Discrete logarithms and smooth polynomials,* Finite Fields: Theory, Applications, and Algorithms (G. L. Mullen and P. J.-S. Shiue, eds.), Contemporary Mathematics **168**, Amer. Math. Soc., 1994, 269-277.

[44] O. Ore, *Number theory and its history,* Dover, New York, 1976.

[45] R. Peralta, *Simultaneous security of bits in the discrete log,* Adv. in Cryptology (Proc. of Eurocrypt '85), Lecture notes in Computer Science, **219**, Springer-Verlag, New York, 1986, 62-72. Springer-Verlag, 1985.

[46] S. Pissanetsky, *Sparse matrix technology,* Academic Press, London, 1984.

[47] S. Pohlig and M. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance,* IEEE Trans. Inform. Theory IT-24 (1978), 106-110.

[48] J. M. Pollard, *Monte Carlo methods for index computation mod p,* Math. Comp. **32** (1978), 118-124.

[49] C. Pomerance, *Fast, rigorous factorization and discrete logarithm algorithms,* Discrete algorithms and complexity; Proc. Japa-U.S. joint seminar, June 4, 1986, Kyoto, Japan, Academic Press, Orlando, 1987, 119-143.

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

[51] O. Schirokauer, D. Weber, and T. Denny, *Discrete logarithms: the effective of the index calculus method,* Algorithmic Number Theory, Lec. Notes Comp. Sci. **1122** (1996), Springer, Berlin, 337-361.

[52] I. A. Semaev, *An algorithm for evaluation of discrete logarithms in some nonprime finite fields,* Preprint, 1994.

[53] D. Shanks, *Class number, a theory of factorization, and genera,* Proc. Symposium Pure Mathematics, AMS, 1972.

[54] L. D. Smith, *Cryptography: The science of secret writing,* Dover, New York, 1943.

[55] D. Stinson, *Cryptography: Theory and Practice.* Boca Raton, Fl.: CRC Press, 1995.

[56] S. S. Wagstaff Jr., *Greateast of the least primes in arithmetic progressiona having a given modulus,* Math. Comp. **33** (1979), 1073-1080.

[57] A. E. Western and J. C. P. Miller, *Tables of indices and primitive roots,* Royal Society Mathematical Tables, vol. 9, Cambridge Univ. Press, 1968.

[58] D. H. Wiedemann, *Solving sparse linear equations over finite fields,* IEEE Trans. Inf. Theory **32** (1986), 54-62.

A primpoly.c

This routine takes as input the degree n of the desired primitive polynomial as well as the factorization of $2^n - 1$. The output is the smallest $f_1(x)$ such that $f(x) = x^n + f_1(x)$ is primitive.

```
#include "BB_pX.h"
#include "BB_pXFactoring.h"
#include "BBFactoring.h"
#include "pair.h"
#include "ZZ.h"
#include "vector.h"
#include "vec_ZZ.h"
#include "tools.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
BB NextPoly(BB) ;
BB PrimitivePoly(vector(ZZ), long) ;
int NZterms(BB) ;


int main()
{
  long n ;
  BB fieldPoly, Xtothen ;
  double tm ;
  vector(ZZ) ord_factors ;

/* Determine the size of the field and construct the primitive polynomial */
  cout << "Enter the exponent of 2 \n" ;
  cin >> n ;
  cout << "Enter the vector of factors of 2^n-1 \n" ;
  cin >> ord_factors ;

  tm = GetTime() ;
  fieldPoly = PrimitivePoly(ord_factors, n) ;
  tm = GetTime() - tm ;
  SetCoeff(Xtothen, n) ;
  cout << fieldPoly-Xtothen << endl ;
  cout << "time = " << tm << " sec." << endl ;

}


/*************************************************************/
/******Increases a polynomial by 1 (integer representaton)****/

BB  NextPoly(BB tP)
{
```

```
  int carry = 1 ;
  long i = 0 ;
  while(carry)
  {
    tP.rep[i]++ ;
    if(tP.rep[i])
    {
      carry = 0 ;
    }
    i++ ;
  }
  return (tP) ;
}

/*************************************************************/
/****Finds the 'smallest' primitive polynomial of degree n****/

BB PrimitivePoly(vector(ZZ) ord_fact, long n)
{
  BB tp, Xtothen, triv, primTest ;
  long num_factors = ord_fact.length() ;
  SetCoeff(tp, n) ;
  set(triv) ;
  Xtothen = tp ;
  int Ok = 1 ;
  while(Ok != 0)
  {
    tp = NextPoly(tp) ;
    if (tp.rep[0])
    {
       if(NZterms(tp))
       {
         if(IterIrredTest(tp))
         {
           if(num_factors==1)
           {
             Ok = 0 ;
           }
           else
           {
             Ok = 1 ;
             while(Ok == 1)
             {
               int i = 1 ;
               while(i <= num_factors)
               {

                 PowerXMod(primTest, ord_fact(i), tp) ;
                 if(primTest == triv)
                 {
                   Ok = 2 ;
                   i = num_factors + 1 ;
```

```
                }
                else
                {
                  if(i == num_factors)
                  {
                    Ok = 0 ;
                    i++ ;
                  }
                  else
                  {
                    i++;
                  }
                }
              }
            }
          }
        }
      }
    }
    return tp ;
}

/*****************************************************************/
/*************Returns 1 if polynomial had odd number**********/
/*************of nonzero terms, 0 if even***********************/

int NZterms(BB Poly)
{
  long nzTerms = 0 ;
  long k ;
  k = deg(Poly) ;
  for(int i=0; i<=k; i++)
  {
    if (coeff(Poly,i))
    {
      nzTerms++ ;
    }
  }
  return(nzTerms % 2) ;
}
```

B indcal.c

This routine takes as input the degree n and the polynomial $f_1(x)$ such that $f(x) = x^n + f_1(x)$ is primitive, as well as the desired smoothness parameter B. The output is the matrix form of the logarithms of the smooth relations.

```
#include "BB_pX.h"
#include "BB_pXFactoring.h"
#include "BBFactoring.h"
#include "pair.h"
#include "ZZ.h"
#include "vector.h"
#include "vec_ZZ.h"
#include "tools.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
double pw(double, double) ;
ZZ Poly2Int(BB) ;
BB Int2Poly(ZZ) ;
vector(ZZ) Factors(vector(pair(BB,long))) ;
vector(long) Exponents(vector(pair(BB,long))) ;
long baseSize(long) ;
int SmoothTest(vector(pair(BB,long)), long) ;

int main()
{

/* Initialization */
  ifstream fin("PolyList") ;
  ofstream fout("indrows.out") ;
  ZZ twotothen, expon ;
  BB constPoly, fieldPoly, trial, primpart, Xtothen, irrpol ;
  vector(pair(BB,long)) factors ;
  vector(ZZ) ind, rowvec, fac ;
  vector(long) expn ;
  long B, n, numSmooth, t ;
  double tm ;

/* Determine the size of the field and factorbase */
  cout << "Enter the exponent of 2 \n" ;
  cin >> n ;
  SetCoeff(Xtothen, n) ;
  cout << "Enter the small part of the primitive polynomial \n" ;
  cin >> primpart ;
  fieldPoly = primpart + Xtothen ;
  BBModulus F ;
  build(F, fieldPoly) ;

/*  B = ceil(pw((double)n, 1.0L/3.0L)*pw(log(n), 2.0L/3.0L)) ;*/
  cout << "Enter degree of factorbase \n" ;
```

```
  cin >> B ;
  cout << "degree of factorbase = " << B << endl ;
  cout << "base size = " << baseSize(B) << endl ;

  int notdone = 1 ;
  while(notdone)
  {
    fin >> irrpol ;
    if(deg(irrpol)>B)
    {
      notdone=0;
    }
    else
    {
      append(ind, Poly2Int(irrpol)) ;
    }
  }
  t = ind.length() ;
  rowvec.SetLength(t) ;

/* More initialization */
  set(constPoly) ;
  numSmooth = 0 ;
  power(twotothen, 2, n) ;
  int Ok = 1 ;

  tm = GetTime() ;
/* Begin searching for smooth relations */
  while (numSmooth <= 2*baseSize(B))
  {
    RandomBnd(expon, twotothen) ;
    PowerXMod(trial, expon, F) ;
    CanZass(factors, trial) ;

/* If the polynomial is smooth, then print out the relation in matrix form */
    if(SmoothTest(factors, B))
    {
      numSmooth++ ;
      clear(rowvec) ;
      rowvec(t) = expon ;
      fac=Factors(factors) ;
      expn=Exponents(factors) ;
      for(int i=1; i<=fac.length(); i++)
      {
        if(fac(i)==2)
        {
          rowvec(t)=rowvec(t)-expn(i) ;
        }
        else
        {
          for(int j=2; j<=t; j++)
          {
```

```
            if(fac(i)==ind(j))
            {
              rowvec(j-1)=rowvec(j-1)+expn(i) ;
            }
          }
        }
      }
      fout << rowvec << endl ;
    }
  }
  tm = GetTime() - tm ;
  cout << numSmooth << endl << "time = " << tm << " sec\n" ;
  return 0 ;
}//end main

/*****************************************************************/
/*************Power function, for noninteger exponents********/

double pw(double x, double p)
{
  return exp(p * log(x)) ;
}

/*****************************************************************/
/*************Changes an integer to a polynomial in f_2*******/

BB Int2Poly(ZZ total)
{
  BB Poly ;
  long k ;
  for(k=0; k<=log(total)/log(2) + 1; k++)
  {
    if (bit(total, k))
    {
      SetCoeff(Poly, k) ;
    }
  }
  return Poly ;
}

/*****************************************************************/
/*************Changes a polynomial in f_2 to an integer*******/

ZZ Poly2Int(BB Poly)
{
  ZZ total = 0 ;
  ZZ temp ;
  long D = deg(Poly) ;
  for(int j=0; j<=D; j++)
  {
    if (coeff(Poly,j)==1)
    {
```

```
      power(temp, 2, j) ;
      total = total + temp ;
    }
  }
  return total ;
}

/*****************************************************************/
/*************Splits the vector pair, returns factors*********/

vector(ZZ) Factors(vector(pair(BB,long)) P)
{
  vector(ZZ) fac ;
  fac.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    fac(j) = Poly2Int(P(j).a) ;
  }
  return fac ;
}

/*****************************************************************/
/*************Splits the vector pair, reurns the exponents****/

vector(long) Exponents(vector(pair(BB,long)) P)
{
  vector(long) expo ;
  expo.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    expo(j) = P(j).b ;
  }
  return expo ;
}

/*****************************************************************/
/*****************Determines the size of the factor base*******/

long baseSize(long t)
{
  double s = 0 ;
  double tmp ;
  for (int i=1; i<=t; i++)
  {
    tmp = pow(2, i);
    s = s + tmp/i ;
  }
  return floor(s) ;
}

/*****************************************************************/
/*************Determines if a polynomial is B-smooth***********/
```

```
int SmoothTest(vector(pair(BB,long)) factors, long B)
{
  int IsSmooth = 1 ;
  for (long j = 1; j <= factors.length() ; j++)
  {
    if(deg(factors(j).a) > B)
    {
      IsSmooth = 0 ;
    }
  }
  return(IsSmooth) ;
}
```

C copper.c

This routine takes as input the degree n and the polynomial $f_1(x)$ such that $f(x) = x^n + f_1(x)$ is primitive, as well as the degree d of the largest $u_1(x)$ and $u_2(x)$ polynomials. The output is the matrix form of the logarithms of the smooth relations.

```
#include "BB_pX.h"
#include "BB_pXFactoring.h"
#include "BBFactoring.h"
#include "pair.h"
#include "ZZ.h"
#include "vector.h"
#include "vec_ZZ.h"
#include "tools.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
double pw(double, double) ;
ZZ Poly2Int(BB) ;
BB Int2Poly(ZZ) ;
long LowBit(long) ;
BB W2_construct(BB, BB, BB, long, long) ;
BB W1_construct(BB, BB, BB, long, long) ;
vector(ZZ) Factors(vector(pair(BB,long))) ;
vector(long) Exponents(vector(pair(BB,long))) ;
long baseSize(long) ;
int SmoothTest(vector(pair(BB,long)), long) ;

int main()
{

/* Initialization */
  ofstream fout("coprows.out") ;
  ifstream fin("PolyList") ;
  BB w2, w1, Xtothen, irrpol ;
  BB euclid, constPoly, fieldPoly, f1 ;
  vector(pair(BB,long)) factors1, factors2 ;
  vector(long) exp_w1, exp_w2 ;
  vector(ZZ) fac_w1, fac_w2, rowvec, ind ;
  long B, t, h, n, twotothek ;
  long  i, tt ;
  long  numSmooth ;
  double tm ;

/* Determine the size of the field and the primitive polynomial */
  cout << "Enter the exponent of 2 \n" ;
  cin >> n ;
  SetCoeff(Xtothen, n) ;
  cout << "Enter the small part of the primitive polynomial \n" ;
  cin >> f1 ;
```

```
  fieldPoly = f1 + Xtothen ;


/* Determine the Coppersmith parameters */
  B = ceil(pw((double)n, 1.0L/3.0L)*pw(log(n), 2.0L/3.0L)) ;
  cout << "degree of factorbase = " << B << endl ;
  twotothek = ceil(pw((n/log(n)),1.0L/3.0L)) ;
  h = floor( n/twotothek ) + 1 ;
  cout << "2^k = " << twotothek << endl << "h = " << h << endl ;
  cout << "base size = " << baseSize(B) << endl ;
  cout << "Enter the maximum size of u1,u2 \n" ;
  cin >> t ;

/* Initialize the vector of irreducibles in the factor base */
  int notdone = 1 ;
  while(notdone)
  {
    fin >> irrpol ;
    if(deg(irrpol)>B) notdone=0;
    else append(ind, Poly2Int(irrpol)) ;
  }
  tt = ind.length() ;
  rowvec.SetLength(tt) ;


/* More initialization */
  set(constPoly) ;
  numSmooth = 0 ;
  tm = GetTime() ;
  int Ok = 1 ;
  BB u1, u2 ;

/* Begin searching for smooth Coppersmith relations */
  while (numSmooth <= 2*baseSize(B))
  {
    random(u1, t+1) ;
    random(u2, t+1) ;
    GCD(euclid, u1, u2) ;

/* If gcd(u1,u2)=1,  then construct and factor w1 */
    if (euclid == constPoly)
    {
      w1 = W1_construct(u1, u2, fieldPoly, h, twotothek) ;
      CanZass(factors1, w1) ;
      if(SmoothTest(factors1, B))
      {

/* If the w1 is smooth, construct and factor w2 */
        w2 = W2_construct(u1, u2, fieldPoly, h, twotothek) ;
        CanZass(factors2, w2) ;
        if(SmoothTest(factors2, B))
        {
```

```
/* If both are smooth, then print the output in matrix form */
          numSmooth++ ;
          clear(rowvec) ;
          fac_w1=Factors(factors1) ;
          fac_w2=Factors(factors2) ;
          exp_w2=Exponents(factors2) ;
          exp_w1 = Exponents(factors1) ;
          for(int jj=1; jj<=exp_w1.length(); jj++)
          {
            exp_w1(jj) = twotothek*exp_w1(jj) ;
          }
          for(int jj=1; jj<=factors1.length(); jj++)
          {
            factors1(jj).b = twotothek*factors1(jj).b ;
          }
          for(int ii=1; ii<=fac_w1.length(); ii++)
          {
            if(fac_w1(ii)==2)  rowvec(tt)=rowvec(tt)-exp_w1(ii) ;
            else
            {
              for(int j=2; j<=tt; j++)
              {
                if(fac_w1(ii)==ind(j))  rowvec(j-1)=rowvec(j-1)+exp_w1(ii) ;
          } } }
          for(int ii=1; ii<=fac_w2.length(); ii++)
          {
            if(fac_w2(ii)==2)  rowvec(tt)=rowvec(tt)+exp_w2(ii) ;
            else
            {
              for(int j=2; j<=tt; j++)
              {
                if(fac_w2(ii)==ind(j))  rowvec(j-1)=rowvec(j-1)-exp_w2(ii) ;
          } } }
          fout << rowvec << endl ;
}
      }
    }
  }
  tm = GetTime() - tm ;
  cout << numSmooth << endl << "time = " << tm << " sec\n" ;
  return 0 ;
}//end main

/*************************************************************/
/************Power function, for noninteger exponents********/

double pw(double x, double p)
{
  return exp(p * log(x)) ;
}
```

```
/*************************************************************/
/*************Changes an integer to a polynomial in f_2*******/

BB Int2Poly(ZZ total)
{
  BB Poly ;
  long k ;
  for(k=0; k<=log(total)/log(2) + 1; k++)
  {
    if (bit(total, k))
    {
      SetCoeff(Poly, k) ;
    }
  }
  return Poly ;
}

/*************************************************************/
/*************Changes a polynomial in f_2 to an integer*******/

ZZ Poly2Int(BB Poly)
{
  ZZ total = 0 ;
  ZZ temp ;
  long D = deg(Poly) ;
  for(long j=0; j<=D; j++)
  {
    if (coeff(Poly,j)==1)
    {
      power(temp, 2, j) ;
      total = total + temp ;
    }
  }
  return total ;
}

/*************************************************************/
/*************Returns the lowest nonzero bit of an integer****/

long LowBit(long t)
{
  long ee = 0 ;
  while(!bit(t, ee))
  {
    ee++;
  }
  return ee ;
}


/*************************************************************/
/*************Constructs the w_2 Coppersmith polynomial*******/
```

```
BB W2_construct(BB u1, BB u2, BB fieldPoly, long h, long twotothek)
{

  BB tmp1, tmp2, tmp3, tmp4, Xtothen, w2 ;
  long n = deg(fieldPoly) ;
  long expon = h*twotothek - n ;
  SetCoeff(tmp1, expon) ;
  SetCoeff(Xtothen, n);
  PowerMod(tmp2, u1, twotothek, fieldPoly) ;
  tmp3 = tmp2*tmp1*(fieldPoly - Xtothen) ;
  PowerMod(tmp4, u2, twotothek, fieldPoly) ;
  w2 = tmp3 + tmp4 ;
  return w2 ;
}

/*****************************************************************/
/*************Constructs the w_1 Coppersmith polynomial*******/

BB W1_construct(BB u1, BB u2, BB fieldPoly, long h, long twotothek)
{
  BB tmp1, tmp2, w1 ;
  SetCoeff(tmp1, h) ;
  MulMod(tmp2, u1, tmp1, fieldPoly) ;
  w1 = tmp2 + u2 ;
  return w1 ;
}

/*****************************************************************/
/*************Splits the vector pair, returns factors*********/

vector(ZZ) Factors(vector(pair(BB,long)) P)
{
  vector(ZZ) fac ;
  fac.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    fac(j) = Poly2Int(P(j).a) ;
  }
  return fac ;
}

/*****************************************************************/
/*************Splits the vector pair, returns the exponents***/

vector(long) Exponents(vector(pair(BB,long)) P)
{
  vector(long) expo ;
  expo.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    expo(j) = P(j).b ;
```

```
  }
  return expo ;
}

/*****************************************************************/
/*****************Determines the size of the factor base*******/

long baseSize(long t)
{
  double s = 0 ;
  double tmp ;
  for (int i=1; i<=t; i++)
  {
    tmp = pow(2, i);
    s = s + tmp/i ;
  }
  return floor(s) ;
}

/*****************************************************************/
/*************Determines if a polynomial is B-smooth************/

int SmoothTest(vector(pair(BB,long)) factors, long B)
{
  int IsSmooth = 1 ;
  for (long j = 1; j <= factors.length() ; j++)
  {
    if(deg(factors(j).a) > B)
    {
      IsSmooth = 0 ;
    }
  }
  return(IsSmooth) ;
}
```

D gordon.c

This routine takes as input the degree n and the polynomial $f_1(x)$ such that $f(x) = x^n + f_1(x)$ is primitive, as well as some sieve parameters. The first is the degree of the largest $u_2(x)$ to sieve over, and then the user inputs the integers corresponding to the $u_1(x)$ to start sieving with and stop sieving with. The output is the matrix form of the logarithms of the smooth relations.

```
#include "BB_pX.h"
#include "BB_pXFactoring.h"
#include "BBFactoring.h"
#include "pair.h"
#include "ZZ.h"
#include "vector.h"
#include "vec_ZZ.h"
#include "tools.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
double pw(double, double) ;
ZZ Poly2Int(BB) ;
long Poly2Long(BB) ;
BB Int2Poly(ZZ) ;
long LowBit(long) ;
vector(long) SieveLoop(BB, long, long, long) ;
BB W2_construct(BB, BB, BB, long, long) ;
BB W1_construct(BB, BB, BB, long, long) ;
vector(ZZ) Factors(vector(pair(BB,long))) ;
vector(long) Exponents(vector(pair(BB,long))) ;
long baseSize(long) ;
int SmoothTest(vector(pair(BB,long)), long) ;

int main()
{

/* Initialization */
  ifstream fin("PolyList") ;
  ofstream fout("gordrows.out") ;
  BB u2, g, quitter, Xtothen, gx1, w2, w1 ;
  BB euclid, constPoly, fieldPoly, f1, irrpol ;
  vector(pair(BB,long)) factors1, factors2 ;
  vector(ZZ) fac_w1, fac_w2, rowvec, ind ;
  vector(long) s, exp_w1, exp_w2 ;
  ZZ ust, usp, tmp ;
  long B, t, h, n, twotothek, bs ;
  long  i, tt ;
  long numSmooth ;
  double tm ;

/* Determine the size of the field and construct the primitive polynomial */
```

```
  cout << "Enter the exponent of 2 \n" ;
  cin >> n ;
  SetCoeff(Xtothen, n) ;
  cout << "Enter the small part of the primitive polynomial \n" ;
  cin >> f1 ;
  fieldPoly = f1 + Xtothen ;


/* Determine the Coppersmith parameters */
  B = ceil(pw((double)n, 1.0L/3.0L)*pw(log(n), 2.0L/3.0L)) ;
  cout << "degree of factorbase = " << B << endl ;
  twotothek = ceil(pw((n/log(n)),1.0L/3.0L)) ;
  h = floor( n/twotothek ) + 1 ;
  cout << "2^k = " << twotothek << endl << "h = " << h << endl ;
  bs = baseSize(B) ;
  cout << "size of factor base = " << bs << endl ;

  cout << "Enter the maximum size of u2 to sieve over\n" ;
  cin >> t ;

/* Initialize the vector of irreducibles in the factor base */
  int notdone = 1 ;
  while(notdone)
  {
    fin >> irrpol ;
    if(deg(irrpol)>B) notdone=0;
    else append(ind, Poly2Int(irrpol)) ;
  }
  tt = ind.length() ;
  rowvec.SetLength(tt) ;

/* Enter the sieve parameters */
  cout << "Enter the integer to start the sieve (u1) \n" ;
  cin >> ust ;
  BB uStart = Int2Poly(ust) ;
  cout << "Enter the integer to stop the sieve " << endl ;
  cin >> usp ;
  BB uStop = Int2Poly(usp) ;


/* More initialization */
  set(constPoly) ;
  numSmooth = 0 ;
  tm = GetTime() ;
  int Ok = 1 ;
  BB u1 = uStart ;
  while (Ok)
  {
    s = SieveLoop(u1, t, B, h) ;

/* Determine which pairs of u1, u2 produce smooth w1 */
    for(i=0; i< s.length() ; i++)
```

```
    {
      if(s[i] >= deg(u1) + h - B)
      {
        u2 = Int2Poly(i) ;
        GCD(euclid, u1, u2) ;

/* If the w1 is smooth and gcd(u1,u2)=1,  then construct and factor w2 */
        if (euclid == constPoly)
        {
  w2 = W2_construct(u1, u2, fieldPoly, h, twotothek) ;
  CanZass(factors2, w2) ;
          if(SmoothTest(factors2, B))
          {

/* If the w2 is smooth, construct and factor w1 */
            w1 = W1_construct(u1, u2, fieldPoly, h, twotothek) ;
            CanZass(factors1, w1) ;
            if(SmoothTest(factors1, B))
            {

/* If both are smooth, then print the output in matrix form */
              numSmooth++ ;
              clear(rowvec) ;
              fac_w1=Factors(factors1) ;
              fac_w2=Factors(factors2) ;
              exp_w2=Exponents(factors2) ;
              exp_w1 = Exponents(factors1) ;
              for(int jj=1; jj<=exp_w1.length(); jj++)
              {
                exp_w1(jj) = twotothek*exp_w1(jj) ;
              }
              for(int ii=1; ii<=fac_w1.length(); ii++)
              {
                if(fac_w1(ii)==2)  rowvec(tt)=rowvec(tt)-exp_w1(ii) ;
                else
                {
                  for(int j=2; j<=tt; j++)
                  {
                    if(fac_w1(ii)==ind(j))  rowvec(j-1)=rowvec(j-1)+exp_w1(ii) ;
              } } }
              for(int ii=1; ii<=fac_w2.length(); ii++)
              {
                if(fac_w2(ii)==2)  rowvec(tt)=rowvec(tt)+exp_w2(ii) ;
                else
                {
                  for(int j=2; j<=tt; j++)
                  {
                    if(fac_w2(ii)==ind(j))  rowvec(j-1)=rowvec(j-1)-exp_w2(ii) ;
              } } }
              fout << rowvec << endl ;
            }
          }
```

```
        }
      }
    }

/* Increment u_1 and see if we are done */
    if ( numSmooth > 2*bs )
    {
      Ok = 0 ;
    }
    tmp = Poly2Int(u1) ;
    tmp++ ;
    if (tmp == usp)
    {
      Ok = 0 ;
    }
    u1 = Int2Poly(tmp) ;
  }
  tm = GetTime() - tm ;
  cout << numSmooth << endl << "time = " << tm << " sec\n" ;
  return 0 ;
}//end main

/*****************************************************************/
/*************Power function, for noninteger exponents********/

double pw(double x, double p)
{
  return exp(p * log(x)) ;
}

/*****************************************************************/
/*************Changes an integer to a polynomial in f_2*******/

BB Int2Poly(ZZ total)
{
  BB Poly ;
  long k ;
  for(k=0; k<=log(total+1)/log(2) + 1; k++)
  {
    if (bit(total, k))
    {
      SetCoeff(Poly, k) ;
    }
  }
  return Poly ;
}

/*****************************************************************/
/*************Changes a polynomial in f_2 to an integer*******/

ZZ Poly2Int(BB Poly)
{
```

```
  ZZ total = 0 ;
  ZZ temp ;
  long D = deg(Poly) ;
  for(int j=0; j<=D; j++)
  {
    if (coeff(Poly,j)==1)
    {
      power(temp, 2, j) ;
      total = total + temp ;
    }
  }
  return total ;
}

/*****************************************************************/
/*************Changes a polynomial in f_2 to a long integer***/

long Poly2Long(BB Poly)
{
  long total = 0 ;
  long D = deg(Poly) ;
  for(int j=0; j<=D; j++)
  {
    if (coeff(Poly,j)==1)
    {
      total = total + pow(2,j) ;
    }
  }
  return total ;
}


/*****************************************************************/
/*************Returns the lowest nonzero bit of an integer****/

long LowBit(long t)
{
  long ee = 0 ;
  while(!bit(t, ee))
  {
    ee++;
  }
  return ee ;
}


/*****************************************************************/
/*************The main Gordon-McCurley sieveing loop**********/

vector(long) SieveLoop(BB u1, long t, long B, long h)
{
  ifstream fin("PolyList") ;
```

```
  vector(long) s ;
  BB Xpoly, g, gxl, u2 ;
  long twototheT = pow(2, t) ;
  s.SetLength(twototheT) ;

    /* Main sieving loop */
    int NotDone = 1;
    while(NotDone)
    {
      fin >> g ;
      long d = deg(g) ;
      if (d <= B)
      {
        long dim = max(t-d, 0) ;
        clear(Xpoly) ;
        SetCoeff(Xpoly, h) ; /* x^h */
        MulMod(u2, u1, Xpoly, g) ; /* u2 = u1*x^h mod g */
        if (deg(u2) < t)
        {
          long twotothedim = pow(2, dim) ;
          for (long i=1; i<=twotothedim; i++)
          {
            long ind1 = Poly2Long(u2) ;
            s[ind1] = s[ind1] + d ;
            clear(Xpoly) ;
            SetCoeff(Xpoly, LowBit(i)) ;
            mul(gxl, g, Xpoly) ;
            add(u2, u2, gxl) ;
          }//end for
        }//end if
      }//end if
      else
      {
        NotDone = 0 ;
      }//end else
    }//end while
  return s ;
}

/*****************************************************************/
/*************Constructs the w_2 Coppersmith polynomial*******/

BB W2_construct(BB u1, BB u2, BB fieldPoly, long h, long twotothek)
{

  BB tmp1, tmp2, tmp3, tmp4, Xtothen, w2 ;
  long n = deg(fieldPoly) ;
  long expon = h*twotothek - n ;
  SetCoeff(tmp1, expon) ;
  SetCoeff(Xtothen, n);
  PowerMod(tmp2, u1, twotothek, fieldPoly) ;
  tmp3 = tmp2*tmp1*(fieldPoly - Xtothen) ;
```

```
  PowerMod(tmp4, u2, twotothek, fieldPoly) ;
  w2 = tmp3 + tmp4 ;
  return w2 ;
}

/*************************************************************/
/************Constructs the w_1 Coppersmith polynomial*******/

BB W1_construct(BB u1, BB u2, BB fieldPoly, long h, long twotothek)
{
  BB tmp1, tmp2, w1 ;
  SetCoeff(tmp1, h) ;
  MulMod(tmp2, u1, tmp1, fieldPoly) ;
  w1 = tmp2 + u2 ;
  return w1 ;
}

/*************************************************************/
/************Splits the vector pair, returns factors*********/

vector(ZZ) Factors(vector(pair(BB,long)) P)
{
  vector(ZZ) fac ;
  fac.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    fac(j) = Poly2Int(P(j).a) ;
  }
  return fac ;
}

/*************************************************************/
/************Splits the vector pair, returns the exponents***/

vector(long) Exponents(vector(pair(BB,long)) P)
{
  vector(long) expo ;
  expo.SetLength(P.length()) ;
  for(int j=1; j<=P.length(); j++)
  {
    expo(j) = P(j).b ;
  }
  return expo ;
}

/*************************************************************/
/****************Determines the size of the factor base*******/

long baseSize(long t)
{
  double s = 0 ;
  double tmp ;
```

```
  for (int i=1; i<=t; i++)
  {
    tmp = pow(2, i);
    s = s + tmp/i ;
  }
  return floor(s) ;
}

/*****************************************************************/
/*************Determines if a polynomial is B-smooth************/

int SmoothTest(vector(pair(BB,long)) factors, long B)
{
  int IsSmooth = 1 ;
  for (long j = 1; j <= factors.length() ; j++)
  {
    if(deg(factors(j).a) > B)
    {
      IsSmooth = 0 ;
    }
  }
  return(IsSmooth) ;
}
```

Index