Public Key Cryptography Using Discrete Logarithms in Finite Fields:

Algorithms, Efficient Implementation and Attacks

L. Maurits (1105909)

Supervisor: Dr R. J. Clarke

Thesis submitted for the degree of Honours in Pure Mathematics

SCHOOL OF MATHEMATICAL SCIENCES
DISCIPLINE OF PURE MATHEMATICS

THE UNIVERSITY OF ADELAIDE AUSTRALIA

Preface

The field of *cryptography* (from the Greek *kryptos*, meaning "hidden" and *graphein*, meaning "to write") is concerned with problems related to the security of information, such as: ensuring that information transmitted over a public channel cannot be understood by an eavesdropper; verifying that received information has indeed been sent by the party claiming to have sent it; and ensuring that the information has not been tampered with by unauthorised parties in transit. These problems have been of interest to humanity since ancient times, historically in connection with military and diplomatic affairs. The "shift cipher" of Julius Caesar and the cipher rods of the Spartan military are well known historical examples of cryptography. More recently, the efforts of the Allied forces in breaking the Nazi "Enigma" code have been credited with having shortened World War II by up to two years.

Since the advent and subsequent widespread adoption of the digital computer, the range of applications for cryptography has exploded and the field has matured from what was once arguably an art into a definite science of the highest mathematical sophistication. Today, cryptography is used to provide security to a host of daily activities, including but not limited to: the electronic transfer of money via ATMs, EFTPOS or internet banking and purchase systems; the transmission of voice, photo and video data over mobile telephone networks; the transmission and storage of electoral votes cast using electronic voting machines; and the use of email, instant messaging and other computer communication for confidential business discussion or collaboration, "whistleblowing", transmission of sensitive data such as police or hospital records and private personal communications.

Perhaps the most important change cryptography has undergone in its long history has been the revolutionary discovery of *public-key cryptography*, an idea both first proposed and accomplished in 1976 by Whitfield Diffie and Martin Hellman. All previous means of secure communication had required that the participating parties agree on a secret key beforehand. The security of any subsequent communication relied upon this key being agreed upon in perfect secrecy and remaining undiscovered by any third parties. This posed a major practical difficulty, especially when secure communication over long distances or between parties with no prior acquaintance was desired. Public-key cryptography removes this requirement and allows secure communication between two parties with no prior secret exchange of keys. This represented a major paradigm shift in the world of cryptography, one which enabled many of the applications of cryptography in wide use today.

Every instance of public-key cryptography has a certain difficult computational problem underpinning its security - security is assured as long as solving this problem is infeasible. While many difficult problems have been proposed as potential bases for public-key cryptography, just two problems underpin the security of all widely used public-key cryptography today. These are the *integer factorisation problem* of number theory and the *discrete logarithm problem* of group theory. Public-key systems based upon the latter problem are the subject of the present text.

This thesis is a survey of existing knowledge related to public-key cryptography which is based upon one particular instance of the discrete logarithm problem, concerning discrete logarithms over *finite fields*, although it contains much material which is relevant to the more general problem, and even to cryptography in general. The author deeply regrets that, due to space and time constraints, it is not exhaustive! Nevertheless, the most important examples of public-key cryptography using discrete logarithms, in terms of wide use and importance of applications, are discussed here. Issues relating to efficient computation in finite fields, which lead to efficient implementations of public-key cryptography, are also discussed, although there is a vast and detailed literature on this subject of which only the surface is scratched. Finally, algorithms for computing discrete logarithms and associated problems in finite fields are presented, which may be used to attack the cryptosystems presented.

A familiarity with the theory of finite fields, and associated algebraic concepts such as groups, rings and polynomials, as well as some basic number theory, is assumed in this thesis. A typical undergraduate education in pure mathematics should be sufficient for the material to be read with no major confusion. The reader who requires a reference on this theory, or who is interested in further reading regarding many of the topics discussed here, is directed towards three books in particular, which have been instrumental to the research which produced this thesis: Lidl and Niederreiter's highly regarded (though now somewhat dated) book *Finite Fields* [44] provides an encyclopedic reference for the fundamental theory of finite fields; Shparlinski's book *Finite Fields: Theory and Computation* [72] presents a fairly modern survey of major computational problems and applications related to finite fields and in particular includes a monumental bibliography of some 3075 books and papers on every aspect of the subject; The book of Blake, Gao, Mullin, Vanstone and Yaghoobian, *Applications of Finite Fields* [9], discusses some basic computational theory and many applications of finite fields, including some material extending that presented here. For general references on cryptography and its implementation, the classic (though again dated) books *Applied Cryptography* [65] by Bruce Schneier and the *Handbook of Applied Cryptography* [47] by Menezes, van Oorschot and Vanstone are recommend, as well as the more modern *Modern Cryptography: Theory and Practice* [45] by Wenbo Mao.

The material contained in this thesis is summarised on a chapter by chapter basis as follows:

Chapter 1 provides the setting and motivation for the remainder of the material. It introduces the public-key cryptography paradigm and also defines discrete logarithms and the discrete logarithm problem. Three widely used public-key "cryptographic primitives" which rely upon the difficulty of computing discrete logarithms for their security are presented: the Diffie-Hellman key exchange protocol, the Elgamal cryptosystem and the Digital Signature Algorithm, part of the Digital Signature Standard.

Chapter 2 discusses some details of efficient computation in finite fields. It presents methods for representing finite fields of both prime and prime power order, which are used in subsequent chapters. It also surveys some non-trivial algorithms for performing basic computational tasks with the given representations, namely multiplication and exponentiation. The problem of constructing irreducible polynomials over finite fields is treated briefly, since this problem is relevant to one of the given field representations. The material in this chapter is relevant to the efficient implementation of both the cryptographic primitives discussed in the previous chapter and the attacks on these primitives developed in later chapters.

Chapter 3 describes and analyses a number of discrete logarithm algorithms which are applicable to arbitrary finite cycle groups (so-called "generic algorithms"). These algorithms may be used to attack any of the primitives presented in Chapter 1. Some empirical observations from computer experiments using these algorithms are presented.

Chapter 4 is devoted to the problem of factoring polynomials over finite fields. The algorithms discussed here will be of crucial importance in a discrete logarithm algorithm which is discussed in the following chapter. Two well known factorisation algorithms are considered, the deterministic algorithm of Berlekamp and the probabilistic algorithm of Cantor and Zassenhaus.

Chapter 5 considers a class of non-generic discrete logarithm algorithms which can be applied to finite fields. These "index calculus" algorithms are the most efficient algorithms known for solving the discrete logarithm problem in the multiplicative group of a finite field. The general template of this class of algorithm is discussed, and then a number of specific instances are considered in detail.

Acknowledgements

First and foremost, my sincerest thanks are due to my supervisor, Bob Clarke. Bob has overseen the development of every stage of this thesis, regularly proofreading the work in progress and providing valuable advice. I am most fortunate to have had a supervisor with not only an inspiring understanding of the algebra in this thesis, but also a knowledge of modern cryptography and the relation between the two.

I would like to thank Distinguished Emeritus Professor Ron Mullin, from the University of Waterloo in Ontario, Canada. Professor Mullin took the time to correspond with me regarding my discovery of an error in a paper of his (and other authors) and confirmed my correction. Thanks are also due to my fellow honours student Edward Watts for his assistance in the early stages of locating this error. Ed verified many of my calculations and offered some helpful suggestions. Of course, Bob also confirmed my final conclusion and offered the advice to contact Mullin, for which I am grateful.

Thank you to Samuel Cohen, who proofread a draft copy of this thesis for mathematical errors, and to Kirsty Hill who did the same for spelling and grammatical errors. Their feedback was most helpful and is appreciated. If any errors of either type remain in this work, the responsibility is mine alone.

Finally, thank you to Kirsty for her patience and understanding as I spent an inordinate amount of this year hidden behind computers and journal papers.

Notation

Symbols

Symbol	Interpretation
$\mathbb{N}$	The set of *natural numbers*, $\{1, 2, 3, \ldots\}$.
$\mathbb{Z}$	The set of *integers*, $\{0, \pm 1, \pm 2, \pm 3, \ldots\}$.
$\mathbb{Z}^+$	The set of *non-negative integers*, $\{0, 1, 2, 3, \ldots\}$.
$\mathbb{Z}_n$	The ring of integers modulo n, $\{0, 1, 2, \ldots, n-1\}$.
$\mathbb{Z}_n^*$	The multiplicative group of integers which are units modulo n.
$\mathbb{F}_q$	The *finite field* (or *Galois field*) of order q.
$\text{char}(\mathbb{F}_q)$	The *characteristic* of the field $\mathbb{F}_q$.
R^*	The multiplicative group of the ring R.
$R[x]$	The ring of polynomials over the ring R, in the indeterminate x.
$\gcd(f(x), g(x))$	The unique monic polynomial of greatest degree dividing both $f(x)$ and $g(x)$.
$\langle \alpha \rangle$	The principal ideal generated by the ring element α.
$\square$	End of proof.

Pseudocode

Many algorithms are presented in this thesis. These algorithms are presented in the form of *pseudocode*; a ficticious programming language designed to make the operation of the algorithm clear without any of the confusing details which may be associated with a real programming language. The reader with even passing familiarity with any real language should find our pseudocode clear. We note the following here:

- Algorithm names are print in smallcaps, with arguments listed within parentheses and separated by commas. For example SQUAREFREE($f(x)$).
- Variable assignment is denoted using $\leftarrow$. For example, $x \leftarrow 42$ stores the value 42 in the variable x.
- Control statements are printed in bold. These include **do**, **For**, **If**, **Return**, **Unconditionally** and **While**. These statements act as they do in most real languages.

Contents

Preface **ii**

Acknowledgements **v**

Notation **vi**

1 Discrete Logarithms and Public Key Cryptography **1**
1.1 Discrete Logarithms in Finite Fields 1
1.2 Public Key Cryptography With Discrete Logarithms 4
1.2.1 Public Key Cryptography 4
1.2.2 Diffie Hellman Key Exchange 5
1.2.3 Elgamal Cryptosystem 6
1.2.4 Digital Signature Algorithm 7
1.3 Further Reading 9

2 Efficient Computation in Finite Fields **10**
2.1 Representations of Finite Fields 10
2.1.1 Representing Prime Order Fields with Integers 11
2.1.2 Representing Prime Power Order Fields with Polynomials 11
2.1.3 Further Reading 12
2.2 Fast Multiplication 12
2.2.1 Reduction of Integer to Polynomial Multiplication 13
2.2.2 The Classical Polynomial Multiplication Method 14
2.2.3 Karatsuba Polynomial Multplication 14
2.2.4 Fast Fourier Transform Polynomial Multiplication 16
2.2.5 Further Reading 23
2.3 Fast Exponentiation 23
2.3.1 Classical Exponentiation 23
2.3.2 Exponentiation by Squaring 24
2.3.3 Addition Chain Exponentiation 24
2.4 Construction of Irreducible Polynomials 26
2.4.1 Further Reading 29

3 Generic Discrete Logarithm Algorithms **31**
3.1 Trial Exponentiation 32
3.2 Shanks' Baby-Step Giant-Step method 32
3.3 Pollard's ρ-method 34
3.4 Pollard's λ-method (Kangaroo Method) 41

3.5 Pohlig-Hellman Method 46

4 Factorisation of Polynomials over Finite Fields 49
4.1 Introduction and Motivation 49
4.2 Some Partial Factorisations 50
4.2.1 Squarefree Factorisation 50
4.2.2 Distinct Degree Factorisation 53
4.3 Berlekamp's Algorithm 55
4.4 The Cantor-Zassenhaus Algorithm 60
4.5 Further Reading 64

5 Index Calculus Algorithms for Finite Fields 66
5.1 Generic Description 67
5.2 A Simple Index Calculus Method for $\mathbb{F}_{p^n}$ 68
5.3 An Improved Method for Fields $\mathbb{F}_{p^n}$ 75
5.4 Improved Methods for Fields of Characteristic 2, $\mathbb{F}_{2^n}$ 77
5.4.1 The Waterloo Work 77
5.4.2 Coppersmith's Work 79
5.5 A Simple Index Calculus Method for Fields $\mathbb{F}_p$ 80
5.6 An Improved Method for Some Fields $\mathbb{F}_p$ 81
5.7 Further Reading 85

A Computer Code 87
A.1 The GNU Multiple Precision Library 87
A.2 Pollard's ρ-method 87

Bibliography 90

Chapter 1

Discrete Logarithms and Public Key Cryptography

This thesis is concerned with a difficult computational problem in group theory. This problem is called the *discrete logarithm problem* and has been the subject of intensive research by the mathematical community for the past thirty years. The primary motivation for this research is the wide range of applications which the discrete logarithm problem has found in the area of cryptography. In particular, the difficulty of the discrete logarithm problem forms the basis of the security for many algorithms in *public key cryptography*, for performing tasks such as exchanging secret keys over public channels, providing confidentiality to communications between two parties with no prior acquaintence, and ensuring the authenticity of electronic messages. We are interested in the discrete logarithm problem as it is posed over a *finite field*.

This chapter is composed of two parts. First, we introduce the concept of discrete logarithms over finite fields and formally define the discrete logarithm problem. We also give some definitions from theoretical computer science which will be used throughout the rest of the thesis. In the second part, we introduce the paradigm of public key cryptography and present three algorithms based upon the discrete logarithm problem in finite fields.

1.1 Discrete Logarithms in Finite Fields

The primary algebraic setting of this thesis is the *finite field* (or *Galois Field*) of order q, which we denote $\mathbb{F}_q$. We briefly recall that $\mathbb{F}_q$ is a set of q elements equipped with two binary operations, multiplication and addition, with the properties that $\mathbb{F}_q$ forms an abelian group under addition (with identity 0) and the non-zero elements $\mathbb{F}_q^* = \mathbb{F} \setminus \{0\}$ form an abelian group under multiplication (with identity 1). We note that finite fields of order q exist only when q is of the form $q = p^n$, where p is a prime number and n is a natural number, and that there is a unique field of any such order.

We are interested in the concept of *discrete logarithms* on the finite field q (a concept which is in fact well defined for all finite cyclic groups), which are defined in analogy to the familiar logarithms of the real numbers. The key property of finite fields to recall here is that the multiplicative group $\mathbb{F}_q^*$ is

cyclic, i.e. there exists a field element $\alpha \in \mathbb{F}_q^*$ with the property that $\mathbb{F}_q^* = \langle\alpha\rangle = \{\alpha^i | i = 0, 1, 2, \ldots\}$. This element α is called a *generator* of $\mathbb{F}_q^*$ or a *primitive element* of $\mathbb{F}_q$.

Definition 1.1.1. Discrete Logarithm

Let α be a primitive element of the finite field $\mathbb{F}_q$. For any element $\beta \in \mathbb{F}_q^*$, the *discrete logarithm of β to the base α*, denoted $\log_\alpha(\beta)$ is the unique integer x in the set $\{0, 1, 2, \ldots, q-1\}$ such that $\alpha^x = \beta$.

The similarity to real logarithms is clear. We note that there are in fact an infinite number of $x \in \mathbb{Z}$ such that $\alpha^x = \beta$ for any $\beta \in \mathbb{F}_q^*$, since $\alpha^{x+k(q-1)} = \alpha^x$ for any $k \in \mathbb{Z}$. We explicitly define the discrete logarithm to be the least non-negative such integer. This allows us to cleanly state the following property of discrete logarithms, which the reader can easily verify and will recognise as the generalisation of a well known property of real logarithms.

Theorem 1.1.2. ***Discrete Logarithms of Products***

For any finite field $\mathbb{F}_q$, primitive element α and collection of elements $\beta_0, \beta_1, \ldots, \beta_{k-1} \in \mathbb{F}_q^*$ it is true that:

$$\log_\alpha \left(\prod_{i=0}^{k-1} \beta_i \right) \equiv \sum_{i=0}^{k-1} \log_\alpha(\beta_i) \pmod{q-1}. \tag{1.1}$$

In light of this result, we usually think of the discrete logarithms of $\mathbb{F}_q^*$ as being elements of the additive group $\mathbb{Z}_{q-1}$. We can then think of a discrete logarithm *function* which is an isomorphism $\log_\alpha : \mathbb{F}_q^* \to \mathbb{Z}_{q-1}$. It is the evaluation of this isomorphism in which we are interested. Precisely, we are concerned with the following problem.

Definition 1.1.3. Discrete Logarithm Problem

The *discrete logarithm problem* in a finite field is: Given a finite field $\mathbb{F}_q$, a generator α of $\mathbb{F}_q^*$ and an element $\beta \in \mathbb{F}_q^*$, compute the discrete logarithm $\log_\alpha(\beta)$.

Our consideration of the discrete logarithm problem (or DLP) is algorithmic in nature. We present well-defined sequences of algebraic operations, suitable for programming into a digital computer, (i.e. algorithms) which will solve the DLP. Some of these algorithms involve random variables. These are called *probabilistic algorithms*. Algorithms which do not involve random variables are called *deteministic algorithms*. We prove results which provide a measure of the amount of time and storage space required by these algorithms. In order to facilitate this style of exposition, we now present some elementary concepts regarding the analysis of algorithms. This subject is known as *computational complexity theory* and is one of the major disciplines of theoretical computer science. The ideas of complexity theory may be stated rigorously in terms of a well developed formalism. However, we forego this, instead giving simplified definitions which we feel are appropriate for use in what is intended as a primarily mathematical thesis with a computational bent. Most of the ideas of from

complexity theory discussed here are treated more formally in [37], and brief discussion is also given in most general cryptography references, for example [65, 47, 45].

Definition 1.1.4. Time and Space Complexity

The *time complexity* of an algorithm is the number of "steps" (instances of some elementary computation) which the algorithm must perform to solve a problem, expressed as a function of the input size (usually denoted n). The *space complexity* of an algorithm is the number of some elementary units of "storage" or "memory" which the algorithm requires, expressed in the same way. In the case of a probabilistic algorithm, the time and space complexities refer to the *expected* numbers of steps and units of storage.

In order to discuss the time and space complexity of algorithms, it is important to decide upon what shall be considered as an elementary computation and an elementary unit of storage. We do not define "computation" or "storage"; it is hoped that these concepts will be understood intuitively. For algorithms dealing mainly with finite fields, obvious choices for an elementary computation are a field addition and/or a field multiplication. Throughout this thesis, we shall consider an elementary computation to be the multiplication of two field elements. The number of field additions an algorithm requires thus has no influence on the algorithm's computational complexity. We feel that this choice is reasonable (and it is certainly common) because additions can usually be performed faster than multiplications on digital computers and because in most algorithms additions do not usually outnumber multiplications by an unreasonable amount. The elementary unit of storage is almost always taken to be a bit. With this measure, the size of an element of $\mathbb{F}_q$ is $\log q$[1].

We now define two classes of algorithm, classifying them by time complexity.

Definition 1.1.5. Polynomial Time Algorithm

An algorithm is said to be a *polynomial time algorithm* (alternatively, is said to *run in polynomial time*) if its time complexity is bounded above by a polynomial function of the input size.

Definition 1.1.6. Exponential Time Algorithm

An algorithm is said to be an *exponential time algorithm* (alternatively, is said to *run in exponential time*) if its time complexity is bounded above by an exponential function of the input size.

It is usual to identify running in polynomial time with being efficient and running in exponential time as being inefficient, even though this is not always the case. A problem is considered "easy" if it can be solved in polynomial time and "hard" if no polynomial time algorithm is known. At the time of writing, no polynomial time algorithm for the discrete logarithm problem is known. All of the algorithms we will see in this thesis for problems other than the DLP run in polynomial time.

We end our discussion of computational complexity by introducing some notation which is very commonly used to express the time and space complexities

[1] Unless otherwise specified, (non-discrete!) logarithms in this thesis are to base 2

of algorithms. This notation is most widely known as "big O" notation, but is also known as asymptotic notation or Landau notation.

Definition 1.1.7. "Big O" Notation

A function $f(n)$ is said to be $O(g(n))$ ("big O $g(n)$", or "order $g(n)$") if:

$$\limsup_{n\to\infty} \left| \frac{f(x)}{g(x)} \right| < \infty$$

If an algorithm has time complexity $O(n^k)$, it is common to say that the algorithm "runs in time $O(n^k)$" or "takes time (n^k)".

We have mentioned that there is no known polynomial time algorithm for solving the DLP (or, equivalently, evaluating the discrete logarithm isomorphism from $\mathbb{F}_q^*$ to $\mathbb{Z}_{q-1}$) and hence that the problem may be considered "hard". However, the inverse function of the discrete logarithm isomorphism is simply the exponentiation isomorphism from $\mathbb{Z}_{q-1}$ to $\mathbb{F}_q^*$ which maps x to α^x. This isomorphism can be trivially evaluated in polynomial time. Using loose language to describe a rigorously defined concept, a function which is easy to compute but hard to invert is called a *one-way function*. The existence of one-way functions as they are formally defined is an open problem in theoretical computer science, but exponentiation in a finite field is a well regarded candidate for being a one-way function. This is the reason why the computation of discrete logarithms has found relevance in the field of cryptography, where one-way functions are used in the construction of secure algorithms for performing various tasks. This cryptographic relevance is our primary motivation for studying the discrete logarithm problem, and is discussed in detail in the following section.

1.2 Public Key Cryptography With Discrete Logarithms

1.2.1 Public Key Cryptography

Until relatively recently in the long history of cryptography, its applications have been limited by the need for shared, pre-established secrets. If Alice and Bob wish to communicate securely, they must both have knowledge of some secret key which is used to encrypt and decrypt messages. The security of their communication relies upon the assumption that they are the only parties in possession of this key; if the key is known by an eavesdropper Eve, then Eve may read all communications. If Alice and Bob are separated by any substantial physical distance, the secure transfer of a secret key by a secure courier or other means can be difficult, expensive or even impossible. Further more, if each member of a large party wishes to be able to communicate privately with every other member, each member is required to know and keep secret a large number of keys. These factors, often collectively referred to as the *key distribution* or *key management problem*, greatly restricted the situations in which cryptography could be used.

In 1976, a new paradigm in cryptography arose which removed these restrictions and led to an explosion in the use of cryptography. This new cryptography removes the need for users to share a pre-established secret. Instead, each user

possesses a so-called *key pair*, consisting of one *public key* and one *private key*. As the names suggest, the public key may be made publically available (e.g. published on a website, attached to emails) while the private key must be kept secret. Messages which are encrypted using a user's public key can only be decrypted using that user's private key. Cryptography following this paridigm is referred to as *public key cryptography*, the "old" alternative being *private key cryptography* (the respective alternate names *asymmetric* and *symmetric cryptography* are also used).

Using public key cryptography, Alice and Bob need only exchange their public keys, without concern as to eavesdropping, before they can communicate privately. This allows secure communication between parties with no prior acquaintence or communication and without the need for a secure means of key transfer. Further, in a party of n users desiring pairwise privacy, each user must keep only one secret key and n keys must be made publically available to all users. By contrast, using private key cryptography each user must keep $n-1$ secret keys, with a total of $n^2 - n$ secret keys having to be generated and securely transferred. The difference in difficulty of key management is thus quite significant. With the advent of public key cryptography, cryptography moved out of its former exclusive domain of government or military use and is today widely used by corporations of all sizes as well as by individuals.

The author wishes to avoid giving the mistaken impression that private key cryptography is today obselete; it is, in fact, still widely used. This is largely due to the fact that public key cryptography tends to be substantially less efficient than private key cryptography, both in terms of the time required to perform an encryption and the amount of space consumed by the encrypted message compared to the original message. For this reason, it is common to use a public key cryptosystem to securely transfer a secret key and then use this secret key to secure further communications using a private key cryptosystem. This strategy is termed *hybrid encryption*.

In this chapter we present three cryptographic primitives which are considered to be public key cryptography. Each primitive allows two parties, Alice and Bob, with no prior acquaintance to perform three different tasks: Securely exchange a secret key over an insecure channel, encrypt and decrypt messages for confidential communication, and sign messages and verify these signatures to ensure messages do not originate from imposters and are not tampered with in transit.

1.2.2 Diffie Hellman Key Exchange

In 1976, W. Diffie and M. E. Hellman proposed an algorithm [22] which allows two parties to securely negotiate a secret key over an insecure channel, the security of which depends upon the difficulty of computing discrete logarithms. This *Diffie-Hellman key exchange* is in wide use today: it is incorporated into most web browsers and and is a part of many protocols and standards for securing internet traffic, including the Secure Sockets Layer (SSL), Transport Security Layer (TSL) and IPsec. Briefly, the algorithm, as executed by two parties Alice and Bob, is as follows:

Diffie-Hellman Key Exchange

1. Alice and Bob use the insecure channel to agree upon a finite field $\mathbb{F}_q$ and a primitive element α of $\mathbb{F}_q^*$.
2. Alice and Bob each select a random integer, x and y respectively, from $\{2, \ldots, q-2\}$ which they keep to themselves.
3. Alice and Bob compute α^x and α^y respectively and send these to each other over the insecure channel.
4. Alice computes $(\alpha^y)^x$ and Bob computes $(\alpha^x)^y$.

At the end of this protocol, both parties now share the group element $\alpha^{xy} = \alpha^{yx}$. This can be used to derive, say, a shared secret key for a private key cryptosystem. An eavesdropper Eve knows only α^x and α^y. It is clear that she must learn either x or y to obtain the shared secret. This can only be accomplished from the information she knows by computing $\log_\alpha(\alpha^x)$ or $\log_\alpha(\alpha^y)$. Thus, if the DLP in $\mathbb{F}_q$ is infeasible then the security of the key exchange is assured.

1.2.3 Elgamal Cryptosystem

In 1985, E. Elgamal proposed a public key cryptosystem and a signature scheme [24], the security of both of which depends upon the difficulty of computing discrete logarithms. Today the *Elgamal cryptosystem* is the default public key cipher for the widely-used GNU Privacy Guard (GPG) cryptographic software suite. Briefly, the cryptosystem, as used by Alice to communicate confidentially with Bob, is as follows:

Elgamal Cryptosystem

1. Key setup:
   (a) Bob selects a finite field $\mathbb{F}_q$ and a primitive element α of this field.
   (b) Bob selects a random integer x from $\{2, \ldots, q-1\}$. This is his *private key.*
   (c) Bob computes $\beta = \alpha^x$ and publishes the values $\mathbb{F}_q, \alpha, \beta$. This is his *public key.*
2. Encryption:
   (a) Alice encodes the message she wishes to send Bob as an element of $\mathbb{F}_q$, m, using a public encoding scheme.
   (b) Alice selects a random integer y from $\{2, \ldots, q-1\}$. This integer should be kept secret and used for only one encryption session.
   (c) Alice computes $c_1 = \alpha^y$ and $c_2 = \beta^y m$ (α and β are known from Bob's public key).
   (d) Alice sends Bob the ciphertext (c_1, c_2).

3. Decryption:
   (a) Bob receives the ciphertext (c_1, c_2).
   (b) Bob computes $c_1^{-x} c_2 = (\alpha^y)^{-x} \beta^y m = \alpha^{-xy} \alpha^{xy} m = m$.
   (c) Bob decodes the message from m, according to the public scheme used by Alice.

It is clear that an Eavesdropper requires knowledge of the private key x in order to decrypt the message. To extract x from the information she has requires computing $\log_\alpha(\beta)$. Thus, if the DLP in $\mathbb{F}_q$ is infeasible, the privacy of Alice and Bob's communication is assured.

1.2.4 Digital Signature Algorithm

In the same paper describing his cryptosystem, Elgamal proposed a digital signature scheme [24] which also based its security upon the infeasibility of discrete logarithms. While this original algorithm is now considered insecure, it has inspired many similar algorithms which are considered secure. These signatures are said to belong to the family of *Elgamal signature schemes*. One scheme from this family was in 1991 proposed and in 1993 accepted as a United States federal standard by the National Institute for Standards Technology. It is known as the *digital signature algorithm*, or DSA (the standard outlining aspects of its use is called the digital signature standard or DSS. These terms are sometimes used interchangably). The latest revision of the DSS, as published by NIST can be found at [55].

The DSA is somewhat more complicated than the previously presented cryptographic algorithms, for two reasons. Firstly, while the algorithm as a whole is set in a large prime order finite field $\mathbb{F}_p$, many computations take place within subgroup of $\mathbb{F}_q^*$ with prime order q. This idea is originally due to Schnorr [66]. It results in a smaller signature size and faster operation, without affecting the security of the algorithm. The DSS mandates the relative sizes of p and q for implementations of the DSA which comply to the standard. Secondly, like all digital signature schemes the DSA requires the use of a *hash function*. For our purposes, a hash function is any function $H : \mathbb{Z}_p \to \mathbb{Z}_q$ with the properties that distinct values in $\mathbb{Z}_p$ map to equivalent values in $\mathbb{Z}_q$ very rarely and that given an evaluation $H(\alpha)$ of the hash function, computing the preimage α is computationally infeasible. The DSS specifies an appropriate hash function for the specified values of p and q.

We now briefly describe the functionality of the DSA.
Digital Signature Algorithm

1. Key Generation
   (a) Alice selects a large prime order finite field $\mathbb{F}_p$.
   (b) Alice selects a prime q such that $q|p-1$.
   (c) Alice computes $g = h^{(p-1)/q} \pmod p$, where $h \in \mathbb{F}_p$ is chosen so that $g \not\equiv 1 \pmod p$.
   (d) Alice selects a random integer x from $\{2, \ldots, q-1\}$. This is her *private key*.

(e) Alice computes $y = g^x \pmod p$ and publishes the four parameters p, q, g, y. The element y is considered Alice's *public key.* The parameters p, q, g may be shared among a group of users.

2. Message Signing

   (a) Alice encodes her message as $m \in \mathbb{F}_p$.

   (b) Alice selects a random integer k from $\{2, \ldots, q-1\}$. This integer should be kept secret and used for only one signature session.

   (c) Alice computes $r = (g^k \pmod p) \pmod q$.

   (d) Alice computes $s = k^{-1}(H(m) + xr) \pmod q$.

   (e) Alice sends the values r, s as the *signature* of m.

3. Signature Verification

   (a) Bob receives the message and signature components m', r', s'.

   (b) Bob computes $w = (s')^{-1} \pmod q$.

   (c) Bob computes $u_1 = H(m')w \pmod q$.

   (d) Bob computes $u_2 = r'w \pmod q$.

   (e) Bob computes $v = (g^{u_1} y^{u_2} \pmod p) \pmod q$.

   (f) If $v = r'$, Bob accepts the message m' as authentic. Otherwise, the message is considered invalid.

We now show the correctness of the DSA, i.e. that if the received message and signature components are equal to the sent message and signature components then v as computed above is equal to r' and thus the message is correctly verified as authentic.

We begin by observing that, since the message is authentic:

$$\begin{aligned} w &\equiv (s')^{-1} \equiv s^{-1} \pmod q, \\ u_1 &\equiv ((H(m')w) \equiv H(m)w \pmod q, \\ u_2 &\equiv ((r')w) \equiv rw \pmod q \end{aligned}$$

We thus have:

$$\begin{aligned} v &\equiv (g^{u_1} y^{u_2} \pmod p) \pmod q \\ &\equiv (g^{H(m)w} y^{rw} \pmod p) \pmod q \\ &\equiv (g^{H(m)w} g^{xrw} \pmod p) \pmod q \\ &\equiv (g^{(H(m)+xr)w} \pmod p) \pmod q \end{aligned}$$

Now,

$$s \equiv k^{-1}(H(m) + xr) \pmod q,$$

so

$$w \equiv s^{-1} \equiv k(H(m) + xr)^{-1} \pmod q.$$

From this it follows that:

$$(H(m) + xr)w \equiv (H(m) + xr)k(H(m) + xr)^{-1} \equiv= k \pmod{q},$$

and so:

$$v \equiv (g^k \pmod{p}) \equiv r \equiv r' \pmod{q},$$

as we require.

It is clear that for a would-be imposter Mallory to forge Alice's digital signature, she requires knowledge of Alice s secret key x. To extract x from the information she has requires computing $\log_\alpha(\beta)$. Thus, if the DLP in $\mathbb{F}_q$ is infeasible, the authenticity of Alice's signature is assured.

1.3 Further Reading

We note that the DLP is well defined not just for finite fields, but in fact for arbitrary finite cyclic groups; this generalisation is quite obvious. While the finite field DLP is certainly the most widely used in cryptography, another instance of the DLP has also attracted considerable attention from cryptographers. In 1987, N. Koblitz [36] proposed the idea of using the abelian group of points on an elliptic curve over a finite field for DLP based cryptography. This is termed *elliptic curve cryptography*, or ECC, and remains a major area of cryptographic research. The most efficient known algorithms for solving the DLP in a finite field (the index calculus algorithms, see Chapter 5) do not seem to extend to solving the DLP in an elliptic curve group. The best known algorithms for the elliptic curve DLP are the generic algorithms of Chapter 3, which have comparatively poor performance. For this reason, the elliptic curve DLP is infeasible for groups much smaller than finite fields for which the DLP is infeasible. This can lead to more efficient cryptographic systems.

We also note that the DLP is not the only difficult problem which is used as the basis for cryptographic primitives. The problem of *integer factorisation* is also believed to be computationally infeasible for sufficiently large integers. This problem is the basis for the RSA cryptosystem [64], which was in fact the first public key cryptosystem developed. Other problems have been proposed as bases for cryptographic primitives, although none have achieved as widespread use as the DLP and integer factorisation problems. These include the problem of distinguishing between different representations of identical linear codes [46], various problems in combinatorial group theory using braid groups [2, 16], and various computational problems on integer lattices [28].

Chapter 2

Efficient Computation in Finite Fields

Before we consider any of the computational problems or algorithms for solving these problems which are presented in this thesis, we devote a chapter to some preliminary considerations. Here we discuss how finite fields may be represented for computation and how some very basic computations can be performed in these representations. In particular, we consider the representation of finite fields as rings of integers and of polynomials, and discuss methods for efficient arithmetic and computation of greatest common divisors (GCDs) in these representations. The details discussed in this chapter will generally not be referred to in later chapters, which are concerned with higher level computations and not how things work "under the hood". This chapter may be skipped without affecting the readability of subsequent chapters.

2.1 Representations of Finite Fields

Finite fields are abstract structures. In order to actually perform computations within them, we need a representation [1] of the field; a concrete description of all the field elements and how they interact under the binary operations of addition and multiplication. Different representations may have different advantages and disadvantages in terms of the amount of memory required to represent a field element on a digital computer and the number of processor cycles required to perform operations.

We will use one representation for prime order fields and one representation for prime power order fields for the vast majority of this thesis, since these two are intuitive, are likely to be familiar to the reader and are easily implemented on a computer. They are by no means the only or the most efficient representations for these fields. An alternative representation of prime order fields is given in Chapter 5, since the representation of fields is of particular relevance to the algorithm discussed there. References to more efficient representations are given at the end of the section.

[1] Our use of the term "representation" here is distinct from its meaning of a homomorphic map from a group to a set of linear transformations of a vector space.

2.1.1 Representing Prime Order Fields with Integers

The following theorem provides a simple representation for finite fields of prime order, which we will refer to as the *integer representation*.

Theorem 2.1.1. Integer Representation of Prime Order Fields

The ring of integers modulo a prime number p, $\mathbb{Z}_p$, is a representation of the finite field of order p, $\mathbb{F}_p$.

This result is well known and so the proof is omitted.

This representation has many virtues and is almost always appropriate for use in applications. The operations of modular integer addition and multiplication are present by default in almost every computer programming language, so that prime order fields can be implemented with no extra work. Algorithms for fast multiplication of integers will be discussed in Section 2.2. Division can be performed by using the well-known extended Euclidean algorithm to compute multiplicative inverses and then performing a multiplication. Greatest common divisors can be computed using the Euclidean algorithm.

2.1.2 Representing Prime Power Order Fields with Polynomials

The following theorem provides a simple representation for finite fields of prime power order, which we will refer to as the *polynomial representation*.

Theorem 2.1.2. Polynomial Representation of Prime Power Order Fields

Let p be a prime number and $k \in \mathbb{N}$. Let $p(x) \in \mathbb{F}_p[x]$ be an irreducible polynomial of degree k over the finite field of p elements. Then the factor ring:

$$\frac{\mathbb{F}_q[x]}{\langle p(x) \rangle}$$

is a representation of the finite field of order p^n, $\mathbb{F}_{p^n}$.

This result is well known and so the proof is omitted.

Addition and multiplication in this representation are the familiar polynomial addition and multiplication from high school, with integer arithmetic performed in the field $\mathbb{F}_p$ and the final result reduced modulo the irreducible polynomial $p(x)$. Multiplication of polynomials of degree at most n can be computed in the obvious manner requiring $O(n^2)$ multiplications in $\mathbb{F}_q$. Algorithms for faster multiplication of polynomials will be discussed in Section 2.2. Since polynomial rings form Euclidean domains with the norm of a polynomial being its degree, inversion of field elements may be performed by using the extended Euclidean algorithm. Modular reduction of polynomials where all degrees are at most n can be performed in time $O(n^2)$ using polynomial long division. The greatest common divisor of two polynomials (the unique monic polynomial of highest degree dividing both polynomials) each of degree at most n can be computed using the Euclidean algorithm. The cost of computing a GCD of two

polynomials of degree at most n in this way is $O(n^2)$ (see [50]). A faster algorithm described by Moenck in [50] can compute GCDs with cost $O(nM(n))$ if two degree n polynomials can be multiplied in time $M(n)$.

The polynomial representation is particularly efficient for prime power order fields of characteristic 2, $\mathbb{F}_{2^k}$. In this representation, a field element is a polynomial over $\mathbb{F}_2$, which can be represented on a computer as a binary integer of as many bits as the degree of the polynomial. For example, the degree 8 polynomial $1 + x^2 + x^5 + x^6 + x^8$ may be represented as either of the 8 bit integers $101100101 = 357$ or $101001101 = 333$. With this representation, addition of polynomials becomes simply the logical XOR operation, which is present in most programming languages by default. For example, the polynomials $1 + x^2 + x^4$ and $x^2 + x^3$ in $\mathbb{F}_2[x]$ have binary representations 10101 and 01100 and sum to $1 + x^3 + x^4$, which has binary representation $11010 = 10101 \oplus 01100$. Further, multiplication of polynomials can be performed using a combination of XORs and bit-shifts. Both of these operations are among the fastest a modern CPU can perform.

From a practical perspective, implementing this representation requires the ability to find an appropriate irreducible polynomial $p(x)$ to use as a modulus. The problem of generating irreducible polynomials of a specific degree over a certain finite field will be considered in Section 2.4.

2.1.3 Further Reading

It is a well known result that any finite field $\mathbb{F}_{q^n}$ forms an n-dimensional vector space over the field $\mathbb{F}_q$. Any basis for this vector space provides us with a representation for $\mathbb{F}_{q^n}$. In the polynomial representation we gave for $\mathbb{F}_{p^n}$, the basis for $\mathbb{F}_{p^n}$ over $\mathbb{F}_p$ was the set $\{1, x, x^2, \ldots, x^{n-1}\}$. This basis is not optimal from the perspective of computation, but it is widely used because it is very easy to work with, both in theory and in implementations.

There exists a class of bases for finite fields $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ called *normal bases*. Using these bases it is possible to perform operations of field multiplication and exponentiation with subtantially greater efficiency than the methods we will see later using our given representations. Normal bases which minimise the complexity of these operations are termed *optimal normal bases*. A detailed discussion of the theory of normal bases and the construction of optimal normal bases can be found in the book of Blake *et. al.* [9].

2.2 Fast Multiplication

One of the most basic computational tasks in a finite field is multiplication, which is usually a more expensive operation to perform than addition. Using the integer and polynomial representations introduced in Section 2.1, the task of field multiplication is either the task of modular integer multiplication or modular polynomial multiplication. For the sake of simplicity and brevity, we consider here simply the task of multiplying either integers or polynomials. Modular reduction in each case can be performed afterwards simply by performing a division and discarding the quotient, keeping the remainder as the result. This is certainly not optima; In the case of integers, a technique due

to P. Montgomery [52] is a well known and efficient alternative which does not separate multiplication and reduction.

2.2.1 Reduction of Integer to Polynomial Multiplication

We show here how the multiplication of large integers can be reduced to the multiplication of polynomials and hence can be performed using the polynomial multiplication methods which comprise the remainder of this section, which are also used for multiplication in prime power fields $\mathbb{F}_{p^n}$.

Theorem 2.2.1. ***Reduction of Integer Multiplication to Polynomial Multiplication***

The problem of multiplying two integers n and m is reducible to the problem of multiplying two polynomials.

Proof: Suppose we wish to multiply two large integers n and m. We may write these integers as their expansion in some base, e.g. their binary or ternary expansions. Suppose n and m have the following expansions in base $b \in \mathbb{N}$:

$$n = \sum_{i=0}^{r} n_i b^i \quad \text{and} \quad m = \sum_{i=0}^{s} m_i b^i.$$

Then we can consider n and m as the values of two polynomials:

$$N(x) = \sum_{i=0}^{r} n_i x^i \quad \text{and} \quad M(x) = \sum_{i=0}^{s} m_i x^i,$$

evaluated at the point b, i.e. $n = N(b), m = M(b)$. Further, it is clear that the product nm is the value of the product of these two polynomials evaluated at b. Hence two integers can be multiplied for the cost of one polynomial multiplication and one polynomial evaluation.

□

The remainder of this section is dedicated to the study of algorithms for the fast multiplication of polynomials. Any of these may be used to construct an algorithm for the fast multiplication of integers, using the above observation. They are also used to provide fast multiplication of elements of prime power order finite fields which are represented using the polynomial representation.

We consider the general problem of multiplying two polynomials $f(x)$ and $g(x)$ with degree n where:

$$f(x) = \sum_{i=0}^{n} a_i x^i \quad \text{and} \quad g(x) = \sum_{i=0}^{n} b_i x^i,$$

and

$$f(x)g(x) = h(x) = \sum_{i=0}^{n} c_i x^i = \sum_{i=0}^{n} \sum_{\substack{0 \le j \le k \le n \\ j+k=i}} a_j b_k x^i. \tag{2.1}$$

Note that if two polynomials to be multiplied are not of equal degree, as is our assumption above, the polynomial of least degree can be made into a polynomial of degree equal to that of the other polynomial by the prepending of an appropriate number of higher power terms with zero coefficients. Considering the problem of multiplying two equal degree polynomials allows us to express the time complexity of an algorithm in terms of a single input parameter, the common degree n.

In general we do not specify the ring R which the polynomials are defined over, as this does not affect the implementation or analysis of the algorithm. Typically, the ring is the non-negative integers $\mathbb{Z}^+$ if we are multiplying large integers using the observation of Theorem 2.2.1, or a finite field of prime order $\mathbb{F}_p$ if we are multiplying two polynomials in the polynomial representation of a prime power order field $\mathbb{F}_{p^n}$. The exception to this is in our discussion of polynomial multiplication using the Fast Fourier Transform, discussed in Section 2.2.4, where the ring may be the complex numbers $\mathbb{C}$, the ring of integers modulo some integer $\mathbb{Z}_N$ or a finite field of some appropriate order $\mathbb{F}_{p^n}$.

2.2.2 The Classical Polynomial Multiplication Method

The method which we call "classical"[2] is the simple method used by school students to multiply polynomials or to expand bracketed multiplications: The leading term of the first polynomial is successively multiplied by each term in the second polynomial, then the same is done for the next term of the first polynomial, until each possible cross-term has been computed, at which stage like terms are collected. Somewhat more directly and formally, the classical method involves direct computation of the coefficients of $h(x)$ via the final equality of (2.1).

This requires the multiplication of n^2 coefficients in R. This gives the classical method of polynomial multiplication a time complexity of $O(n^2)$. We will see that this can be improved upon with more elaborate methods, however it should be noted that the asymptotically faster algorithms usually involve some overhead which means that they only become faster than the classical method for polynomials of degree higher than some threshold the *crossover point*. For some fast algorithms the crossover point can be rather high, and thus classical polynomial multiplication still has a place as the algorithm of choice when only low degree polynomials are to be multiplied.

2.2.3 Karatsuba Polynomial Multplication

Theorem 2.2.2. *There exists a deterministic algorithm which multiplies two degree n polynomials over $\mathbb{F}_q^*$ using $O(n^{1.5850})$ multiplications in $\mathbb{F}_q^*$.*

The algorithm which constitutes proof of this theorem was published by A. Karatsuba [3] in 1962 [34]. An English-language treatment is given by R. Moeck in [51] and our treatment is based upon this. Surprisingly, this relatively recent algorithm appears to be the first published algorithm for multiplying polynomials which improves on the $O(n^2)$ complexity on the classical multiplication.

[2] Our "classical method" goes by several names in the literature, including *gradeschool method*, *high school method*, *pencil-and-paper method*, and *die Schulmethode*.

[3] The Russian name Karatsuba is sometimes transliterated at Karacuba in the literature.

The method assumes that the common degree of the polynomials n is even - if this is not the case for the polynomials as given, it can easily be made so by appending a single higher power term with coefficient zero. With this done, the even-degree polynomial $f(x)$ can be written in the following form, which we term the Karatsuba decomposition:

Definition 2.2.3. The *Karatsuba decomposition* of $f(x)$ is:

$$\begin{aligned} f(x) &= a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} + a_n x^n \\ &= \sum_{i=0}^{n/2} a_i x^i + x^{n/2} \sum_{i=1}^{n/2} a_{n/2+i} x^i \\ &= f_0(x) + x^{n/2} f_1(x), \end{aligned}$$

where the implicitly defined polynomials $f_0(x)$ and $f_1(x)$ each have degree $n/2$.

The key to Karatsuba's method is the following observation:

Lemma 2.2.4. *Let $f(x)$ and $g(x)$ have Karatsuba decompositions $f(x) = g_0(x) + x^{n/2}g_1(x)$ and $g(x) = g_0(x) + x^{n/2}g_1(x)$. Then, omiting indeterminates for notational clarity, we have:*

$$\begin{aligned} fg &= \left(f_0 + x^{n/2} f_1\right)\left(g_0 + x^{n/2} g_1\right) \\ &= f_0 g_0 + x^{n/2}(f_0 g_1 + f_1 g_0) + x^n f_1 g_1 \\ &= (1 + x^{n/2} - x^{n/2}) f_0 g_0 + x^{n/2}(f_0 g_1 + f_1 g_0) + (x^n + x^{n/2} - x^{n/2}) f_1 g_1 \\ &= (x^{n/2} + 1) f_0 g_0 - x^{n/2}(f_1 - f_0)(g_1 - g_0) + (x^n + x^{n/2}) f_1 g_1 \end{aligned}$$

Proof of Theorem 2.2.2:
Consider the following algorithm:

Algorithm 2.2.5. KARATSUBA($f(x), g(x)$)

Input: Polynomials $f(x), g(x) \in \mathbb{F}_q[x]$.
Output: $f(x)g(x)$.

---

1. **If** $\deg(f(x)) = \deg(g(x)) > 0$ do:
   (a) **If** $\deg(f(x)) = \deg(g(x))$ is odd **do**:
      i. $f(x) \leftarrow f(x) + 0x^{n+1}$.
      ii. $g(x) \leftarrow g(x) + 0x^{n+1}$.
   (b) **Return** $(x^{n/2} + 1)$KARATSUBA$(f_0(x), g_0(x)) - \ldots$
   $\ldots x^{n/2}$KARATSUBA$(f_1(x) - f_0(x), g_1(x) - g_0(x)) + \ldots$
   $\ldots (x^n + x^{n/2})$KARATSUBA$(f_1(x), g_1(x))$.
2. **Else do**:
   (a) **Return** $a_0 b_0$.

The behaviour of this algorithm is clear. It uses Lemma 2.2.4 recursively to continuously break a polynomial product down into three smaller products; If KARATSUBA is called with arguments $f(x), g(x)$ of degree $n > 0$, then the three calls it makes to itself are given arguments of degree $n/2$. Eventually the polynomials are broken down into constants which are then simply multiplied.

We note that the various factors involving x^n and $x^{n/2}$ in front of these products can be accounted for by simply increasing the powers of some terms and performing some addition of coefficients. No coefficient multiplication is required and so the computational effort involved is considered negligible in our analysis. To find an expression for the complexity of Karatsuba's method, we use a recurrence relation. When multiplying two polynomials of degree n, our recursive algorithm calls itself to evaluate 3 products of polynomials of degree $n/2$. If Karatsuba's method requires $T(n)$ multiplications to multiply two degree n polynomials then it is clear that $T(n)$ satisfies $T(n) = 3T(n/2)$. To solve this recurrence relation, we seek a solution of the form $T(n) = n^c$. Substituting this into the recurrence relation and taking logarithms to base 2 we see that:

$$\begin{aligned} & kn^c && = 3k(n/2)^c \\ & && = 3kn^c 2^{-c} \\ \Rightarrow \quad & \log(n^c) && = \log(3n^c 2^{-c}) \\ & c\log(n) && = \log(3) + c\log(n) - c\log(2) \\ \Rightarrow \quad & c && = \log(3) \simeq 1.5850. \end{aligned}$$

So Karatsuba's recursive algorithm requires $O(n^{1.5850})$ multiplications in R to multiply two polynomials of degree n, as opposed to $O(n^2)$ multiplications for the classical method.

□

2.2.4 Fast Fourier Transform Polynomial Multiplication

In this section we discuss the manner in which a generalisation of the fast Fourier transform (FFT) from its familiar setting of the complex numbers $\mathbb{C}$ to more general rings, including finite fields, can be used to construct fast algorithms for polynomial multiplication.

The ideas discussed here originated with the work of V. Strassen in 1968, which were not published. Strassen later collaborated with A. Schönhage to improve his method and the two published an integer multiplication algorithm in 1971 [68] which remains the asymptotically fastest algorithm known to date. Their method uses the reduction mentioned in Theorem 2.2.1 as well as the FFT and will be discussed in this section. J. M. Pollard published a paper in the same year [58] which generalised the FFT to certain finite fields and showed how it could be used to achieve, among other things, fast multiplication of polynomials over these fields.

We begin with some preliminaries concerning Fourier transforms.

Definition 2.2.6. Discrete Fourier Transform

Let R be a commutative ring with identity and let n be a unit in R, i.e. division by n is possible. Let $A = (a_0, a_1, \ldots, a_{n-1}) \in R^n$ be a vector of n

elements of R and let ω_n be an n-th primitive root of unity[4] in R. Then we define the (n-th order) *Discrete Fourier Transform* (DFT) of A (with respect to the root ω_n) to be the vector $\hat{A} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_n) \in R^n$ defined by:

$$\hat{a}_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \quad r = 0, 1, \ldots, n-1.$$

The *inverse DFT* of a vector A is the vector$\hat{A}$ defined by:

$$\hat{a}_j = \frac{1}{n} \sum_{k=0}^{n-1} a_k \omega_n^{-jk}, \quad r = 0, 1, \ldots, n-1.$$

We denote the DFT of a vector A by $\mathcal{F}(A)$ and the inverse DFT of A by $\mathcal{F}^{-1}(A)$.

This concept is most familiar in the case where $R = \mathbb{C}$, the field of complex numbers, and the n-th primitive root of unity is $\omega_n = e^{2\pi i/n}$, but most of the known results about this complex DFT generalise readily to arbitrary rings of the type described above, and these are more useful for our present purposes. The following result is fundamental:

Theorem 2.2.7. ***Relation Between DFT and inverse DFT***

For any vector A which has a well-defined DFT, the DFT and inverse DFT are related by:

$$\mathcal{F}^{-1}(\mathcal{F}(A)) = A \quad \textit{and} \quad \mathcal{F}(\mathcal{F}^{-1}(A)) = A$$

Proof: We prove the first equality, the proof of the second one being similar. By definition, the j-th component of $\mathcal{F}^{-1}(\mathcal{F}(A))$, say b_j, is:

$$b_j = \frac{1}{n} \sum_{k=0}^{n-1} \left(\sum_{l=0}^{n-1} a_l \omega_n^{kl} \right) \omega_n^{-jk} = \frac{1}{n} \sum_{l=0}^{n-1} \left(a_l \sum_{k=0}^{n-1} \omega_n^{k(l-j)} \right). \tag{2.2}$$

We consider the innermost sum, i.e. that indexed by k. When $l = j$, this sum is:

$$\sum_{k=0}^{n-1} \omega_n^{k(j-j)} = \sum_{k=0}^{n-1} \omega_n^0 = \sum_{k=0}^{n-1} 1 = n.$$

When $l \neq j$, the sum is:

$$\sum_{k=0}^{n-1} \omega_n^{k(l-j)} = \sum_{k=0}^{n-1} (\omega_n^{l-j})^k$$

We recognise this as a finite geometric series in ω_n^{l-j}, and using the well known formula for such sums as well as the fact that $\omega_n^n = 1$ we see that the value of the sum is:

$$\frac{1 - (\omega_n^{l-j})^n}{1 - \omega_n^{l-j}} = \frac{1 - (\omega_n^n)^{l-j}}{1 - \omega_n^{l-j}} = \frac{1 - 1^{l-j}}{1 - \omega_n^{l-j}} = 0.$$

So the sum can be replaced by $n\delta_{lj}$, where δ is the Kronecker delta, and (2.2) becomes:

$$b_j = \frac{1}{n} \sum_{l=0}^{n-1} a_l n \delta_{lj} = \sum_{l=0}^{n-1} a_l \delta lj = a_j,$$

[4]Recall that a ring element α is a primitive n-th primitive root of unity if $\alpha^n = 1$ and $\alpha^m \neq 1$ for any $m < n$, i.e. α has multiplicative order n.

which is what we require.

□

We now provide the link between the DFT and polynomial multiplication which is the basis of this multiplication method. The key concept is that of a cyclic convolution, defined here:

Definition 2.2.8. Cyclic Convolution

Let R be a ring. If $A = (a_0, a_1, \ldots, a_{n_1}), B = (b_0, b_1, \ldots, b_{n-1}) \in R^n$, then the *cyclic convolution* of A and B, denoted $A * B$, is the vector $C = (c_0, c_1, \ldots, c_{n-1})$ defined by:

$$c_j = \sum_{k=0}^{n-1} a_k b_{j-k},$$

where the subscript $j - k$ is computed modulo n.

We now make an observation relating polynomial products to cyclic convolutions:

Theorem 2.2.9. Polynomial Products as Cyclic Convolutions

For any ring R and any polynomial $f(x) \in R[x]$ of degree n define the coefficient vector *of $f(x)$ to be the vector $F \in R^{n+1}$ whose components are the coefficients of $f(x)$, ordered so that the first component of F is the constant term of $f(x)$. Let $f(x)$ and $g(x)$ be two particular polynomials in $R[x]$ of degree n. Define coefficient vectors F and G of $f(x)$ and $g(x)$, respectively, of length $2n + 1$ by considering $f(x)$ and $g(x)$ to be polynomials of degree $2n$ with zero coefficients for the n highest power terms. Then the cyclic convolution $F * G$ is the coefficient vector of the polynomial product $f(x)g(x)$.*

Proof: The j^{th} term of the cyclic convolution $F * G$, say C_j, is given by:

$$C_j = \sum_{k=0}^{2n} a_k b_{j-k},$$

with $j - k$ computed modulo $2n$. Since we have "padded" the coefficient vector F to length $2n + 1$ with zeros, $f_k = 0$ for $k > n$ and this becomes:

$$C_j = \sum_{k=0}^{n} a_k b_{j-k}. \tag{2.3}$$

If $j < n$ we split this into two sums as follows:

$$C_j = \sum_{k=0}^{j} a_k b_{j-k} + \sum_{k=j+1}^{n} a_k b_{j-k}.$$

In the first of these sums, $k \leq j$ so $0 \leq j - k \leq j < n < 2n$, so reduction modulo $2n$ has no effect and we may interpret the subscript $j - k$ as being "exact". In the second sum, $k > j$ so $j - k < 0$ and the reduction modulo $2n$ is important.

As k ranges from $j+1$ to n, $j-k$ ranges from -1 to $j-n$ and thus the reduction of $j-k$ modulo $2n$ ranges from $2n-1$ to $j+n$. The values in this range are always $> n$, so, since G was also padded with zeros, $b_{j-k} = 0$ and the second sum vanishes. We are left with:

$$C_j = \sum_{k=0}^{j} a_k b_{j-k},$$

where the $j-k$ computed exactly. This is, in fact, equivalent to:

$$C_j = \sum_{\substack{0 \le i \le k \le n \\ i+k=j}} a_i b_k, \tag{2.4}$$

which is the coefficient of the j-th power term of $f(x)g(x)$, exactly what we want.

If $j \geq n$, then the splitting into two sums is not needed. Our analysis of the first sum in the $j < n$ case applies directly to (2.3), since $k \leq j$, and so (2.3) is again the desired sum.

□

Thus we see that the problem of computing polynomial products can be reduced to the problem of computing cyclic convolutions. The following theorem provides the first step towards showing how this can be done efficiently. In this theorem, the product of two vectors A and B should be understood as a component-wise multiplication, i.e. if $A = (a_0, a_1, \ldots, a_{n-1})$ and $B = (b_0, b_1, \ldots, b_{n-1})$ then $AB = (a_0b_0, a_1b_1, \ldots, a_{n-1}b_{n-1})$.

Theorem 2.2.10. ***Convolution Theorem***

If $A = (a_0, a_1, \ldots, a_{n_1}), B = (b_0, b_1, \ldots, b_{n-1}) \in \mathbb{R}^n$, *then:*

$$\mathcal{F}(A * B) = \mathcal{F}(A)\mathcal{F}(B) \quad \textit{or} \quad A * B = \mathcal{F}^{-1}(\mathcal{F}(A)\mathcal{F}(B))$$

Proof: We prove the first statement of the theorem; the second is equivalent by Theorem 2.2.7.

By definition, the j-th component of $\mathcal{F}(A * B)$, say $\hat{c}_j$, is:

$$\hat{c}_j = \sum_{k=0}^{n-1} c_k \omega_n^{jk} = \sum_{k=0}^{n-1} \left(\sum_{l=0}^{n-1} a_l b_{k-l} \right) \omega_n^{jk} = \sum_{l=0}^{n-1} a_l \omega_n^{jl} \left(\sum_{k=0}^{n-1} b_{k-l} \omega_n^{j(k-l)} \right)$$

where we have changed the order of the sums and introduced a factor of $\omega_n^{jl}\omega_n^{-jl} = 1$. Recalling that the $k-l$ subscript is computed modulo n and that powers of ω_n may be reduced modulo n since $\omega_n^n = 1$, we recognise the bracketed term of the final equality as in fact being $\hat{b}_j$, the j-th term of $\mathcal{F}(B)$. So we have:

$$\hat{c}_j = \sum_{l=0}^{n-1} a_l \omega_n^{jl} \hat{b}_j = \hat{a}_j \hat{b}_j,$$

which is what we require.

□

The convolution theorem, combined with our earlier observation that multiplying two polynomials can be accomplished by computing the convolution of the coefficient vectors of these two polynomials, show how we can use the DFT to multiply polynomials: we compute the DFT of the coefficient vectors of each polynomial, multiply these two vectors component-wise and then compute the inverse DFT of the result. It is clear that the DFT of a vector $A \in R^n$ can be computed using $O(n^2)$ multiplications in R by proceeding directly according to the definition. The effort involved in computing the DFTs dominates the effort involved in multiplying two polynomials in this way, so the multiplication can be performed with $O(n^2)$ multiplications. This offers no improvement over classical multiplication - in fact, when one considers the magnitude of the constant hidden by the asymptotic notation, it is worse! Fortunately, this naive method of computing DFTs is not optimal, which motivates the following definition:

Definition 2.2.11. Fast Fourier Transform
A *Fast Fourier Transform* is an algorithm for computing the DFT of a vector $A \in R^n$ using less than $O(n^2)$ multiplications in R.

Many FFT algorithms are known. The first, and by far the most widely used, is the Cooley-Tukey algorithm, published by J. W. Cooley and J. W. Tukey in 1965 [18] (although apparently known, much earlier, to Gauss). In this thesis, "FFT" should be understood to mean the Cooley-Tukey FFT. By using the FFT to compute DFTs with complexity less than $O(n^2)$, we can multiply complex polynomials faster than the classical method. This is the idea used by Schönhage and Strasssen to multiply large integers and by Pollard to multiply polynomials over finite fields.

We discuss a specific case of the Cooley-Tukey FFT, which is the simplest to describe and is known as the "radix 2, decimation in time" (DIT) case. This case can be used only to transform vectors whose length is a power of 2. In practice, this is not a significant limitation, as any vector can be made into such a vector by adding additional 0 components without affecting the transform. This is not the only form of the FFT possible and alternatives which place less restrictions upon the vector to be transformed are discussed in [18].

Theorem 2.2.12. ***Cooley-Tukey FFT (Radix 2, Decimation in Time (DIT))***
Let R be a commutative ring with identity in which 2 is a unit, i.e. division by 2 is possible. Let n be a power of 2 and let $A = (a_0, a_1, \ldots, a_n - 1) \in R^n$. Then the DFT $\hat{A}$ of A or its inverse may be computed using $O(n \log(n))$ multiplications in R.

Proof: Consider the definition of the j-th component of $\hat{A}$. Recalling that if ω_n is an n-th root of unity, then ω_n^2 is an $n/2$-th root of unity, which we denote

$\omega_{n/2}$, we see that:

$$\begin{aligned}\hat{a}_j &= \sum_{k=0}^{n-1} a_k \omega_n^{jk} \\ &= \sum_{k=0}^{n/2-1} a_{2k} \omega_n^{2jk} + \sum_{k=0}^{n/2-1} a_{2k+1} \omega_n^{j(2k+1)} \\ &= \sum_{k=0}^{n/2-1} a_{2k} \omega_{n/2}^{jk} + \omega_n^j \sum_{k=0}^{n/2-1} a_{2k+1} \omega_{n/2}^{jk}.\end{aligned}$$

We recognise the two sums in the final equality as two $n/2$-th order DFTs, of the vectors:

$$A_{\text{EVEN}} = (a_0, a_2, \ldots, a_{n-2}) \quad \text{and} \quad A_{\text{ODD}} = (a_1, a_3, \ldots, a_{n-1}).$$

Thus we see that we can compute the n components of $\hat{A}$ by computing two DFTs of half the order of our original transform, as well as n multiplications by the additional factors ω_n^j. These factors have become traditionally known as "twiddle factors".

Note that this "splitting" of the original DFT was possible because n was even. Since n was chosen to be a power of 2, $n/2$ is also even and so we can split the two DFTs of order $n/2$ in a similar manner, ending up with four DFTs of order $n/4$ and some additional twiddle factors. We can continue to apply this procedure recursively until we are computing "DFTs of order 1". Observing the similarity in form between the DFT and the inverse DFT, it should be clear that we can compute an inverse DFT in a similar way. Note that the the inverse DFT requires a division by n. Our requirement that 2 was a unit in R means that division by any power of 2 is possible, and so the inverse DFTs required at any step of our recursive algorithm can always be computed.

We now consider the complexity of this approach. It is clear that the total number of splittings of the original DFT which are performed is $\log(n)$. At each stage of the algorithm, we need to perform some multiplications by twiddle factors. The number of such multiplications is always $O(n)$, as we have n multiplications in the first splitting, $n/2$ in the second, etc. Thus, the total number of multiplications in R we need to perform is $O(n \log(n))$, which proves the theorem.

□

We now consider the application of the above theory to the task of multiplication.

Integer Multiplication

We may use Theorem 2.2.1 to reduce the task of multiplying two integers to that of multiplying two polynomials with integer coefficients. By combining Theorem 2.2.9, the Convolution Theorem and Theorem 2.2.12, we can do this multiplication quickly by using an FFT over any appropriate ring which contains the relevant integer coefficients and in which these coefficients multiply in the same manner as they do in $\mathbb{Z}$.

Since $\mathbb{Z} \subset \mathbb{C}$, we could use the complex numbers, with $\omega_n = e^{2\pi i/n}$ our n-th root of unity. This is not particularly efficient for practical implementation, as there is some loss of precision involved in working with floating point approximations to complex numbers. We can usually recover the exact result by rounding, but since our coefficients are all integers we would like to be able to multiply them precisely.

One solution is to perform the transformation in the ring of integers modulo F_N, $\mathbb{Z}_{F_N}$, where F_N is an appropriately chosen *Fermat number*, i.e. is of the form $F_N = 2^{2^N} - 1$ for some natural N. If N is large enough, then the products of our integer coefficients will not exceed F_N and so the result will be the same as multiplying the polynomials in $\mathbb{Z}$. The roots of unity necessary for radix 2 DIT FFTs are abundant in this ring, as for any 2^n we may set $\omega_{2^n} = 2^{2^{N-n}}$ and then $\omega_{2^n}^{2^n} = (2^{2^{N-n}})^{2^n} = 2^{2^N} \equiv 1 \pmod{2^{2^N} - 1}$. Similarly, division by 2, and hence by any power of 2, is possible, as we observe that if $2^{-1} = 2^{2^{N-1}}$ then $2.2^{-1} = 2^{-1}.2 = 2^{2^N} \equiv 1 \pmod{2^{2^N} - 1}$.

The approaches with $R = \mathbb{C}$ and $R = \mathbb{Z}_{F_N}$ are both considered by Schönhage and Strassen in [68]. They show that the second method, using Fermat numbers, is the most efficient, and allows two n-bit integers to be multiplied in $O(n \log(n) \log(\log(n)))$ operations.

Multiplication of Polynomials over Finite Fields

We may once again combine Theorem 2.2.9, the Convolution Theorem and Theorem 2.2.12 to arrive at the following result:

Corollary 2.2.13. *Let $\mathbb{F}_q$ be a finite field. If $\mathbb{F}_q$ supports the FFT, then two polynomials $f(x)$ and $g(x) \in \mathbb{F}_q[x]$ of degree $\leq n$ may be multiplied in $O(n \log(n))$ multiplications in $\mathbb{F}_q$.*

Note the requirement that $\mathbb{F}_q$ supports the FFT. There are two possible obstacles here:

The first is that if our polynomials are over a field $\mathbb{F}_q$ which is of characteristic 2, then in that field $2 = 1 + 1 = 0$, which is certainly not a unit. Division by 2 is impossible and so the radix 2 DIT FFT which we have described above cannot be used. This is not a particularly severe problem, as it is possible to define a radix 3 FFT which recursively splits DFTs of vectors with length a power of 3 into 3 DFTs of one third the size. Such an algorithm requires divisibility by powers of 3, which in a field of characteristic 2 is simply equal to 1, which is obviously invertible. The details of such an FFT are explained in [18]. The fast multiplication of polynomials over finite fields of characteristic 2 is also discussed by Schönhage in [67]. A different approach to the fast computation of DFTs over fields of characteristic 2, using the theory of linearised polynomials, is given by S. Federenko and P. Trifinov in [76].

The second problem which may arise is that a particular field may not contain one of the primitive roots of unity required to support a DFT. In this case it may be necessary to perform the transformation in an extension field of the field of interest, which contains these roots.

2.2.5 Further Reading

The basic idea behind Karatsuba's method is breaking a polynomial down into two polynomials of lesser degree and using this decomposition to speed up multiplication. This idea has been extended by Toom [75] and Cook [17]. There is a family of Toom-Cook methods for polynomial multiplication which break a polynomial down into k polynomials of lesser degree to speed up multiplication. For $k = 1$, the Toom-Cook method is exactly classical multiplication, and for $k = 2$ it is exactly Karatsuba's method. The Toom-Cook algorithm with $k = 3$ is commonly used and is faster than Karatsuba's method. As the value of the parameter k is increased, the efficiency of the Toom-Cook eventually begins to degrade.

A very interesting paper by D. J. Bernstein from 2001 [7] has the ambitious goal of "first, to present every known technique for computing the product of two large integers; second, to present every known technique for computing the product of two polynomials over a commutative ring". The methods of Karatsuba, Toom, and Schönhage and Strassen are all discussed, as are FFT based methods. Furthermore, each of these methods is considered in a very algebraic manner, quite different to our discussions here, as some combination of "liftings" and "reductions" between certain rings.

The use of Fourier transforms to multiply polynomials over rings can be generalised substantially. In 1991, D. G. Cantor and E. Kaltofen [14] showed that it could be used to multiply two degree n polynomials over an arbitrary, not necessarily commutative and not necessarily associative algebra in $O(n \log(n))$ algebra multiplications and $O(n \log(n) \log(\log(n)))$ algebra additions.

2.3 Fast Exponentiation

Closely related to the now discussed subject of multiplication in finite fields is the subject of exponentiation. It should come as no surprise that cryptographic algorithms relying upon the difficulty of the discrete logarithm problem are heavily dependent upon exponentiation. Diffie-Hellman key exchange, Elgamal encryption and Digital Signature Standard signing all require at least one and usually many exponentiations within a finite field. Fast exponentiation is essential to efficient implementation of these algorithms. In this section, we consider two faster alternatives to "classical exponentiation". We note that, using our integer and polynomial representations of finite fields, exponentiation in these fields is simply exponentiation performed modulo some prime or irreducible element, which we shall denote by p.

2.3.1 Classical Exponentiation

The naive and obvious method for computing $\alpha^n \pmod p$ is to multiply the element α by itself and n times, reducing the result modulo p at each stage. This method clearly requires $O(n)$ field multiplications to obtain the final result. Even if we perform these n multiplications using the fast multiplication algorithms of the previous section, this method is far from optimal. Furthermore, unlike the problem of polynomial multiplication, in which the classical algorithm is the fastest for multiplying polynomials of low degree, classical ex-

ponentiation can always be improved upon, even for low exponents. This costly method of exponentiation should never be used in practice.

2.3.2 Exponentiation by Squaring

A much more efficient method for exponentiation, which can be used in any finite field (indeed, in any ring), goes by the name *exponentiation by squaring*[5]. In this method, α^n is computed using the recursive algorithm ExpSquare defined below:

Algorithm 2.3.1. ExpSquare(α, n)

Input: Field element $\alpha \in \mathbb{F}_q$ and exponent $n \in \mathbb{N}$.
Output: α^n.

---

1. **If** $n = 1$ **do**:
   (a) **Return** α.
2. **Else if** n is even **do**:
   (a) **Return** ExpSquare$(\alpha^2, n/2)$.
3. **Else do**:
   (a) **Return** αExpSquare$(\alpha^2, (n-1)/2)$.

This method obtains a result with a number of field multiplications of the order $\log(n)$, which is vastly more efficient than the naive method.

2.3.3 Addition Chain Exponentiation

Addition chain exponentiation is an exponentiation method which can at times require fewer multiplications than exponentiation by squaring, but which has certain precomputation and storage overheads which do not make it appropriate for all situations. We require a preliminary definition:

Definition 2.3.2. Addition Chain

An *addition chain* of length n is a sequence of integers $\{a_i\}_{i=0}^{n}$ with the properties:"

1. $a_0 = 1$,
2. For $k > 0, a_k = a_i + a_j$ for some $i, j < k$.

An addition chain of length n may be called an addition chain *for* a_n, e.g.: an addition chain for 26 (but not the only one) is:

$$\{1, 2, 4, 6, 10, 14, 20, 26\},$$

as $2 = 1 + 1, 4 = 2 + 2, 6 = 4 + 2, \ldots, 26 = 20 + 6$.

[5]This method is also known as "square and multiply", "binary exponentiation" and the "Russian peasant method"

The first n numbers of the well-known Fibonacci sequence are always an addition chain of length n, due to the definition of the Fibonacci numbers. It is clear from the definition that the second term of any addition chain must be equal to 2.

The central idea behind addition chain exponentiation is to compute an addition chain for the desired exponent n. Suppose that $\{a_i\}_{i=0}^{k}$ is such a chain. We know $\alpha^{a_0} = \alpha^1$ and may compute $\alpha^{a_1} = \alpha^2$ using a single multiplication. Now, $a_2 = a_i + a_j$ for $i, j < 2$ and so $\alpha^{a_2} = \alpha^{a_i}\alpha^{a_j}$. We may compute this readily since we know α^{a_i} and α^{a_j} for all the $i, j < 2$, i.e. $i, j = 0, 1$. Once we have computed α^{a_3}, we can in a similar manner compute α^{a_4}, since a_4 must be a sum of two earlier terms in the addition chain. Proceeding in this way, we may "climb" the addition chain until we have computed $\alpha^{a_k} = \alpha^n$. Algorithm 3.2.2 formally defines this process:

Algorithm 2.3.3. AddChainExp(α, n)

Input: Field element $\alpha \in \mathbb{F}_q$ and exponent $n \in \mathbb{N}$.
Output: α^n.

---

1. Compute an addition chain $\{a_i\}_{i=0}^{k}$ for the exponent n.
2. **For** l from 1 to k **do**:
   (a) Compute and store α^{a_l} as $\alpha^{a_l} = \alpha^{a_i}\alpha^{a_j}$ for the appropriate $i, j < l$.
3. **Return** α^{a_k}.

The number of multiplications required for an addition chain exponentiation is clearly dependent upon the length of the addition chain. In some cases, addition chains can be found of short enough length that addition chain exponentiation will be more efficient than exponentiation by squaring. The lowest exponent for which this occurs is $n = 15$. Using exponentiation by squaring, we can compute α^{15} as $\alpha(\alpha(\alpha\alpha^2)^2)^2$, using 6 multiplications. However, using the addition chain $\{1, 2, 3, 6, 12, 15\}$ we may compute α^{15} as $\alpha\alpha^2((\alpha\alpha^2)^2)^2$, using 5 multiplications. Greater savings can be achieved for higher exponents.

Addition chain exponentiation suffers from many complications. For instance, of obvious interest is the length of the shortest possible addition chain for a given exponent, the so-called *optimal addition chain*. There are many outstanding conjectures but few proven results providing estimates or bounds for the length of optimal addition chains. Furthermore, the task of actually computing an optimal addition chain is quite expensive, potentially defeating the advantage provided by addition chain exponentiation. There are, however, algorithms for efficiently computing addition chains which are approximately optimal and still offer better performance than exponentiation by squaring. For a discussion of these issues, see Knuth [35]. Finally, we note that addition chain exponentiation requires storing intermediate results which may be required later further up the chain, whereas exponentiation by squaring requires only the storage of the current result.

Because of the expense of computing addition chains, addition chain exponentiation is best suited to applications where exponentiation with one particular exponent will be required many times, in which case an addition chain can be

precomputed once and then stored for all further uses. Later in the thesis we will see a situation where this is definitely the case; using the Cantor-Zassenhaus polynomial factorisation algorithm (introduced in Section 4.4) repeatedly to factor different polynomials over the same field, as a part of the index calculus algorithm for computing discrete logarithms (introduced in Chapter 5).

2.4 Construction of Irreducible Polynomials

Throughout the entirety of this thesis, we perform computations in prime power order fields $\mathbb{F}_{p^n}$ using the polynomial representation given in Theorem 2.1.2, i.e. as the factor ring of polynomials over $\mathbb{F}_p$ modulo the ideal generated by an irreducible polynomial $p(x)$ of degree n. Thus the problem of how we can generate irreducible polynomials of a given degree is of immediate practical concern. The problem turns out to be quite simple; we can efficiently find irreducible polynomials via the obvious method of successively generating random polynomials and testing for irreducibility. There are certainly less naive methods which are more effective for large degrees n. It is also true that not all irreducible polynomials are equally attractive for use in polynomial representations. These details are discussed briefly at the end of the section.

The main result of this section is:

Theorem 2.4.1. *There exists a probabilistic algorithm which, given as input a finite field $\mathbb{F}_q$ and positive integer n, produces as output an irreducible polynomial $p(x) \in \mathbb{F}_q[x]$ of degree n using $O(n^4 \log(q))$ operations.*

The algorithm which constitutes proof of Theorem 2.4.1 is the immediately obvious one of random generation and testing. To establish the given complexity, we require two lemmas.

The following lemma gives an explicit formula for the exact number of irreducible monic polynomials over $\mathbb{F}_q$ of degree n. A complete derivation of this result may be found in [44]. Although the derivation is not particularly difficult, we do not present details here in order to conserve space. The formula we will present involves the following function, well studied in number theory:

Definition 2.4.2. The Möbius Function

The *Möbius function* $\mu : \mathbb{N} \to \{-1, 0, 1\}$ is defined as follows:

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1, \\ (-1)^k & \text{if } n \text{ is the product of } k \text{ distinct primes,} \\ 0 & \text{if } n \text{ is divisible by the square of a prime .} \end{cases}$$

We have:

Lemma 2.4.3. ***On the Number of Irreducible Monic Polynomials of Degree n in $\mathbb{F}_q[x]$***

The number $N_q(n)$ of monic irreducible polynomials of degree n in $\mathbb{F}_q[x]$ is given by:

$$N_q(n) = \frac{1}{n} \sum_{d|n} \mu\left(\frac{n}{d}\right) q^d = \frac{1}{n} \sum_{d|n} \mu(d) q^{\frac{n}{d}}$$

Proof: Omitted.
From this it follows that:

Corollary 2.4.4. *The probability $P_q(n)$ that a non-zero polynomial in $\mathbb{F}_q[x]$ of degree n selected at random using a uniform probability distribution over all such polynomials is irreducible is approximately $1/n$.*

Proof: It is clear that every irreducible polynomial in $\mathbb{F}_q[x]$ may be written as a monic irreducible polynomial multiplied by a non-zero constant, so that there are $(q-1)N_q(n)$ irreducible polynomials of degree n. The total number of non-zero polynomials of degree n is $(q-1)q^n$, so that the probability that a random non-zero polynomial is irreducible is:

$$P_q(n) = \frac{(q-1)N_q(n)}{(q-1)q^n} = \frac{1}{nq^n}\sum_{d|n}\mu(d)q^{n/d}.$$

Separating the case $d = 1$ from the other terms in the sum above and observing $\mu(1) = 1$ we get:

$$P_q(n) = \frac{1}{n} + \frac{1}{nq^n}\sum_{d|n,d>1}\mu(d)q^{n/d}.$$

It is easy to bound the sum in the above equation, which represents the error in our approximation, and see that it is "small". For instance, the author has shown that the sum is $< q^{1-n/2}$, so that even when $q = 2$, the smallest possible value, we need only consider polynomials of degree $n \geq 22$ before the error is less than 0.001. For fields of interest in present day public-key cryptography, such as $\mathbb{F}_{2^{1024}}$ or $\mathbb{F}_{2^{2048}}$, the error is truly negligible.

□

The second lemma we require to prove Theorem 2.4.1 is the following result, due to Gauss. This result shall be used again later in the thesis; we term it "Gauss' product lemma" for ease of reference.

Lemma 2.4.5. *For a given $n \in \mathbb{N}$, the product of all monic irreducible polynomials in $\mathbb{F}_q[x]$ having degrees d such that $d \mid n$ is the polynomial $x^{q^n} - x$.*

Proof: Omitted.
We shall use the following corollary to Gauss' product lemma.

Corollary 2.4.6. *A monic polynomial $f(x) \in \mathbb{F}_q[x]$ of degree n is irreducible if and only if it satisfies both:*

1. $f(x) \mid x^{q^n} - x$,
2. $\gcd(f(x), x^{q^{n/p}} - x) = 1$ *for all prime divisors p of n.*

Proof: We prove the "if" statement of the theorem. The "only if" statement may be established in a similar way.

Suppose $f(x)$ satisfies condition 1. Then it is an immediate consequence of Gauss' product theorem that $f(x)$ is either an irreducible polynomial of degree n or a product of several irreducible polynomials, say $f_0(x), f_2(x), \ldots, f_m(x)$, of degrees $n_0, n_2, \ldots, n_m$ such that $\prod_{i=0}^{m} n_i = n$. Suppose that the latter is the case. If $n_i = n/p_i$ then $f_i(x)$ is an irreducible polynomial of degree dividing

n/p_i and so contains as a factor a monic irreducible polynomial of the same degree, say $f_i'(x)$. By Gauss' product theorem $f_i'(x)$ is also a factor of $x^{q^{n/p}} - x$, so $\gcd(f(x), x^{q^{n/p}} - x) \neq 1$ and $f(x)$ does not satisfy condition 2. Hence if $f(x)$ satisfies both conditions it must indeed be an irreducible polynomial of degree n.

□

We now present an algorithm which shall be used as a subroutine in the algorithm proving Theorem 2.4.1.

Lemma 2.4.7. Testing Polynomials for Irreducibility

There exists a deterministic algorithm which determines if a given input polynomial $p(x) \in \mathbb{F}_q[x]$ is irreducible over $\mathbb{F}_q$ in time $O(n^3 \log(q))$.

Proof: Consider the following algorithm:

Algorithm 2.4.8. TESTIRRED($p(x)$)

Input: A polynomial $p(x) \in \mathbb{F}_q[x]$.
Output: The flag **True** if $p(x)$ is irreducible, the flag **False** otherwise.

---

1. **If** $p(x) \not| \, x^{q^n} - x$ **do**:
   (a) **Return False**.
2. **For** all prime divisors p of n **do**:
   (a) **If** $\gcd(p(x), x^{q^{n/p}} - x) \neq 1$ **do**:
      i. **Return False**.
3. **Return True**.

It is clear from Corollary 2.4.6 that this algorithm returns **True** if and only if $p(x)$ is irreducible. This test involves one division and then as many GCD computations as there are prime divisors of n. For ease of analysis, we observe that we may test $p(x) \not| \, x^{q^n} - x$ by testing that $\gcd(p(x), x^{q^n} - x) = 1$. The total number of GCDs thus required is certainly less than n. The largest polynomial in each GCD has degree at most x^{q^n}, which is exponential in the input size. However, we can avoid an exponential algorithm by using the fact that $\gcd(p(x), x^{q^n} - x) = \gcd(p(x), x^{q^n} - x \pmod{p(x)})$. To reduce $x^{q^n} - x$ modulo $p(x)$ it is sufficient to reduce x^{q^n}. We can do this using exponentiation by squaring with reduction modulo $p(x)$ at each stage. Beginning with x, this requires $O(\log(q^n)) = O(n \log(q))$ steps, with each step requiring the multiplication of two polynomials of degree at most n, taking time $O(n^2)$, and the reduction modulo $p(x)$ of the resulting product of degree at most $2n$, taking again time $O(n^2)$. Thus this reduction takes time $O(n^2 \log(q))$. We then compute the GCD of two polynomials of degree at most n, taking time $O(n^2)$, which is dominated by the time taken for the reduction. This process of reduction and GCD computation must be repeated n times, for a total time of $O(n^3 \log(q))$.

□

We now prove Theorem 2.4.1.

Proof of Theorem 2.4.1:

We use TestIrred (Algorithm 2.4.8) as a subroutine in the following probabilistic algorithm:

Algorithm 2.4.9. GenIrred($\mathbb{F}_q, n$)

Input: A finite field $\mathbb{F}_q$ and a degree $n \in \mathbb{N}$.
Output: An irreducible polynomial $p(x) \in \mathbb{F}_q[x]$ with $\deg(p(x)) = n$.

---

1. **Unconditionally do:**
   (a) Select a monic $p(x) \in \mathbb{F}_q[x]$ of degree n at random, using a uniform probability distribution on the set of all such polynomials.
   (b) **If** TestIrred($p(x)$) is **True do**:
      i. **Return** $p(x)$.

The functionality of this algorithm is clear. It will continually generate random polynomials in $\mathbb{F}_q[x]$ of degree n until one is determined to be irreducible by our previous algorithm TestIrred, then return that polynomial. From Corollary 2.4.4 we expect the algorithm to have to generate and test n polynomials before an irreducible one is found. We showed that each irreducibility test using TestIrred requires $O(n^3 \log(q))$ operations, so finding an irreducible polynomial using GenIrred requires $O(n^4 \log(q))$. Since $p(x)$ is monic, the input size is $n \log(q)$, so GenIrred is a polynomial time algorithm, as $O(n^4 \log(q)) < O((n \log(q))^4)$.

□

2.4.1 Further Reading

There are certainly more sophisticated methods for constructing irreducible polynomials over finite fields than the trial and error method given here. When polynomials of large degree are required, use of these may be preferable due to the time required by GenIrred increasing in proportion to n^4.

A paper due to Brawle and Carlitz from 1987 [10] describeds a binary operation on a subset of $\mathbb{F}_q[x]$; that is, a way to combine two polynomials over $\mathbb{F}_q$ of a certain kind to obtain a third such polynomial. This operation is termed the *composed product*, and has the property that if two polynomials of relatively prime degree are each irreducible then so to is their composed product. This fact can be used to "build up" large irreducible polynomials from smaller ones constructed using the simple method presented here. A discussion of how to efficiently compute composed products using matrix multiplication can be found in [9].

A sophisticated probabilistic algorithm for the fast construction of irreducible polynomials due to Shoup [70] was asymptotically the fastest algorithm for the task at the time of its publication. The paper introduced two new irreducibility tests, and also gives several references to other tests.

Sometimes there is an incentive to consruct irreducible polynomials of a certain kind. For instance, irreducible *binomials* and *trinomials* (polynomials

with 2 and 3 non-zero terms, respectively) are desirable for use in the polynomial representation of finite fields, as reduction modulo such "sparse" polynomials is more efficient than that modulo "dense" polynomials. A number of results regarding irreducible binomials and trinomials can be found in the book by Blake *et. al.* [9].

Chapter 3

Generic Discrete Logarithm Algorithms

In this chapter, we present five algorithms for computing discrete logarithms and analyse their time and space complexity. The algorithms, in order of presentation, are:

1. Trial Exponentiation,
2. Shanks' "Baby-Step Giant-Step" Method,
3. Pollard's ρ-method (1978),
4. Pollard's λ-method (1978).
5. The Pohlig-Hellman method (1978).

Any of these algorithms may be used to mount an attack against any of the cryptographic primitives which were presented in Chapter 1. As such, an understanding of the complexities of these algorithms is essential in deciding how large a finite field must be used in a cryptographic primitive in order to render an attack infeasible. Any of the techniques for efficient computation in finite fields discussed in Chapter 2 may be used in the implementation of these algorithms.

The algorithms in this chapter are known as *generic* discrete logarithm algorithms. The reason for this is that they can be used to solve the DLP in an arbitrary finite cyclic group. In keeping with the theme of this thesis, we shall present the algorithms and the result in terms of finite fields; the generalisations to arbitrary groups are trivial and immediately obvious. To fix notation throughout the chapter, we shall concern ourselves with the computation of the logarithm of β with respect to the primitive element α of the finite field $\mathbb{F}_q$.

We shall see that the fastest algorithms in this chapter require $O(\sqrt{q})$ steps to compute a logarithm in $\mathbb{F}_q^*$. For this reason, the algorithms are sometimes referred to as *square root attacks*. It is, in fact, a proven result that this is the least complexity a generic algorithm can accomplish; see, for example, the paper of V. Shoup [71]. Later, in Chapter 5, we present a class of non-generic algorithms which are specific to finite fields. These algorithms allow us to achieve a lower complexity than $O(\sqrt{q})$; indeed, these algorithms achieve the lowest complexities known.

3.1 Trial Exponentiation

Theorem 3.1.1. Trial Exponentiation
There exists a deterministic algorithm which solves the DLP in $\mathbb{F}_q^*$ *in time* $O(q\log(q))$.

The algorithm proving this theorem is the immediately obvious method of "trial and error", also known as *trial exponentiation*. To find $\log_\alpha(\beta)$ the algorithm successively raises α to various powers and compares the result to β, exhaustively searching the space of possible logarithms $\mathbb{Z}_{q-1}$ until the correct answer is found. This method is obviously correct and simple to implement for any finite field, but is too inefficient to be used in any but the smallest fields. For fields of cryptographic interest, it is completely inpractical. We mention it only to place an upper bound on the difficulty of the DLP.

3.2 Shanks' Baby-Step Giant-Step method

Theorem 3.2.1. Shanks' Baby-Step Giant-Step Method

There exists a deterministic algorithm which solves the DLP in $\mathbb{F}_q^*$ *in time* $O(\sqrt{q-1})$ *using space* $O(\sqrt{q-1}\log(q))$.

The algorithm described by Theorem 3.2.1 is described by D. Knuth in [35], where it is attributed to D. Shanks. No work on the method by Shanks appears to have been published, and [35] is the standard reference in the literature. The method is related to trial exponentiation, in that it requires the trialing of a number of possible exponents, but the number of trials required is decreased. This decrease in the number of trials is achieved at the expense of requiring the storage of a several field elements; the algorithm is a time-space tradeoff.

Proof of Theorem 3.2.1:
Consider the following algorithm:

Algorithm 3.2.2. SHANKS($\mathbb{F}_q, \alpha, \beta$)

Input: A finite field $\mathbb{F}_q$, a primitive element α of $\mathbb{F}_q$ and $\beta \in \mathbb{F}_q$.
Output: $\log_\alpha(\beta)$.

---

1. $m \leftarrow \lceil\sqrt{q-1}\rceil$.
2. **For** i from 0 to m **do**:
   (a) Compute α^i and store the pair (i, α^i).
3. **For** j from 0 to m **do**:
   (a) Compute $\beta\alpha^{-jm}$ and search the stored pairs for a pair (i, α^i) such that $\beta\alpha^{-jm} = \alpha^i$.
   (b) **If** such a pair is found:
      i. **Return** $i + jm$.

The correctness of this algorithm is based upon the observation that any element x of $\mathbb{Z}_{q-1}$, including the sought logarithm, may be written as $x = i + jmq - 1$, where both $i, j < m$. To see this, simply set $j = \lfloor x/m \rfloor$ and then $i = x - jm$. It is clear that $j < m$ since $x \leq q - 1$ and so $x/\sqrt{q-1} \leq \sqrt{q-1}$. It is clear by construction that $i < m$.

The algorithm precomputes α^i for all i in this range and then searches for a value of j such that $\beta\alpha^{-jm} = \alpha^i$. When this occurs it is clear that $\beta = \alpha^{i+jm}$ and hence that $\log_\alpha(\beta) = i + jm$.

Considering the time complexity of this algorithm in the same manner as elsewhere in the thesis, the only contribution to the running time of this algorithm is that required to perform the computation of the powers of α. Computing α^i for $i = 0, 1, \ldots, m-1$ requires $O(\sqrt{q-1})$ multiplications and computing $\beta\alpha^{-jm}$ for $j = 0, 1, \ldots, m-1$ requires the same, giving a total complexity of $O(\sqrt{q-1})$. However, one could consider this analysis too simplistic, as it does not account for the time required to search the list of ordered pairs for the desired match. A discussion of different techniques for searching lists is beyond the scope of this thesis. The interested reader may see [35] for such a discussion. For our purposes, we consider it reasonable to expect that the time required for searching is dominated by the time required for computations and give the time complexity as $O(\sqrt{q-1})$.

The space complexity of Shanks's method is determined by the space required to store the m ordered pairs. The components of each pair are an integer $< m < q$ and an element of $\mathbb{F}_q$, both of which can be stored using $\log(q)$ bits. Thus the space required is $O(\sqrt{q-1}\log q)$.

□

Example

We demonstrate the use of Shanks' Baby-Step Giant-Step method by solving a DLP in the group $\mathbb{F}^*_{257} \cong \mathbb{Z}^*_{257}$, which has order $n = \phi(257) = 256$ (since 257 is prime). The element $g = 5$ is a generator of this group and we shall compute the base 5 logarithm of the element $h = 116$.

The method begins by computing and storing the values $(i, 5^i)$ for $0 \leq i < 16$. These computed pairs are shown in Table 3.1.

i	0	1	2	3	4	5	6	7
$5^i \pmod{257}$	1	5	25	125	111	41	205	254
i	8	9	10	11	12	13	14	15
$5^i \pmod{257}$	242	182	139	181	134	156	9	45

Table 3.1: Table of values (i, g^i) for BSGS computation of $\log_5(116)$ in $\mathbb{F}^*_{257}$.

Now we compute 116×5^{-16j} for $0 \leq j < 16$ until a value is found which ocurrs in the second row of Table 3.1. Observe that $103 \times 5 \equiv 1 \pmod{257}$ and so $5^{-16j} = (5^{-1})^{16j} = 103^{16j}$. Hence we compute:

$116 \times 103^{0\times16} \equiv 116 \pmod{257}$, which is not in the table.
$116 \times 103^{1\times16} \equiv 157 \pmod{257}$, which is not in the table.

$116 \times 103^{2\times 16} \equiv 228 \pmod{257}$, which is not in the table.
$116 \times 103^{3\times 16} \equiv 25 \pmod{257}$, which is in table entry $(2, 5^2)$.

So we have $i = 2, j = 3$ and the method gives $\log_5(116) = 2 + 3 \times 16 = 50$. We can then verify that, indeed, $5^{50} \equiv 116 \pmod{257}$.

3.3 Pollard's ρ-method

Theorem 3.3.1. ***Pollard's ρ-method***

There exists a probabilistic algorithm which solves the DLP in $\mathbb{F}_q^*$ with time complexity $O(\sqrt{q-1})$ and space complexity $O(\log(q))$.

The algorithm which constitutes proof of Theorem 3.3.1 is due to J. Pollard from 1978 [60] and is known as *Pollard's ρ-method* [1]. It is certainly the most well known generic discrete logarithm algorithm and is the most efficient in terms of time and space. It uses a "collision" in the terms of a pseudorandom sequence of field elements to establish a linear recurrence for the desired logarithm. This recurrence can then be solved using any of the usual methods, such as the extended Euclidean algorithm. The time complexity of the algorithm is derived from an assumption about the behaviour of the pseudorandom sequence, which is supported by heuristic observation.

The following proposition introduces the pseudorandom sequence considered by the method and also states the assumption about its behaviour which we will use in our later complexity analysis.

Proposition 3.3.2. ***Pseudorandom Sequences in $\mathbb{F}_q^*$***

Let $\{S_0, S_1, S_2\}$ be a partition of $\mathbb{F}_q^*$, with $S_0, S_1, S_2 \subset \mathbb{F}_q^*$ of roughly equal cardinality, and define the map $f : \mathbb{F}_q^* \to \mathbb{F}_q^*$ according to:

$$f(x) = \begin{cases} \beta x_i & \text{if } x_i \in S_0, \\ {x_i}^2 & \text{if } x_i \in S_1, \\ \alpha x_i & \text{if } x_i \in S_2. \end{cases} \tag{3.1}$$

Then the sequence of field elements $\{x_i\}_{i=0}^{\infty}$ defined by $x_0 = 1$, $x_{i+1} = f(x_i)$, $i = 1, 2, \ldots$ has the statistical properties of a sequence of field elements selected independently and at random from a uniform probability distribution over $\mathbb{F}_q^*$.

We note now that the sequence $\{x_i\}$ implicitly defines two sequences of integers, $\{a_i\}$ and $\{b_i\}$, whose terms are the powers of α and β in the factorisation of the corresponding terms of $\{x_i\}$. That is, $\{a_i\}$ and $\{b_i\}$ are defined such that $x_i = \alpha^{a_i}\beta^{b_i}$, so that $a_0 = b_0 = 0$ and:

$$a_{i+1} = f_a(a_i) = \begin{cases} a_i & \text{if } x_i \in S_0, \\ 2a_i \pmod{n} & \text{if } x_i \in S_1, \\ a_i + 1 \pmod{n} & \text{if } x_i \in S_2, \end{cases} \tag{3.2}$$

[1] J. Pollard has developed an algorithm for factorising integers which is very similar to his algorithm for computing logarithms, to the extent that they share the ρ name. The name "Pollard('s) ρ-method" seems to refer more often in the literature to the factorisation algorithm. While considered unnecessary in the context of this thesis, the qualification "Pollard('s) ρ-method for logarithms" is seen often, and should probably be used in general.

$$b_{i+1} = f_b(b_i) = \begin{cases} b_i + 1 \pmod{n} & \text{if } x_i \in S_0, \\ 2b_i \pmod{n} & \text{if } x_i \in S_1, \\ b_i & \text{if } x_i \in S_2. \end{cases} \tag{3.3}$$

Since $\mathbb{F}_q^*$ is finite, it is an obvious consequence of the well known "pigeon hole principle" that amongst the first q terms of this sequence, at least one field element must occur twice. We term this a *collision* in the sequence. Since each term in the sequence is entirely determined by the previous term, it follows that after the first collision the sequence becomes periodic. The name "ρ-method" arises from a visualisation of this; the 'tail' of the ρ represents the segment of $\{x_i\}$ before the first collision occurs, and the 'loop' of the ρ represents the cycle which the sequence repeats thereafter.

We now find the expected number of sequence terms which must be generated until a collision occurrs, using the assumption of Proposition 3.3.2.

Lemma 3.3.3. Birthday Paradox Result

Let S be a finite set with $|S| = n$. Suppose we successively select elements of S at random, replacing the chosen element after each selection, such that the selections are independent and at each selection each element is equally likely. Let X_n be the discrete random variable corresponding to the number of elements which must be selected until some element is selected for the second time. Then:

$$\lim_{n\to\infty} E(X_n) \simeq \sqrt{\pi n/2}$$

Proof: We use the fact that, for a discrete random variable X taking non-negative integer values less than n:

$$E(X) = \sum_{i=1}^{n} P(X \geq i). \tag{3.4}$$

To see this, consider expanding the sum above to get:

$$\sum_{i=1}^{n} P(X \geq i) = \sum_{i=1}^{n} \left(\sum_{j=i}^{n} P(X = j) \right).$$

Observe that only one $P(X = 1)$ term is contributed by the innermost sum, in the case $i = 1$. Similarly, two $P(X = 2)$ terms are contributed, when $i = 1, 2$. Continuing in this manner we see that there are a total of k $P(X = k)$ terms and so:

$$\sum_{i=1}^{n} P(X \geq i) = \sum_{i=0}^{n} iP(X = i),$$

which is $E(X)$ by definiton.

We now use a Taylor series to find an approximation to $P(X_n \geq i)$, observing that this is the probability that no collision occurs among the first $i-1$ selections.

Suppose the first selection has been made. The probability that a second selection is different is $(1-1/n)$. The probability that the third choice is different to the first two is $(1 - 2/n)$. Continuing in this manner and recalling that

the selections are independent, we see that the probability that the first $i-1$ selections are all different is:

$$P(X_n \geq i) = \prod_{j=0}^{i-2}(1 - \frac{j}{n}). \tag{3.5}$$

We now find an approximation to $P(X_n \geq i)$ which is valid when n is "large". Recall that the Taylor series expansion of e^x is:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Thus we have:

$$\begin{aligned} e^{-j/n} &= 1 - \frac{j}{n} + \frac{j^2}{2!n^2} - \frac{j^3}{3!n^3} + \dots \\ &= 1 - \frac{j}{n} + O\left(\left(\frac{j}{n}\right)^2\right). \end{aligned}$$

If n is "large" then 2^{nd} and higher order terms above are "small" and we may approximate $(1 - j/n)$ well by $e^{-j/n}$ and so (3.5) becomes:

$$P(X_n \geq i) \simeq \prod_{j=0}^{i-2} e^{-j/n} = e^{-(i-1)(i-2)/2n},$$

where we have used the well-known result $\sum_{0=1}^{i} j = i(i+1)/2$. Thus, using (3.4), we have for large n:

$$E(X_n) = \sum_{i=1}^{n} P(X_n \geq i) \simeq \sum_{i=1}^{n} e^{-(i-1)(i-2)/2n} \simeq \sum_{i=0}^{n} e^{-i^2/2n}.$$

We now approximate this final sum by a standard integral, giving:

$$E(X_n) \simeq \int_0^n e^{-x^2/2n} dx,$$

and hence:

$$\lim_{n\to\infty} E(X_n) = \int_0^\infty e^{-x^2/2n} dx = \sqrt{\pi n/2} \simeq 1.23\sqrt{n},$$

the improper definite integral being well-known.

□

This result is sometimes known as the "birthday paradox", because it establishes the surprising (though not actually paradoxical) result that, assuming people's birthdays ocurr as random and independent selections from a set of 365 possible days, each equally likely, then only around 23 people need to be present in a room before we expect two of them to share a birthday.

As a final consideration before presenting Pollard's ρ-method, we discuss a manner in which a collision in a sequence can be efficiently found. The idea behind the following lemma is known as "Floyd's cycle-finding algorithm". It was developed by R. Floyd in 1967.

Lemma 3.3.4. Floyd's Cycle-Finding Algorithm

Let $\{x_i\}_{i=1}^{\infty}$ be an "eventually periodic" sequence, i.e. suppose there exists an integer λ such that $\{x_i\}_{i=\lambda}^{\infty}$ is periodic, with period say μ. For example, $\{1, 2, 3, 4, 5, 4, 5, 4, 5, \ldots\}$ is "eventually periodic" with $\lambda = 4$ and $\mu = 2$. Then the two sequences $\{x_i\}_{i=1}^{\infty}$ and $\{x_{2i}\}_{i=1}^{\infty}$ collide after at least λ and at most $\lambda + \mu$ terms.

Proof: First we show that the two sequences do in fact collide. Once $i \geq \lambda$, the terms of $\{x_i\}$ are in the periodic part of the sequence, so $x_i = x_{i+k\mu}$ for any integer k. When i is a multiple of μ, say $k\mu$ then $x_{2i} = x_{2k\mu} = x_{k\mu+k\mu} = x_{k\mu} = x_i$ and a collision has occurred.

We now show that the collision occurs after at least λ and at most $\lambda + \mu$ terms. The lower bound is clear: the first $\lambda - 1$ terms of $\{x_i\}$ are in the non-periodic part of the sequence. If the value taken by each of these terms occurs exactly once in the entire sequence (as is the case in our earlier example), then clearly x_i and x_{21} will always be distinct. To establish the upper bound, note that a collision occurrs as soon as i is a multiple of μ. Suppose $\lambda > \mu$, say:

$$\lambda = k\mu + l, \text{ with } l < \mu.$$

Then the first multiple of μ after λ is:

$$\lambda + \mu - l < \lambda + \mu.$$

If $\lambda = \mu$, then the collision clearly occurs after λ terms. Finally, if $\lambda < \mu$, say:

$$\mu = k\lambda + l, \text{ with } l < \lambda,$$

then the first multiple of μ after λ is:

$$\lambda + (k-1)\lambda + l < \lambda + \mu.$$

□

Pollard's ρ-method applies this method of finding sequence collisions to the sequence in $\mathbb{F}_q^*$ generated by the function f given earlier.

We are now in a position to prove Theorem 3.3.1:

Proof of Theorem 3.3.1:

Consider Algorithm 3.3.5, PollardRho. This algorithm uses Floyd's cycle-finding algorithm to find a collision in the sequence of field elements defined by the map $f : \mathbb{F}_q^* \to \mathbb{F}_q^*$ defined in (3.1). On each pass through the unconditional loop, x_1 is updated by f once (at step 3a) and x_2 is updated by f twice (at step 3b), so that on the i-th pass through the loop x_1 corresponds to the i-th term in the sequence and x_2 the $2i$-th term. We update the variables a_1, b_1, a_1, and b_2 in a similar way with the maps f_a and f_b. We are certain that there exists a collision in the sequence generated by f, and so by Lemma 3.3.4 we know that after some finite number of passes through the loop we will have $x_1 = x_2$. This event is caught by the **If** statement.

Algorithm 3.3.5. PollardRho($\mathbb{F}_q, S, \alpha, \beta$)

Input: A finite field $\mathbb{F}_q$, a partition $\{S_0, S_1, S_2\}$ of $\mathbb{F}_q^*$, a primitive element α of $\mathbb{F}_q$ and $\beta \in \mathbb{F}_q$.
Output: $\log_\alpha(\beta)$.

---

1. $x_1, x_2 \leftarrow 1$.
2. $a_1, b_1, a_2, b_2 \leftarrow 0$.
3. **Unconditionally do**:
   (a) $x_1 \leftarrow f(x_1), a_1 \leftarrow f_a(a_1), b_1 \leftarrow f_b(b_1)$.
   (b) $x_2 \leftarrow f(f(x_2)), a_2 \leftarrow f_a(f_a(a_2), b_2 \leftarrow f_b(f_b(b_2))$.
   (c) **If** $x_1 = x_2$ **do**:
      i. $r \leftarrow b_1 - b_2 \pmod{q-1}$.
      ii. $s \leftarrow a_1 - a_2 \pmod{q-1}$.
      iii. **If** the congruence $rx \equiv s \pmod{q-1}$ is solvable **do**:
         A. Solve the congruence $rx \equiv s \pmod{q-1}$.
         B. Compute α^x for all solutions x.
         C. **Return** the x such that $\alpha^x = \beta$.
      iv. **Else do**:
         A. Set a_1 and a_2 to the same randomly selected value in $\mathbb{Z}_{q-1}$.
         B. Set b_1 and b_2 to the same randomly selected value in $\mathbb{Z}_{q-1}$.
         C. $x_1 \leftarrow \alpha^{a_1}\beta^{b_1}, x_2 \leftarrow \alpha^{a_2}\beta^{b_2}$.
         D. *Go to step* 3.

Observe that at the time $x_1 = x_2$ and the algorithm enters the stage beneath the **If** statement, the variables a_1, b_1, a_2, and b_2 are such that:

$$\begin{aligned} & \alpha^{a_1}\beta^{b_1} = \alpha^{a_2}\beta^{b_2} \\ \Rightarrow \quad & \beta^{b_1-b_2} = \alpha^{a_2-a_1} \\ \Rightarrow \quad & \log_\alpha((\beta^{b_1-b_2}) = \log_\alpha(\alpha^{a_2-a_1}) \\ \Rightarrow \quad & (b_1-b_2)\log_\alpha(\beta) \equiv a_2 - a_1 \pmod{q-1}. \end{aligned}$$

So we can derive from our knowledge of a_1, b_1, a_2, and b_2 a linear congruence for the logarithm of interest. This congruence may be solved for $\log_\alpha(\beta)$ using the extended Euclidean algorithm under the usual conditions. If these conditions are not met and the congruence has no solution, we may try again by restarting the algorithm with x_1 and x_2 initialised to some equal but random starting value distinct from 1. Eventually a useful congruence should be generated.

Once we have a solvable congruence, $\log_\alpha(\beta)$ will be among its solutions. The set of solutions is typically small enough that trial exponentiation can reveal the logarithm with minimal effort. This demonstrates the correctness of

the algorithm.

To establish the time complexity of the algorithm, recall from Lemma 3.3.4 that x_1 and x_2 will be equal after at most $\lambda + \mu$ passes through the unconditional loop. Under the assumption of Proposition 3.3.2, $\lambda + \mu$ is expected to be $O(\sqrt{q-1})$. Thus we must update the variables x_1, a_1, b_1 and x_2, a_2, b_2 a total of $O(\sqrt{q-1})$ times. The updating of the a_i and b_i variables requires modular multiplication of integers, which is exactly as expensive as updating x_1 and x_2 if q is prime or less expensive if q is a prime power. So the time to find the sequence collision is $O(\sqrt{q-1})$. The solving of the linear recurrence takes substantially less time than this, so $O(\sqrt{q-1})$ is the running time of the algorithm.

The space complexity of Pollard's rho-method is trivially $O(\log q)$. No storage is required other than the current value of $(x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i})$. In this sense the Pollard ρ-method is a vast improvement upon Shanks' Baby-step Giant-step method. It has the same time complexity, but a vastly reduced space complexity.

□

Example

To demonstrate the use of the Pollard ρ-method we once again consider the group $\mathbb{F}^*_{257}$, introduced in the example for Shanks' Baby-Step Giant-Step method. We again use the generator $g = 5$, but compute the logarithm of the element $h = 178$. Our partition $\{S_0, S_1, S_2\}$ shall be:

$$\begin{aligned} S_0 &= \{x \in \mathbb{Z}^*_{257} | x \equiv 1 \pmod 3\}, \\ S_1 &= \{x \in \mathbb{Z}^*_{257} | x \equiv 0 \pmod 3\}, \\ S_2 &= \{x \in \mathbb{Z}^*_{257} | x \equiv 2 \pmod 3\}, \end{aligned} \tag{3.6}$$

as this partition is the simplest to implement in most computer programming languages, and since it guarantees that the condition $|S_0| \simeq |S_1| \simeq |S_2|$ is satisfied.

Applying the Pollard ρ-method to this problem produces the sequences shown in Table 3.2.

We see that after 16 comparisons a collision between the sequences $\{x_i\}$ and $\{x_{2i}\}$ has occurred, with $x_{16} = x_{2\times 16} = 22$. This is exactly the expected number of comparisons $m = \lceil\sqrt{256}\rceil = \lceil 16\rceil = 16$. Using the values of $a_{16}, b_{16}, a_{2\times 16}, b_{2\times 16}$ we establish the equation:

$$5^8 178^{26} \pmod{257} = 5^{76} 178^{240} \pmod{257},$$

from which we can deduce, in the manner described earlier, the linear congruence:

$$42 \log_5(178) \equiv 68 \pmod{256}.$$

Since $d = \gcd(42, 256) = 2|68$ we know that this congruence has 2 solutions up to congruence modulo 256. Using the extended Euclidean algorithm we determine these solutions and hence that:

i	a_i	b_i	$x_i = g^{a_i}h^{b_i}$	a_{2i}	b_{2i}	$x_{2i} = g^{a_{2i}}h^{b_{2i}}$
0	0	0	1	0	0	1
1	0	1	178	0	2	73
2	0	2	73	0	6	176
3	0	3	144	1	7	127
4	0	6	176	1	9	19
5	1	6	109	2	10	205
6	1	7	127	2	12	59
7	1	8	247	4	12	190
8	1	9	19	8	26	22
9	1	10	41	8	28	64
10	2	10	205	16	58	117
11	2	11	253	33	116	83
12	2	12	59	35	116	19
13	3	12	38	36	117	205
14	4	12	190	36	119	59
15	4	13	153	38	119	190
16	8	26	**22**	76	240	**22**

Table 3.2: Steps in the Pollard ρ-method's computation of $\log_5(178)$ in $\mathbb{F}_{257}^*$

$$\log_5(183) \in \{x \in \mathbb{Z}_{256} | 42x \equiv 68 \pmod{256}\} = \{26, 154\}.$$

This set of possible logarithms is sufficiently small that trial exponentiation can quickly reveal which element is the logarithm. In this case we see that $5^{26} \equiv 178 \pmod{257}$ and so $\log_5(178) = 26$.

Empirical Observations

The author has implemented Pollard's ρ-method for multiplicative subgroups of fields of prime order on a computer in the C programming language. The code can be found in appendix section A.2. Empirical running times have been recorded for using the algorithm to compute logarithms for various prime fields. These times are tabulated in Table 3.3.

The prime field orders used were the first primes of size 20 bits, 24 bits, 28 bits, ..., 64 bits. The computations were performed on a machine with a 2.0 GHz Intel® Pentium®-M Centrino® processor and 1 GB of RAM, running the NetBSD® 3.0 operating system. The average running times were obtained by consecutively computing 500 discrete logarithms in each field and dividing the total time taken (obtained using the UNIX `time` command) by 500. The problems used randomly, independently selected pairs of elements and generators, which were precomputed so as not to affect the running times.

Modifications

We briefly mention and give reference to three modifications to the Pollard ρ-method which may be of interest.

Prime field order	Average running time
524309	0.00232 s
8388617	0.00601 s
134217757	0.02024 s
2147483659	0.07798 s
34359738421	0.47566 s
549755813911	1.76512 s
8796093022237	5.30934 s
140737488355333	33.10700 s
2251799813685269	126.04900 s
36028797018963971	453.96400 s
576460752303423619	2171.91200 s

Table 3.3: Empirical running times for Pollard ρ-method in prime order fields

1. The iterating function $f : \mathbb{F}_q \to \mathbb{F}_q$ used by Pollard to produce the pseudorandom sequence is not optimal and the performance of the algorithm actually falls short of that predicted by our analysis, which assumed that f acted like a uniformly distributed random function. In 2001 E. Teske gave an alternative method of sequence generation [74] which gives performance matching that predicted by our analysis. Teske suggests that this modification improves the algorithm's performance by about 20%.

2. Floyd's cycle detecting algorithm is not optimal. In 1980, R. Brent presented an improvement [11] to Pollard's ρ-method for factorising integers [59], which is conceptually similar to his ρ-method for computing discrete logarithms and also used Floyd's cycle detecting algorithm. Brent's improvement replaced Floyd's algorithm with an alternative, and this alternative can just as well replace Floyd's algorithm in the ρ-method for logarithms. In 1984, H. Lenstra and C. Schnorr also presented an integer factorisation algorithm [43] which included a modification of Pollard's ρ-method, again with an improved cycle-detection method which may be adapted to the ρ-method for logarithms.

3. In 1999, P. van Oorschot and M. Wiener published a paper considering the parallelisation of collision searches [77], which have many applications in cryptography including, but not limited to, discrete logarithm computations. This paper explicitly considers the parallelisation of Pollard's ρ-method to an arbitrary number of processors and shows that a near linear speedup can be achieved.

3.4 Pollard's λ-method (Kangaroo Method)

Theorem 3.4.1. Pollard's λ-method

There exists a deterministic algorithm which, given as input a finite field $\mathbb{F}_q$, a primitive element α of $\mathbb{F}_q^*$, $\beta \in \mathbb{F}_q^*$ and two integers $a, b \in \mathbb{Z}_{q-1}$, computes $\log_\alpha(\beta)$ in $\mathbb{F}_q^*$ if $a \leq \log_\alpha(\beta) \leq b$, in time $O(\sqrt{q-1})$ and space $O(\log q)$.

The algorithm proving this theorem is known as the Pollard λ-method, and was proposed by J. Pollard at the same time as his ρ-method [60]. Like the ρ-method, the λ-method is based upon finding sequence collisions and the algorithm's time complexity is estimated using the birthday paradox result Lemma 3.3.3. An explanation of the name "λ-method" will be given later in this section. It is noted that this method is also sometimes called the "(Pollard) Kangaroo Method". This is because Pollard's original description of the algorithm is couched in a perculiar analogy involving the trapping of a "wild" kangaroo by a "tame" one.

The λ-method considers two sequences $\{x_i\}$ and $\{y_i\}$ of elements in $\mathbb{F}_q^*$ which are thought of as the locations of two kangaroos which travel throughout the field in a series of pseudorandom "bounds". The length of each bound is a function of the take-off position. Associated with these two sequences are two other sequences $\{d_i\}$ and $\{d'_i\}$, which are thought of as the readings of "distance recorders" carried by the kangaroos, which record the length of each bound. One kangaroo is considered to be wild, the other tame. It is the task of the tame kangaroo to capture the wild kangaroo in a trap. The tame kangaroo begins at a particular starting position (field element) and then completes some number N of bounds, at which point it sets a trap. The wild kangaroo begins at an unknown starting position (field element), corresponding to the discrete logarithm we wish to compute. If the wild kangaroo lands after any bound on a position previously occupied by the tame kangaroo, then it will follow the tame kangaroo's path to the trap. Once the wild kangaroo has been trapped, a comparison of the distance recorders on each kangaroo together with our knowledge of the tame kangaroo's starting point will enable us to determine the wild kangaroo's starting point and hence the discrete logarithm. The name "λ-method" arises from a visualisation of this. The two kangaroos initially travel along two distinct paths, represented by the two "legs" of the lower half of the λ. Eventually the kangaroos collide and their paths coincide forever after, represented by the single "tip" of the λ.

We represent this rather fanciful description mathematically as follows. Let the sequence $\{x_i\}$ of field elements correspond to the position of the tame kangaroo. We begin this sequence at $x_0 = \alpha^B$, a field element with a known logarithm. The sequence $\{y_i\}$ then corresponds to the position of the wild kangaroo. We begin this sequence at $y_0 = \beta$, a field element with an unknown logarithm. Subsequent terms of both sequences are determined by the same rule (as both kangaroos have identical jumping behaviour):

$$z_{i+1} = \alpha^{f(z_i)} z_i \quad \text{for } z = y, x. \tag{3.7}$$

The function f is a pseudorandom map $f : \mathbb{F}_q^* :\to S$, where S is a set of integers. $f(\alpha)$ is the length of the bound taken by a kangaroo which takes off from α. We see now that the length of a bound is equal to the difference in the discrete logarithm of the field elements which correspond to the bound's points of take off and landing.

The distance recorder sequences are defined as follows. The sequence $\{d_i\}$ records the total distance travelled by the tame kangaroo after the i-th bound and the sequence $\{d'_i\}$ does the same for the wild kangaroo. We set $d_0 = d'_0 = 0$

and then for $i = 1, 2, \ldots$ we have:

$$d_i = \sum_{j=0}^{i} f(x_j) \quad \text{and} \quad d'_i = \sum_{j=0}^{i} f(y_j).$$

Note that from this definition we have:

$$x_i = \alpha^{B+d_i} \quad \text{and} \quad y_i = \beta\alpha^{d'_i}.$$

We begin by choosing a number of bounds N for the tame kangaroo to complete. We compute and store $\{x_0, x_1, \ldots, x_N\}$ and compute $\{d_0, d_1, \ldots, d_N\}$, storing only d_N. We then consider the path of the wild kangaroo. We compute successive terms of $\{y_i\}$ and $\{d'_i\}$ until either:

1. $y_j = x_N$ for some integer j, at which point capture has occurred, or
2. $d'_j > B - A + d_N$, at which point the wild kangaroo has travelled further from the tame kangaroo's starting point than the tame kangaroo did and hence has necessarily avoided the trap.

If capture has occurred with $y_j = x_N$, then we have:

$$x_N = y_j \Rightarrow \alpha^{B+d_N} = \beta\alpha^{d'_j} \Rightarrow \beta = \alpha^{B+d_N-d'_j},$$

and hence $\log_\alpha(\beta) \equiv B + d_N - d'_j \pmod{q-1}$. If capture does not ocurr, the method has failed and we must start again with a different choice of f and/or of S and continue to do so until capture occurs.

We now prove Theorem 3.4.1.

Proof of Theorem 3.4.1:

Consider Algorithm 3.4.2, PollardLambda. This algorithm implements the kangaroo catching ideas discussed above, from which its correctness follows. It remains to show that the claimed complexities are satisfied. Since the mapping $f : \mathbb{F}_q^* \leftarrow S$ is pseudorandom, the movement of the kangaroos through the field is also pseudorandom. By Lemma 3.3.3, we must randomly select $O(\sqrt{q-1})$ elements from $\mathbb{F}_q^*$ before we expect one to have been chosen twice. Since we have set $N = \lfloor\sqrt{q-1}\rfloor$, the tame kangaroo occupies $O(\sqrt{q-1})$ randomly selected points in $\mathbb{F}_q^*$. We expect that very few bounds by the wild kangaroo, corresponding to the random selection of more points in $\mathbb{F}_q^*$, must ocurr before we see a collision and the wild kangaroo lands on a point previously occupied by the tame kangaroo. Once this occurs, the wild kangaroo must make at the very most N further bounds to land on the trap. Thus the total number of bounds completed by both kangaroos is $2N$ plus the "very few" bounds required before the collision ocurrs. Not counting these very few bounds gives us a total of $2N = O(\sqrt{q-1})$ bounds. Implementing each bound requires multiplying a field element by a power of α. If we have these $|S|$ powers precomputed then this is simply a field multiplication. $O(\sqrt{q-1})$ filed multiplications gives the algorithm a time complexity of $O(\sqrt{q-1})$.

At any stage of the algorithm, all that we have stored is the position of a kangaroo, a field element requiring $O(\log(q))$ storage, and the value of a kangaroo's distance recorder, an integer in $\mathbb{Z}_{q-1}$ also requiring $O(\log(q))$. This proves the claimed space complexity.

□

Algorithm 3.4.2. PollardLambda($\mathbb{F}_q, \alpha, \beta, A, B, S, f$)

Input: A finite field $\mathbb{F}_q$, a primitive element α of $\mathbb{F}_q$, $\beta \in \mathbb{F}_q$, integers $A, B \in \mathbb{Z}_{q-1}$ such that $A \leq \log_(\alpha)(\beta) \leq B$, a set of integer "jump lengths" S and a pseudorandom function $f : \mathbb{F}_q^* \to S$.
Output: $\log_\alpha(\beta)$.

---

1. $N \leftarrow \lfloor \sqrt{q-1} \rfloor$.
2. $i \leftarrow 0$, $x_i \leftarrow \alpha^B$, $d_i \leftarrow 0$.
3. **While** $i < N$ **do**:
   (a) $x_i \leftarrow \alpha^{f(x_i)} x_i$.
   (b) $d_i \leftarrow d_i + f(x_i)$.
   (c) $i \leftarrow i + 1$.
4. $j \leftarrow 0$, $y_j \leftarrow \beta$, $d'_j \leftarrow 0$.
5. textbfWhile $d'_j < B - A + d_N$ **do**:
   (a) $y_j \leftarrow \alpha^{f(y_j)} y_j$.
   (b) $d'_i \leftarrow d'_i + f(y_i)$. **If** $y_j = x_N$ **do**:
      i. **Return** $B + d_N - d'_j \pmod{q-1}$.
6. Pick new S and/or f and go to Step 2.

Example

As an example of Pollard's λ-method, we again consider the computation of $\log_5(178)$ in $\mathbb{F}_{257}$. This logarithm was computed in our example of Pollard's ρ-method earlier (see 3.3) and found to be 26. Knowing this, we use $[0, 50]$ as our interval $[A, B]$. For our set of bound lengths S we take:

$$S = \{s_1 = 1, s_2 = 2, s_3 = 4, s_4 = 16\},$$

with the pseudorandom function f defined by $f(\alpha) = s_i$ if $\alpha \equiv i \pmod 4$. We allow our tame kangaroo to complete $N = \lceil \sqrt{256} \rceil = 16$ bounds. We release the tame kangaroo from the position $x_0 = 5^{50} \equiv 116 \pmod{257}$ and observe the subsequent movement through the field shown in Table 3.4.

We then begin the wild kangaroo at the position $y_0 = 178$ and observe the movement shown in table 3.5.

We see that after 12 bounds, the wild kangaroo occupies position $y_{12} = 69$, which was previously occupied by the tame kangaroo, with $x_8 = 69$. This collision is marked in bold on the tables above. Due to the kangaroos sharing jumping behaviour, the wild kangaroo now follows the tame kangaroo's path until it reaches the trap placed at x_{16}. As shown, this ocurrs at y_{20}.

We compute the desired logarithm by observing:

$$x_{16} = y_{20} \Rightarrow \alpha^{50+d_{16}} = \beta\alpha^{d'_{20}} \Rightarrow \alpha^{50+71} = \beta\alpha^{95} \Rightarrow \beta = \alpha^{50+71-95} = \alpha^{26}.$$

i	1	2	3	4	5	6	6	8
x_i	66	130	38	106	201	142	85	**69**
d_i	1	5	9	13	17	19	23	**25**
i	9	10	11	12	13	14	15	16
x_i	183	82	107	194	203	39	186	86
d_i	27	35	39	47	51	59	67	71

Table 3.4: Tame kangaroo movement in the Pollard λ-method's computation of $\log_5(178)$ in $\mathbb{F}^*_{257}$.

i	1	2	3	4	5	6	6	8
y_i	226	157	70	60	43	126	108	26
d'_i	4	8	10	14	15	23	27	28
i	9	10	11	12	13	14	$\cdots$	20
y_i	59	143	168	**69**	183	82	$\cdots$	86
d'_i	32	40	48	**49**	51	59	$\cdots$	95

Table 3.5: Wild kangaroo movement in the Pollard λ-method's computation of $\log_5(178)$ in $\mathbb{F}^*_{257}$.

So Pollard's λ-method has given us $\log_5(178) = 26$ in $\mathbb{F}^*_{257}$, in agreement with Pollard's ρ-method.

Parallelisation

The Pollard λ-method is well suited to parallelisation. We consider briefly consider three examples of this.

1. An immediately obvious way to parallelise the λ-method is to have each of the P processors run the algorithm unmodified, each searching a distinct subinterval of $[A, B]$ of length $(B-A)/P$. This clearly achieves a running time of $O(\sqrt{(B-A)/P})$. While this is indeed an improvement, much better improvements can be achieved using the following two methods.

2. In the same paper in which they considered parallelisation of Pollard's ρ-method, van Oorschot and Wiener [77] considered parallelisation of the λ-method. The approaches to parallelising both of Pollard's methods are essentially identical. Again we see that a near linear speedup can be achieved.

3. In 2000, Pollard published a paper [61] in which he revisited the ρ- and λ-methods, detailing improvements which had been published by other authors since his first paper [60] and improving the complexity analysis of the λ-method. In this paper he presents a modification of van Oorschot and Wiener's parallelisation of the λ-method which eliminates a minor problem, in which "useless" collisions between sequences would occasionally ocurr .

3.5 Pohlig-Hellman Method

Theorem 3.5.1. Pohlig Hellman Algorithm

There exists a deterministic algorithm which solves the DLP in $\mathbb{F}_q^*$ in time $O(\sqrt{p})$, where p is the greatest prime divisor of $q-1$.

The basis for the this algorithm, which was proposed by M. Hellman and S. Pohlig[2] in 1978 [57], is the Chinese remainder theorem. Suppose we wish to compute logarithms in $\mathbb{F}_q^*$ and the order of this group $q-1$ factors into prime powers as $q-1 = \prod_{i=1}^{k} p_i^{e_i}$, where the p_i are distinct primes and the e_i are natural exponents. Then if we can compute:

$$\begin{aligned} x_1 &\equiv \log_\alpha(\beta) \pmod{p_1^{e_1}}, \\ x_2 &\equiv \log_\alpha(\beta) \pmod{p_2^{e_2}}, \\ &\vdots \\ x_k &\equiv \log_\alpha(\beta) \pmod{p_k^{e_k}}, \end{aligned}$$

the Chinese remainder theorem will allow us to recover, from $x_1, x_2, \ldots, x_k$, $\log_\alpha(\beta)$ modulo $\prod_{i=1}^{k} p_i^{e_i} = q-1$, i.e. to recover $\log_\alpha(\beta)$ exactly.

The Pohlig-Hellman algorithm finds the above x_i via the following result, which shows that computing the x_i is equivalent to solving a DLP in cyclic subgroup of $\mathbb{F}_q^*$ (recall that the DLP is well posed in any finite cyclic groups). In this sense, the method reduces one DLP to several "smaller" DLPs. For this reason, the method is sometimes called the Pohlig-Hellman *reduction*.

Lemma 3.5.2. Pohlig Hellman Reduction

Let $x \equiv \log_\alpha(\beta) \pmod{p^e}$. Write x in the form:

$$x = c_0 + c_1 p + c_2 p^2 + \ldots c_{e-1} p^{e-1}.$$

Then the coefficients c_i satisfy:

$$c_i = \log_{\overline{\alpha}}(\overline{\beta}_i),$$

where $\overline{\alpha} = \alpha^{(q-1)/p}$ and $\overline{\beta}_i = (\beta\gamma_i^{-1})^{n/p^{i+1}}$, with $\gamma_i = \alpha^{c_0 + c_1 p + \ldots + c_{i-1}p^{i-1}}$, where c_{-1} is taken to be 0.

Proof: We note that $\overline{\alpha}$ is a field element of order p, as $\overline{\alpha}^p = \alpha^{(q-1)/p} = \alpha^{q-1} = 1$. Consequently, at any stage we may reduce any exponent of $\overline{\alpha}$ modulo p. We will do this several times in what follows.

[2]It is generally acknowledged that the same method was discovered earlier by R. Silver, but this work was not published. Pohlig and Hellman's discovery was independent. The method is sometimes referred to as the "Pohlig-Hellman-Silver" method for this reason.

Observe:

$$\begin{aligned}
\overline{\beta}_i &= (\beta\gamma_i^{-1})^{(q-1)/p^{i+1}} \\
&= (\alpha^{\log_\alpha(\beta)-c_0-c_1p-\ldots-c_{i-1}p^{i-1}})^{(q-1)/p^{i+1}} \\
&= (\alpha^{x_i-c_0-c_1p-\ldots-c_{i-1}p^{i-1}})^{(q-1)/p^{i+1}} \\
&= (\alpha^{c_ip^i+\ldots+c_{e-1}p^{e-1}})^{(q-1)/p^{i+1}} \\
&= (\alpha^{(q-1)/p^{i+1}})^{c_ip^i+\ldots+c_{e-1}p^{e-1}} \\
&= (\alpha^{(q-1)/p})^{c_i+\ldots+c_{e-1}p^{e-1-i}} \\
&= \overline{\alpha}^{c_i}
\end{aligned}$$

from which the result follows.

□

We may use this result to construct an algorithm which computes $\log_\alpha(\beta)$ by solving a series of smaller DLPs. This proves Theorem 3.5.1:

Proof of Theorem 3.5.1:

Consider the following algorithm:

Algorithm 3.5.3. PohligHellman

Input: A finite field $\mathbb{F}_q$, the canonical factorisation of $q-1 = \prod_{i=1}^{k} p_i^{e_i}$, a primitive element α of $\mathbb{F}_q$ and $\beta \in \mathbb{F}_q$.
Output: $\log_\alpha(\beta)$.

---

1. **For** i from 1 to k **do**:
   (a) $p \leftarrow p_i$, $e \leftarrow e_i$.
   (b) $\overline{\alpha} \leftarrow \alpha^{(q-1)/p}$.
   (c) $\gamma \leftarrow 1$.
   (d) **For** j from 0 to $e-1$ **do**:
      i. $\overline{\alpha} \leftarrow \alpha^{(q-1)/p}$.
      ii. $\gamma \leftarrow \gamma\alpha^{c_{j-1}p^{j-1}}$.
      iii. $\overline{\beta} \leftarrow (\beta\gamma^{-1})^{(q-1)/p^{j+1}}$.
      iv. $c_j \leftarrow \log_{\overline{\alpha}}(\overline{\beta})$.
   (e) $x_i \leftarrow c_0 + c_1p + c_2p^2 + \ldots + c_{e-1}p^{e-1}$.
2. Recover $\log_\alpha(\beta)$ from $x_1, x_2, \ldots, x_k$ using the Chinese remainder theorem.
3. **Return** $\log_\alpha(\beta)$.

The correctness of this algorithm follows from Lemma 3.5.2. It is easy to observe that on the j-th iteration of the **For** loop, γ is equal to γ_j as defined in the lemma, and $\overline{\beta} = \overline{\beta}_j$. Thus c_j is indeed given by the computed logarithm.

The time required to reassemble $\log_\alpha(\beta)$ from the x_i is dominated by the time required to solve the sub-DLPs. The largest of these sub-DLPs is posed in a subgroup of $\mathbb{F}_q^*$ of order p, where p is the largest prime divisor of $q-1$. If this

sub-DLP is solved using Shanks' Baby-Step Giant-Step method (see Section 3.2) or Pollard's ρ-method (see Section 3.3) then the time required will be $O(\sqrt{p})$. Note that here we have used the fact that these are generic discrete logarithm algorithms, since the subgroups of $\mathbb{F}_q^*$ in which the sub-DLPs are posed are not the multiplicative groups of finite fields.

□

For large finite fields, the extra computational effort required to exploit Pohlig-Hellman reduction is small compared to that required to solve the largest sub-DLP. If we solve the sub-DLPs using one of the generic DLP algorithms of this chapter, say Shanks' Baby-Step Giant-Step method or Pollard's ρ-method, the DLP in $\mathbb{F}_q^*$ can be solved in time $O(\sqrt{p})$, where p is the largest prime divisor of $q-1$. In light of this, the manner in which $q-1$ factorises is of extreme importance in the implementation of cryptography. For a cryptographic primitive based on the infeasibility of the DLP in $\mathbb{F}_q$ to be infeasible, q must not only be large, but must be chosen such that $q-1$ has as many large prime divisors as possible, so that the sub-DLPs resulting from a Pohlig-Hellman reduction are still challenging.

We note that in the case of fields $\mathbb{F}_{2^n}$ of characteristic 2, it is possible to choose n so that $2^n - 1$ is in fact a prime, in which case the Pohlig-Hellman reduction is of absolutely no benefit to an attacker. A prime number of the form $2^n - 1$ is known as a *Mersenne prime*. Mersenne primes have been well studied and are actively searched for. As of September, 2006, a total of 44 Mersenne primes are known, many of them suitable for use in cryptography. For example, choosing $n = 1279, n = 2203$, or $n = 2281$ makes $\mathbb{F}_{2^n}$ large enough for modern cryptographic purposes and also makes $2^n - 1$ a Mersenne prime. For fields of odd characteristic, it is never possible for $q-1$ to be a prime, as q is always odd, hence $q-1$ is even and so divisible by 2. However, there are still choices of q which minimise the usefullness of the Pohlig-Hellman reduction. For instance, there exist prime numbers p known as *Sophie Germaine primes*, which have the property that $2p+1$ is also a prime. The corresponding primes $2p+1$ are sometimes called *safe primes*, since if p is large and we set $q = 2p+1$, then $\mathbb{F}_q$ is a large finite field and $q - 1 = 2p$ has p as a large prime divisor, making DLP-based cryptography in $\mathbb{F}_q$ safe from Pohlig-Hellman attacks. More details regarding Mersenne and Sophie Germaine primes can be found in the book by Ribenboim [63].

Chapter 4

Factorisation of Polynomials over Finite Fields

4.1 Introduction and Motivation

In this chapter we momentarily divert our attention from the computation of discrete logarithms to consider the computational problem of factoring polynomials over finite fields. While this problem has many applications in various areas of mathematics, and is interesting in its own right, our motivation for considering polynomial factoring is its applicability to a certain class of discrete logarithm algorithms which we consider in the next chapter.

It is well known that any polynomial over a finite field may be uniquely factored into a product of polynomials irreducible over that field. We term this the *canonical factorisation* of the polynomial, and it is this factorisation which we are interested in computing. In this chapter, if we speak of "factoring" a polynomial with no further qualification, we refer to finding its canonical factorisation. Any non-trivial factorisation of a polynomial which is not the canonical factorisation shall be referred to as a *partial factorisation*. We will see that partial factorisations are useful for reducing the problem of factoring an arbitrary polynomial to that of factoring a polynomial with some special form, whose properties we may exploit. In this chapter we present two partial factorisations and algorithms for computing them, and two algorithms for factoring polynomials of a certain form. Taken together, the algorithms presented here enable us to factor an arbitrary polynomial over any finite field.

We concern ourselves with the factorisation of $f(x) \in \mathbb{F}_q[x]$, where $f(x)$ is a monic polynomial of degree n. We take $f(x)$ to have m distinct irreducible factors, and we denote the canonical factorisation of $f(x)$ by:

$$f(x) = \prod_{i=0}^{m-1} p_i(x)^{e_i}, \tag{4.1}$$

where each $p_i(x)$ is irreducible over $\mathbb{F}_q$ with degree n_i (so that $n = \sum_{i=0}^{m-1} e_i n_i$) and each $e_i \in \mathbb{N}$. Note that the restriction that $f(x)$ be monic causes no loss of

generality; given a non-monic polynomial $f(x)$ with leading coefficient $\alpha \neq 1$, we may factor $\alpha^{-1}f(x)$, which is monic, and then include an extra irreducible factor of α.

4.2 Some Partial Factorisations

All commonly used modern polynomial factoring algorithms work by first computing a simple partial factorisation of the relevant polynomial $f(x)$, say $f(x) = f_1(x)f_2(x)\dots f_k(x)$, where $k < n$. It is clear that if we can factor each $f_i(x)$ then we can factor $f(x)$. The partial factorisations used are chosen so that the polynomials $f_i(x)$ share some particular property which admits their being factored in an algorithmic manner. Two such partial factorisations are the "squarefree factorisation" and the "distinct degree factorisation". We describe both of these factorisations in this section, give simple, deterministic algorithms for computing them and analyse their complexity.

4.2.1 Squarefree Factorisation

The central concept of this subsection is that of a "squarefree polynomial" over a finite field, which we define as follows in analogy to the number theoretic concept of a squarefree integer:

Definition 4.2.1. Squarefree Polynomial

A polynomial $f(x) \in \mathbb{F}_q[x]$ is called a *squarefree polynomial* if its canonical factorisation (4.1) satisfies $e_i = 1$ for $i = 1, 2, \dots, n$. If a polynomial $f(x)$ is not squarefree, then we call the highest degree squarefree polynomial dividing $f(x)$ the *squarefree part* of $f(x)$.

In Section 4.3 we will see a polynomial time (for fixed q or n), deterministic algorithm for factoring any squarefree polynomial, known as *Berlekamp's algorithm*. This algorithm can in fact be used to factor arbitrary polynomials by virtue of the following theorem, which is the main result of this subsection:

Theorem 4.2.2. Squarefree Factorisation

There exists an algorithm which, given as input a polynomial $f(x) \in \mathbb{F}_q[x]$, returns as output a number of polynomials $f_1(x), f_2(x), \dots, f_k(x)$ such that each $f_i(x)$ is squarefree and $f(x) = \prod_{i=1}^{k} f_i(x)$, using $O(n^3)$ multiplications in $\mathbb{F}_q$.

The partial factorisation of $f(x)$ given in Theorem 4.2.2 is called a *squarefree factorisation*. To prove Theorem 4.2.2, we require the following definition and some associated lemmas.

Definition 4.2.3. Formal Derivative of a Polynomial

If $f(x)$ has coefficients:

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n,$$

then the *formal derivative* of $f(x)$ is the polynomial denoted $f'(x)$ of degree $n-1$ given by:

$$f'(x) = \sum_{i=1}^{n} i a_i x^{i-1} = a_1 + 2a_2 x + \ldots + n a_n x^{n-1}.$$

The reader may verify that, even though the formal derivative of a polynomial is an explicit definition rather than the consequence of some general notion of a "derivative", some familiar results from differential calculus still hold for the formal derivative. In particular, for $f(x), g(x) \in \mathbb{F}_q[x]$, the formal derivative obeys the *power rule*:

$$(f(x)^e)' = e f(x)^{e-1} f'(x),$$

and the *product rule*:

$$(fg)'(x) = f'(x)g(x) + f(x)g'(x).$$

We use this fact to prove the following two lemmas. In these lemmas, if $g(x)^e$ divides $f(x)$ for some integer $e > 1$, then $g(x)$ is called a *repeated factor* of $f(x)$ with *exponent* e. Thus, a squarefree polynomial is a polynomial which contains no repeated factors.

Lemma 4.2.4. *If $g(x)$ is a repeated factor of $f(x)$ with exponent e then it is also a (possibly repeated) factor of $f'(x)$ with exponent $e' = e - 1$.*

Proof: Set $h(x) = f(x)/g(x)^e$, so that $f(x) = g(x)^e h(x)$. Then, using the power rule and product rule:

$$\begin{aligned} f'(x) &= e g(x)^{e-1} g'(x) h(x) + g(x)^e h'(x) \\ &= g(x)^{e-1} \left(e g'(x) h(x) + g(x) h'(x) \right). \end{aligned}$$

□

Corollary 4.2.5. *If $f(x) \in \mathbb{F}_q[x]$, the polynomial $f(x)/\gcd(f(x), f'(x))$ is squarefree.*

Proof: Suppose $f(x)$ has repeated irreducible factors $q_i(x)$ with respective exponents e_i. By Lemma 4.2.4 each $q_i(x)$ is a factor of $f'(x)$ with exponent $e_i - 1$, i.e.:

$$f(x) = \left(\prod_{i=0}^{k} q_i(x)^{e_i} \right) h_1(x) \quad \text{and} \quad f'(x) = \left(\prod_{i=0}^{k} q_i(x)^{e_i - 1} \right) h_2(x),$$

where $h_1(x)$ and $h_2(x)$ are the squarefree parts of $f(x)$ and $f'(x)$, respectively. Hence:

$$\gcd(f(x), f'(x)) = \left(\prod_{i=0}^{k} q_i(x)^{e_1 - 1} \right) \gcd(h_1(x), h_2(x))$$

and so

$$f(x)/\gcd(f(x), f'(x)) = \left(\prod_{i=0}^{k} q_i(x) \right) \frac{h_1(x)}{\gcd(h_1(x), h_2(x))}.$$

Since each $q_i(x)$ is irreducible, the bracketed term of the final equality is squarefree and since $h_1(x)$ is squarefree so too is the non-bracketed term. By the definition of $h_1(x)$ these two terms are relatively prime and so their product must also be squarefree.

□

We note in advance a perculiarity whereby the above corollary may sometimes become trivial. If the polynomial $f(x)$ is such that there exists a second polynomial $g(x)$ with $f(x) = g(x)^{\mathrm{char}(\mathbb{F}_q)}$, then by the power rule $f'(x) = \mathrm{char}(\mathbb{F}_q)g(x)^{\mathrm{char}(\mathbb{F}_q)-1} = 0$. In this case, $\gcd(f(x), f'(x)) = f(x)$ and so $f(x)/\gcd(f(x), f'(x)) = 1$, which is trivially squarefree. We may be able to find a non-trivial factor of $f(x)$ in this case by computing $\gcd(g(x), g'(x))$.

We are now in a position to prove Theorem 4.2.2.

Proof of Theorem 4.2.2: Consider the following algorithm:

Algorithm 4.2.6. SQUAREFREE

Input: A polynomial $f(x) \in \mathbb{F}_q[x]$.
Output: Squarefree polynomials $f_0(x), ..., f_k(x)$, with $f(x) = \prod_{i=0}^{k} f_i(x)$.

---

1. $i \leftarrow 1$.
2. **While** $f(x) \neq 1$, **do**:
   (a) **If** $\gcd(f(x), f'(x)) \neq 0$ **do**:
      i. $f_i(x) \leftarrow f(x)/\gcd(f(x), f'(x))$.
      ii. $f(x) \leftarrow f(x)/f_i(x)$.
      iii. $i \leftarrow i + 1$.
   (b) **else do**:
      i. Set k to the highest power of $\mathrm{char}(\mathbb{F}_q)$ which divides all powers of x in $f(x)$.
      ii. $g(x) \leftarrow f(x)^{1/k}$.
      iii. $f_i(x) \leftarrow g(x)/\gcd(g(x), g'(x))$.
      iv. $f(x) \leftarrow f(x)/f_i(x)$.
      v. $i \leftarrow i + 1$.
3. **Return** $\{f_1(x), \ldots, f_i(x)\}$.

The correctness of this algorithm is fairly clear. If $\gcd(f(x), f'(x)) \neq 0$ then the previously mentioned peculiarity does not ocurr and so by Corollary 4.2.5, each $f_i(x)$ produced in this case is squarefree, as we require them to be. If the GCD *is* equal to zero, then we find a factor $g(x)$ of $f(x)$ for which this is not the case, and extract a squarefree factor of $f(x)$ from $g(x)$. It is also clear that the $f_i(x)$ multiply to give $f(x)$. On every pass through the **While** loop, a factor of $f(x)$ is removed, so eventually $f(x)$ must be irreducible or a power of an irreducible polynomial, in which case $f_i(x) = \gcd(f(x), f'(x)) = 1$ and $f(x)/f_i(x) = 1$, at which point the algorithm terminates.

This algorithm requires at most n traversals of the **While** loop, since a polynomial of degree n can clearly have no more than n squarefree factors. Each

run through the loop requires the computation of one or possibly two formal derivatives, a GCD and a polynomial division, with the inputs in each case having degree at most n. Computing the derivative of a degree n polynomial requires $O(n)$ multiplications, while computing the GCD and the division each require $O(n^2)$ and thus these steps dominate the time required in each loop. The total cost of the algorithm is thus $O(n^3)$.

□

An interesting paper by D. Panario [56], which studies the properties of random polynomials over finite fields, contains the following unexpected result, which we do not prove:

Theorem 4.2.7. Proportion of Polynomials which are Squarefree

If $n \geq 2$, then the proportion of polynomials in $\mathbb{F}_q[x]$ of degree n which are squarefree is $1 - 1/q$.

Proof: Omitted.

So, for large fields almost all polynomials will be squarefree, and even for $\mathbb{F}_2$ there is a probability of 1/2 that a randomly selected polynomial will be squarefree. Thus we expect that we will most often have no need to apply SQUAREFREE to $f(x)$ and may proceed directly to an algorithm for factoring squarefree polynomials, such as Berlekamp's algorithm.

4.2.2 Distinct Degree Factorisation

In this subsection we discuss a partial factorisation which we can compute for any polynomial and which will allow us to use an algorithm presented later to find such a polynomial's canonical factorisaiton.

An algorithm which computes the canonical factorisation of a polynomial whose irreducible factors are all of equal degree is called an *equal degree factorisation algorithm*. In Section 4.4 we will see a polynomial time, probabilistic equal degree factorisation algorithm, known as the *Cantor-Zassenhaus algorithm*. This algorithm can in fact be used to find the canonical factorisation of an arbitrary polynomial by virtue of the following theorem, which is the main result of this subsection:

Theorem 4.2.8. Distinct Degree Factorisation

There exists an algorithm which, given as input a squarefree polynomial $f(x) \in \mathbb{F}_q[x]$, returns as output n polynomials $f_1(x), f_2(x), \ldots, f_n(x)$ such that $f(x) = \prod_{i=1}^{k} f_i(x)$ and the irreducible factors of $f_i(x)$ have degree i, using $O(n^3 \log(q))$ multiplications in $\mathbb{F}_q$.

The partial factorisation of $f(x)$ given in Theorem 4.2.8 is called a *distinct degree factorisation*. To prove Theorem 4.2.8, we require the following result:

Proof of Theorem 4.2.8:

Consider the following algorithm:

Algorithm 4.2.9. DistDeg

Input: A squarefree polynomial $f(x) \in \mathbb{F}_q[x]$ of degree n.
Output: Polynomials $f_0(x), ..., f_{k-1}(x)$ such that all irreducible factors of $f_i(x)$ have degree i, with $f(x) = \prod_{i=0}^{k-1} f_i(x)$.

---

1. $i \leftarrow 0$.
2. **While** $\deg(f(x)) > 0$, **do**:
   (a) $f_i(x) \leftarrow \gcd(f(x), x^{q^i} - x)$.
   (b) $f(x) \leftarrow f(x)/f_i(x)$.
   (c) $i \leftarrow i + 1$.
3. **Return** $\{f_1(x), \ldots, f_{i-1}(x)\}$.

The correctness of this algorithm is based on Gauss' product Lemma (Lemma 2.4.5). We illustrate the idea of the algorithm by first describing the first two passes through the **While** loop. When $i = 1$, the product lemma gives that $x^{q^i} - x$ is the product of all monic irreducible polynomials of degree dividing 1, i.e. of degree 1. Thus $\gcd(f(x), x^{q^1} - x)$ is the product of all irreducible factors of $f(x)$ of degree 1. We label this product $f_1(x)$ and remove it from $f(x)$. Now when $i = 2$, $x^{q^i} - x$ is the product of all monic irreducible polynomials of degree dividing 2, i.e. of degree 1 or 2. Since $f(x)$ has no factors of degree 1 (these having been removed in the previous step), $\gcd(f(x), x^{q^1} - x)$ is the product of all factors of $f(x)$ of degree 2, which we then remove.

The algorithm continues in this manner. At each value of i, $\gcd(f(x), x^{q^1} - x)$ is the product of all irreducible factors of $f(x)$ of degree dividing i. All factors of degree less than i will have been removed from $f(x)$ at previous steps, so the GCD is in fact all the factors of $f(x)$ of degree exactly i. If the highest degree irreducible factor of $f(x)$ has degree m, say, then $f(x)/f_m(x) = 1$, so $\deg(f(x)) \not> 0$ and the algorithm terminates. Note that at this point, i has been increased to $m + 1$, so the algorithm only returns $f_1(x), \ldots, f_{i-1}(x)$.

To establish the complexity of the algorithm, we observe that it requires n GCD computations and n divisions. The most time consuming GCD is the final one computed, when $i = n$, and involves polynomials of degree at most q^n. By the same technique of modular reduction used in GenIrred (Algorithm 2.4.9), this GCD can be computed in time $O(n^2 \log(q))$. The most time consuming division is the first one, where $f(x)$ of degree n is divided by $f_1(x)$. This takes time $O(n^2)$, which is dominated by the time taken for the GCD computation. The value of k is at most n, so the total time taken by the algorithm is at most $O(n^3 \log(q))$.

□

In Section 4.4 we will see a polynomial factorisation algorithm which is designed for factorising polynomials with equal order irreducible factors.

An alternative method for computing distinct degree factorisation, which utilises matrix multiplication and may sometimes be more efficient, is described in [5].

We observe that if all of the irreducible factors of $f(x)$ have distinct degrees, then its distinct degree factorisation is, in fact, its canonical factorisation and there is no need to employ an equal degree factorisation algorithm to factor $f(x)$. The previously mentioned paper of Panario [56] contains the following result regarding the probability of this occurring:

Theorem 4.2.10. *As $n \to \infty$, the probability that the distinct degree factorisation of a random polynomial of degree n over $\mathbb{F}_q$ is equal to that polynomial's canonical factorisation is asymptotic to:*

$$c_q = \prod_{k=1}^{\infty} \left(1 + \frac{I_k}{q^k - 1}\right) (1 - q^{-k})^{I_k},$$

where I_k is the number of monic irreducible polynomials of degree k over $\mathbb{F}_q$. In particular, $c_2 = 0.6656\ldots$.

Proof: Omitted.

4.3 Berlekamp's Algorithm

The material presented in this subsection will ultimately enable us to prove the following main result:

Theorem 4.3.1. ***Berlekamp's Factorisation Algorithm***

There exists a deterministic algorithm which, given as input a squarefree polynomial $f(x) \in \mathbb{F}_q[x]$, returns as output the canonical factorisation of $f(x)$ using $O(n^3 q)$ multiplications in $\mathbb{F}_q$.

The algorithm which constitutes proof of Theorem 4.3.1 is due to E. Berlekamp, and was the first general algorithm for factoring polynomials over finite fields. It was first published in 1967 [3], but this paper is nowadays most readily found in its republished form in Berlekamp's book [4]. Strictly speaking, Berlekamp's algorithm is an algorithm for factoring squarefree polynomials, but it can be used in conjunction with SQUAREFREE (Algorithm 4.2.6) to factor any polynomial. The algorithm is deterministic and comprised of linear algebra over $\mathbb{F}_q$ and the computation of polynomial GCDs. The algorithm is most efficient when applied to polynomials over fields of small order.

We begin with a lemma, which we do not prove here.

Lemma 4.3.2. *A polynomial $g(x) \in \mathbb{F}_q[x]$ satisfies:*

$$g(x)^q - g(x) = \prod_{s \in \mathbb{F}_q} (g(x) - s)$$

Proof: Omitted.

The following result provides the theoretical core of Berlekamp's algorithm:

Theorem 4.3.3. *If $f(x)$ is an arbitrary monic polynomial in $\mathbb{F}_q[x]$, and $g(x)$ is a monic polynomial in $\mathbb{F}_q[x]$ satisfying $g(x)^q \equiv g(x) \pmod{f(x)}$ then:*

$$f(x) = \prod_{s \in \mathbb{F}_q} \gcd(f(x), g(x) - s) \tag{4.2}$$

Proof: We establish the equality by showing that each side divides the other. Since both sides of the equality are monic polynomials, they must thus be equal.

$g(x)$ satisfies $g(x)^q \equiv g(x) \pmod{f(x)}$ and hence $f(x) \mid g(x)^q - g(x)$ or, via Lemma 4.3.2, $f(x) \mid \prod_{s\in\mathbb{F}_q} \gcd(f(x), g(x) - s)$. So the lefthand side of (4.2) divides the righthand side.

Each term $\gcd(f(x), g(x) - s)$ of the product in (4.2) clearly divides $f(x)$. Further, $g(x) - s_i$ and $g(x) - s_j$ are relatively prime if $i \neq j$ and hence so too are $\gcd(f(x), g(x) - s_i)$ and $\gcd(f(x), g(x) - s_j)$. Thus we have the result that $\prod_{s\in\mathbb{F}_q} \gcd(f(x), g(x) - s) \mid f(x)$. So the righthand side of (4.2) divides the lefthand side.

□

Provided $g(x)$ has degree ≥ 1, this theorem will provide a non-trivial factorisation of $f(x)$. Berlekamp's algorithm is primarily an algorithm for finding such $g(x)$. These polynomials can be found using linear algebra, as we now detail.

The set of all possible irreducible factors of $f(x)$ can be considered a subset of the factor ring $R = \mathbb{F}_q[x]/\langle f(x)\rangle$. If we consider this ring as an algebra (the vectorspace being over $\mathbb{F}_q$), then the reader may easily verify that the set of polynomials $g(x) \in \mathbb{F}_q[x]$ satisfying the congruence $g(x)^q \equiv g(x) \pmod{f(x)}$ form a subalgebra of R. This is known as the *Berlekamp subalgebra* of R, which we shall denote $\mathcal{B}$. The strategy of Berlekamp's algorithm is to determine a basis for $\mathcal{B}$ so that we may easily generate polynomials $g(x)$ which will yield a non-trivial factor of $f(x)$ via Theorem 4.3.3. The method by which a basis is found relies upon some results concerning the following matrix.

Definition 4.3.4. The Berlekamp Matrix of a Polynomial

Let $f(x) \in \mathbb{F}_q[x]$ be a polynomial of degree n. Define the n polynomials:

$$Q_i(x) = \sum_{j=0}^{n} q_{i+1,j+1} x^j \quad \text{for } i = 0, ..., n-1,$$

where the coefficients q_{ij} are chosen such that $x^{iq} \equiv Q_i(x) \pmod{f(x)}$. The *Berlekamp matrix* of $f(x)$ is the $n \times n$ matrix $\mathcal{Q} = [q_{i,j}]$, i.e.:

$$\mathcal{Q} = \begin{bmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,n} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ q_{n,1} & q_{n,2} & \cdots & q_{n,n} \end{bmatrix}.$$

We have the following result:

Lemma 4.3.5. *A polynomial $g(x) \in \mathbb{F}_q[x]$ of degree $< n$, given by:*

$$g(x) = \sum_{i=0}^{n} g_i x^i, \tag{4.3}$$

with g_n and possibly other coefficients $= 0$, is an element of $\mathcal{B}$ if and only if the row vector $\boldsymbol{g} = (g_0, g_1, \ldots, g_n)$ is in the null space of the corresponding $\mathcal{Q}$, i.e.:

$$\boldsymbol{g}\mathcal{Q} = \boldsymbol{0}.$$

Proof: Observe that:

$$\begin{aligned} g(x)^q &= g(x^q) = \sum_{i=0}^{n} g_i x^{iq} \\ &\equiv \sum_{i=0}^{n} g_i Q_i(x) \pmod{f(x)} \\ &= \sum_{i=0}^{n} \left(\sum_{j=0}^{n} g_i q_{i+1,j+1} x^j \right) \\ &= \sum_{j=0}^{n} \left(\sum_{i=0}^{n} g_i q_{i+1,j+1} \right) x^j \\ &= \sum_{j=0}^{n} [\boldsymbol{g}\mathcal{Q}]_j x^j \end{aligned}$$

The polynomial in the final equality is the zero polynomial if and only if $[\boldsymbol{g}\mathcal{Q}]_j = 0$ for $j = 0, 1, \ldots, n$, i.e. if and only if $\boldsymbol{g}\mathcal{Q} = \mathbf{0}$.

□

The reader may be more familiar with the nullspace of a matrix being the set of column vectors which when premultiplied by the matrix to give zero, rather than the set of row vectors which give zero when postmultiplied. The above result may be rewritten in these terms if we redefine the Berlekamp matrix to be the transpose of its current definition. This practice is sometimes seen in the literature.

We immediately have the corollary:

Corollary 4.3.6. *A polynomial $g(x) \in \mathbb{F}_q[x]$ of degree $< n$, given by* (4.3) *satisfies $g(x)^q - g(x) \equiv 0 \pmod{f(x)}$ if and only if the row vector $\boldsymbol{g} = (g_0, g_1, \ldots, g_n)$ is in the null space of the corresponding $\mathcal{Q} - I$, where I is the $n \times n$ identity matrix, i.e.:*

$$\boldsymbol{g}(\mathcal{Q} - I) = \mathbf{0} \tag{4.4}$$

We need one final result before we proceed to Berlekamp's algorithm.

Lemma 4.3.7. *The number of distinct irreducible polynomial factors of $f(x) \in \mathbb{F}_q[x]$ is equal to the dimension of the null space (the* nullity*) of the corresponding matrix $\mathcal{Q} - I$.*

Proof: Suppose $f(x)$ has n factors, with cannonical decomposition:

$$f(x) = \prod_{i=0}^{n} p_i(x)^{e_i}. \tag{4.5}$$

We seek an expression for the number of polynomials $g(x) \in \mathbb{F}_q[x]$ satisfying $g(x)^q \equiv g(x) \pmod{f(x)}$, or, equivalently, the number of $g(x)$ satisfying:

$$f(x) \mid g(x)^q - g(x),$$

i.e.

$$\prod_{i=0}^{n} p_i(x)^{e_i} \Big| \prod_{s \in \mathbb{F}_q} (g(x) - s)\,, \tag{4.6}$$

from (4.5) and Lemma 4.3.2. Since the $p_i(x)^{e_i}$ are relatively prime and so too are the $g(x) - s$, (4.6) is satisfied if and only if each $p_i(x)^{e_i}$ divides $g(x) - s_i$ for some $s_i \in \mathbb{F}_q$, or, equivalently, $g(x) \equiv s_i \pmod{p(x)^{e_i}}$.

The Chinese remainder theorem for polynomials states that given any set $S = \{s_1, s_2, \ldots, s_n\} \subset \mathbb{F}_q$ there exists a unique $g(x) \pmod{f(x)}$ satisfying $g(x) \equiv s_i \pmod{p(x)^{e_i}}$ for $i = 1, 2, 3, \ldots, n$. Since there are q^n possible choices for the set S, there are q^n polynomials $g(x) \in \mathbb{F}_q[x]$ with $g(x)^q \equiv g(x) \pmod{f(x)}$.

Now, by Corollary 4.3.6, these q^n polynomials correspond in an injective fashion with the row vectors in the null space of the matrix $\mathcal{Q} - I$, so there must be q^n vectors in this null space. Since these vectors are over the field $\mathbb{F}_q$, basic linear algebra gives that the number of vectors in the null space is $q^{\dim}$, where dim is the dimension of the null space. Hence we have dim $= n$.

□

We have now developed the necessary machinery to prove Theorem 4.3.1. Before continuing, we note briefly that the previous lemma can be used as the basis for an irreducibility test. If a polynomial is reducible, then the nullity of the corresponding $\mathcal{Q} - I$ will be > 1. If the nullity is 1, the polynomial is either irreducible or a power of an irreducible polynomial. We may use SQUAREFREE (Algorithm 4.2.6) to distinguish between these two situations and, if the polynomial is a power of an irreducible polynomial, extract that irreducible polynomial. This observation was first made by Butler [13].

Proof of Theorem 4.3.1:

Consider Algorithm 4.3.8, BERLEKAMP. We establish the correctness of this algorithm as follows. The first three steps compute the Berlekamp matrix of $f(x)$, reduce it and then use the reduced form to find a basis B matrix's nullspace. By Corollary 4.3.6, the polynomials corresponding to the vectors of B form a basis for the Berlekamp subalgebra of R.

The variable i shall count the number of factors of $f(x)$ computed by the algorithm at any point. It is naturally initialised to 0. By Lemma 4.3.7, we know that $f(x)$ has $m = |B|$ irreducible factors. Thus we shall continue to find new factors until $i = m$, at which point the known factors must constitute the canonical factorisation. This justifies the condition on which the **While** loop terminates. The "best" factorisation known at any point is stored in the set F. What we mean by "best" will become clear.

Each pass of the **While** loop uses Theorem 4.3.1 to compute a partial factorisation of $f(x)$, using a particular polynomial $g(x)$ in the Berlekamp subalgebra. The vector corresponding to this polynomial is then removed from B to avoid it being reused in later passes of the loop, which would waste effort. At the end of the **For** loop, the set F' contains all the non-trivial factors of $f(x)$ which were found using Theorem 4.3.1 with the current choice of polynomial $g(x)$. We compare F' to F. Initially, $F = \{f(x)\}$, so the factorisation contained in F' will certainly be "better". We update F by setting it equal to F'. On subsequent passes through the **While** loop, we compare the two factorisations as follows. The elements in F multiply to $f(x)$ and so to do the elements of F'. If we set $A = F \setminus F'$ and $B = F' \setminus F$ then it follows from this that the product of the

Algorithm 4.3.8. BERLEKAMP($f(x)$)

Input: A squarefree polynomial $f(x) \in \mathbb{F}_q[x]$.
Output: The canonical factorisation of $f(x)$.

---

1. Compute the Berlekamp matrix $\mathcal{Q}$ of $f(x)$.
2. Reduce $\mathcal{Q} - I$ to row echelon form.
3. Compute a basis B for $\mathcal{B}$, the nullspace of $\mathcal{Q} - I$.
4. $i \leftarrow 0, m \leftarrow |B|$.
5. $F = \{f(x)\}$.
6. **While** $i < m$ **do**:
   (a) Select a vector $\boldsymbol{g} \in B$ and set $g(x)$ to the corresponding polynomial.
   (b) $B \leftarrow B \setminus \{\boldsymbol{g}\}$.
   (c) $F' \leftarrow \{\}$.
   (d) **For** every $\alpha \in \mathbb{F}_q$ **do**:
      i. $h(x) \leftarrow \gcd(f(x), g(x) - \alpha))$.
      ii. **If** $\deg(h_i(x)) > 0$ **do**:
         A. $F' \leftarrow F' \cup \{h(x)\}$.
   (e) Compare F' to F. Update F if any factors have been refined.
   (f) $i \leftarrow |F|$.

---

elements of A equals the product of the elements of B. If $|B| > |A|$ then we set $F \leftarrow (F \setminus A) \cup B$. Since the elements of A and the elements of B multiply to give the same product, this does not compromise the fact that the elements of F multiply to give $f(x)$, and since $|B| > |A|$ the factorisation contained in F is "improved" by this update. The algorithm continues to refine the factorisation in F until it is the canonical factorisation.

We establish the complexity of the algorithm in stages. The first stage is the computation of the Berlekamp matrix. This is equivalent to the computation of $x^{iq} \pmod{f(x)}$ for $i = 0, 1, \ldots, n$. $x^0 \pmod{f(x)} = 1$ regardless of $f(x)$, so no time is required to compute the first row of the matrix. Finding $x^q \pmod{f(x)}$ can be done using exponentiation by squaring with $\log q$ multiplications and modular reductions of polynomials of degree at most $2n$, requiring $O(n^2 \log(q))$ multiplications in $\mathbb{F}_q$. We can then find each subsequent row by multiplying the previous row by $x^q \pmod{f(x)}$ and reducing. Each such multiplication and reduction takes time $O(n^2)$ and we require approximately n of them, so computing the rest of the matrix requires $O(n^3)$ multiplications. Thus the total time required to compute the Berlekamp matrix is $O(n^3 + n^2 \log(q))$.

Once the Berlekamp matrix has been computed, it can be reduced using Gaussian elimination. The complexity of this method is well known to be (and can easily be seen to be) $O(n^3)$. The time taken in this stage is dominated by

the time taken to compute the matrix.

We now consider the time required for the several GCD computations needed to find non-trivial factors of $f(x)$. Applying Theorem 4.3.1 with one particular $g(x)$ requires the computation of q GCDs of polynomials of degree at most n, thus taking time $O(n^2q)$. In the worst case scenario, we will have to use all m of the polynomials $g(x)$ in the Berlekamp subalgebra to obtain the canonical factorisation. This requires $O(mn^2q)$. Since $m \leq n$, this is at most time $O(n^3q)$, making this the time dominating part of the algorithm, which establishes the complexity stated in the theorem.

□

The worst case scenario used in the proof of Theorem 4.3.1 (i.e. that we must use every possible $g(x)$) is pessimistic. In practice, less $g(x)$ may be rquired and Berlekamp's algorithm often performs much better than suggested by this analysis.

We note now that the requirement that $f(x)$ be squarefree is not strictly necessary. We have used it only to determine when we have achieved the canonical factorisation, since if $f(x)$ is squarfree the number of irreducible factors is the dimension of the Berlekamp subalgebra. We can easily use the machinery of Berlekamp's algorithm to produce non-trivial factors of arbitrary polynomials. It will then be necessary, however, to perform irreducibility testing on the factors produced to determine when we have the canonical factorisation.

The factor of q in the complexity of Berlekamp's algorithm means that it is feasible only for factorising polynomials over fields of small order q. For large q, the number of GCD computations required is excessive or even infeasible. In 1970 Berlekamp published a paper [5] presenting an improved version of the algorithm which performed better for large finite fields. Space restrictions prevent a discussion of this improvement here, however we mention that the basis of the improvement is a method for determining which $\alpha \in \mathbb{F}_q$ provide non-trivial factors via Theorem 4.3.1 so that fewer than q GCDs must be computed to obtain a partial factorisation.

4.4 The Cantor-Zassenhaus Algorithm

The material presented in this subsection will ultimately enable us to prove the following main result:

Theorem 4.4.1. Cantor-Zassenhaus Factorisation Algorithm

There exists a probabilistic algorithm which, given as input a polynomial $f(x) \in \mathbb{F}_q[x]$ whose irreducible factors are all of equal degree, returns as output with probability $> 1/2$ at least two non-trivial factors $f_1(x)$ of $f(x)$ using $O(n^2 \log(q))$ multiplications in $\mathbb{F}_q$.

The algorithm which constitutes a proof of this theorem was proposed by D. Cantor and H. Zassenhaus in 1981 [15]. It is a probabilistic algorithm, and conceptually much simpler than Berlekamp's algorithm. The Cantor-Zassenhaus algorithm is an equal degree factorisation algorithm, but can be used in conjunction with SQUAREFREE and DISTDEG (Algorithms 4.2.6 and 4.2.9, respectively) to factor arbitrary polynomials. It involves no linear algebra, simply polynomial

exponentiation and GCD computation. The size of the field q affects only the probability of success of a single run of the algorithm, without substantially affecting the running time.

The following theorem describes the algebraic structure of the set of possible factors of a polynomial $f(x)$ which has equal degree irreducible factors. This structure is exploited by the Cantor-Zassenhaus algorithm to find a non-trivial factor.

Theorem 4.4.2. *Let $f(x) \in \mathbb{F}_q[x]$, $f = p_1(x)p_2(x)\dots p_n(x)$, where the $p_i(x)$'s are distinct and irreducible in $\mathbb{F}_q[x]$ and $\deg(p_i(x)) = d$, for $i = 1, 2, \dots, n$ (and hence $\deg(f(x)) = nd$). Let R be the factor ring $R = \mathbb{F}_q[x]/\langle f(x)\rangle$. Let S be the direct sum of fields:*

$$S = \bigoplus_{i=1}^{n} \frac{\mathbb{F}_q[x]}{\langle p_i(x)\rangle}.$$

For any polynomial $g(x) \in R$, let $g_i(x)$ denote the reduction of $g(x)$ modulo $p_i(x)$, i.e. $g(x) \equiv g_i(x) \pmod{p_i(x)}$. Then the function $\varphi : R \to S$ defined by:

$$\varphi(g(x)) = (g_1(x) + \langle p_1(x)\rangle, g_2(x) + \langle p_2(x)\rangle, \dots, g_n(x) + \langle p_n(x)\rangle)$$

is a ring isomorhphism, i.e.:

$$\frac{\mathbb{F}_q[x]}{\langle f(x)\rangle} \cong \bigoplus_{i=1}^{n} \frac{\mathbb{F}_q[x]}{\langle p_i(x)\rangle}. \tag{4.7}$$

Proof: Familiar properties of congruence are sufficient to show that φ is a homomorphism. If $g(x) \equiv g_i(x) \pmod{p_i(x)}$ and $h(x) \equiv h_i(x) \pmod{p_i(x)}$ then $g(x) + h(x) \equiv g_i(x) + h_i(x) \pmod{p_i(x)}$, giving:

$$\begin{aligned}
\varphi(g(x) + h(x)) &= (g_1(x) + h_1(x) + \langle p_i(x)\rangle, \dots, g_n(x) + h_n(x) + \langle p_n(x)\rangle) \\
&= (g_1(x) + \langle p_i(x)\rangle, \dots, g_n(x) + \langle p_n(x)\rangle) + \\
&\quad (h_1(x) + \langle p_i(x)\rangle, \dots, h_n(x) + \langle p_n(x)\rangle) \\
&= \varphi(g(x)) + \varphi(h(x)),
\end{aligned}$$

and also $g(x)h(x) \equiv g_i(x)h_i(x) \pmod{p_i(x)}$, giving:

$$\begin{aligned}
\varphi(g(x)h(x)) &= (g_1(x)h_1(x) + \langle p_i(x)\rangle, \dots, g_n(x)h_n(x) + \langle p_n(x)\rangle) \\
&= (g_1(x)\langle p_i(x)\rangle, \dots, g_n(x) + \langle p_n(x)\rangle) \times \\
&\quad (h_1(x) + \langle p_i(x)\rangle, \dots, h_n(x) + \langle p_n(x)\rangle) \\
&= \varphi(g(x))\varphi(h(x)).
\end{aligned}$$

We now show that φ is bijective, and hence an isomorphism. Consider any $s = (g_1(x) + \langle p_1(x)\rangle, \dots, g_n(x) + \langle p_n(x)\rangle) \in S$. Since the $p_i(x)$ are relatively prime, the Chinese remainder theorem for polynomials gives that there exists a *unique* polynomial $g(x)$ modulo $p_1(x)p_2(x)\dots p_n(x) = f(x)$ (i.e. a unique polynomial $g(x) \in R$) such that $g(x) \equiv g_i(x) \pmod{p_i(x)}$ - that is, a unique polynomial $g(x) \in R$ such that $\varphi(g(x)) = s$. So every element of S is the image under φ of a unique element of R and hence φ is injective. Now, the i^{th} summand of the direct sum in S is isomorphic to the field $\mathbb{F}_{q^d}$, since $p_i(x)$ is an irreducible polynomial of degree d. Consequently $|S| = (q^d)^n = q^{nd} = \deg f(x) = |R|$. Since an injective function between two sets of equal cardinality is surjective,

φ is a surjection, hence a bijection and hence an isomorphism.

□

The details of the Cantor-Zassenhaus algorithm are slightly different for each of two possible cases - the case where the field order q is a power of an odd prime and the case where q is a power of 2. In the discussion that follows, until mentioned otherwise, we shall assume that q is a power of an odd prime. The minor changes required for using the algorithm in fields of characteristic 2 are presented at the end of the discussion.

The following theorem provides the core of the Cantor-Zassenhaus Algorithm:

Theorem 4.4.3. Cantor-Zassenhaus Algorithm *Let $a(x) \in R$ be a polynomial satisfying:*

$$a(x) \neq 0, \pm 1 \tag{4.8}$$

and also:

$$\varphi(a(x)) = (a_1, a_2, \ldots, a_n), \quad a_i \in \{0, -1, 1\}, i = 0, 1, \ldots, n. \tag{4.9}$$

Let $A = \{i \mid a_i = 0\}, B = \{i \mid a_i = 1\}$ and $C = \{i \mid a_i = -1\}$. Then, if any two of A, B, or C is non-empty, we have the non-trivial factors:

$$\gcd(f(x), a(x)) = \prod_{i \in A} p_i(x), \tag{4.10}$$

$$\gcd(f(x), a(x) - 1) = \prod_{i \in B} p_i(x), \tag{4.11}$$

$$\gcd(f(x), a(x) + 1) = \prod_{i \in C} p_i(x). \tag{4.12}$$

Proof: We prove the first equation: the others follow in a similar way. For every $i \in A$, $f(x) \equiv 0 \pmod{p_i(x)}$ and so $p_i(x) \mid f(x)$. Conversely, for every $i \notin A$, $f(x) \not\equiv 0 \pmod{p_i(x)}$ and so $p_i(x) \not| f(x)$. Recalling that the GCD of two polynomials is the product of all irreducible polynomials which divide both polynomials, the result follows from these observations.

□

Thus, we see that polynomials $a(x) \in R$ satisfying (4.8) and (4.9) provide a non-trivial factorisation of $f(x)$. In most cases they do not provide the canonical factorisation that is sought, but by applying Theorem 4.4.3 recursively, the non-trivial factorisation obtained can be successively refined until this is found. This is the strategy used by the Cantor-Zassenhaus algorithm.

The only issue remaining is that of how such polynomials $a(x)$ can be obtained. They cannot be constructed via the Chinese remainder theorem as the irreducible moduli $p_i(x)$ are (of course!) unknown. The Cantor-Zassenhaus algorithm exploits the fact that the $p_i(x)$ have equal degrees to generate such $a(x)$ at random in the manner described below.

Let $b(x)$ be a randomly selected non-constant polynomial from R. If we set $m = (q^d - 1)/2$ then, since φ is an isomorphism, it is clear that:

$$\varphi(b(x)^m) = (b_1(x)^m, b_2(x)^m, \ldots, b_n(x)^m).$$

Recall that each summand in (4.7) was isomorphic to $\mathbb{F}_{q^d}$ and hence $\alpha^{q^d-1} = 1$ for any $\alpha \neq 0$ in each summand. Bringing this consideration to bear on the $b_i(x)$ above we see that if $b_i(x) \neq 0$ then $b_i(x)^m = (b_i(x)^{q^d-1})^{1/2} = 1^{1/2} = \pm 1$. If $b_i(x) = 0$ then $b_i(x)^m = 0$. Thus we see that $b(x)^m$ satisfies (4.9). If $b(x)^m \neq 0, \pm 1$ then it also satisfies (4.8) and so it may be used to apply Theorem 4.4.3 to $f(x)$, obtaining a non-trivial factorisation.

We are now in a position to prove Theorem 4.4.1.

Proof of Theorem 4.4.1:

Consider the following algorithm:

Algorithm 4.4.4. CantorZass

Input: A polynomial $f(x) \in \mathbb{F}_q[x]$ with equal degree irreducible factors.
Output: A set $S = \{f_1(x), \ldots, f_k(x)\}$ of k non-trivial factors of $f(x)$, $2 \leq k \leq 3$.

---

1. Select $b(x) \in R$ such that $b(x) \neq 0, \pm 1$ at random, using a uniform probability distribution over all such polynomials
2. Compute $a(x) = b(x)^{(q-1)/2}$.
3. $f_1(x) \leftarrow \gcd(f(x), a(x))$.
4. $f_2(x) \leftarrow \gcd(f(x), a(x) + 1)$.
5. $f_3(x) \leftarrow \gcd(f(x), a(x) - 1)$.
6. **Return** $\{f_i(x) | f_i(x) \neq 1\}$.

The correctness of this algorithm is clear. Since $b(x)$ is selected so that $b(x) \neq 0, \pm 1$, $a(x)$ will be such that by Theorem 4.4.3 at least 2 and perhaps 3 of the $f_i(x)$ will be non-trivial factors.

We now consider the complexity of the algorithm. The exponent $(q-1)/2$ is clearly $< q$ and so the exponentiation in step 2 may be computed using ExpSquare with $O(\log(q))$ multiplications by $b(x)$. Since $\deg(b(x)) < n$, these multiplications require at most $O(n^2)$ multiplications in $\mathbb{F}_q$, so the exponentiation takes time $O(n^2 \log(q))$. Once $g(x)$ has been computed, we require the computation of one or more GCDs. The polynomials whose GCD we require have degrees at most n, so each GCD takes time $O(n^2)$. The total time for a single run of the algorithm is hence $O(n^2(1 + \log(q))) = O(n^2 \log(q))$. This is polynomial in the input size $n \log(q)$, as $O(n^2 \log(q)) < O((n \log(q))^2)$.

The Cantor-Zassenhaus algorithm is a probabilistic algorithm, since polynomials $b(x)$ are selected at random until a polynomial with a certain property is found. The number of such polynomials which must be generated and tested will obviously influence the complexity of the algorithm. We are interested primarily in the expected number of $b(x)$'s which must be generated before one is found satisfying $b(x)^m \neq 0, \pm 1$. To obtain this we estimate the probability that a random $b(x)$ is not appropriate for use in the algorithm, supposing that the $b(x)$ are selected at random via a uniform probability distribution.

A $b(x)$ is rejected if $b(x)^m = 0, \pm 1$, i.e. if either $b_i(x) = 0, b_i(x) = 1$, or $b_i(x) = -1$ for $i = 1, 2, \ldots, n$. Recall that each $b_i(x)$ is an element of a field of order q^d.

The only $b_i(x) \in \mathbb{F}_{q^d}$ satisfying $b_i(x)^m = 0$ is $b_i(x) = 0$, so the probability that a randomly selected $b_i(x)$ satisfies $b_i(x)^m = 0$ is $1/q^d$. Thus probability that a random $b(x)$, uniquely determined by n random $b_i(x)'s$, is $(1/q^d)^n = 1/q^{nd}$, or $1/q^k$ if we label $\deg(f(x)) = nd$ as k.

Now suppose $b_i(x)^{q^d-1} = 1$ for all $b_i(x) \in \mathbb{F}_{q^d}$ and so $b_i(x)^m = 1$ for $(q^d-1)/2$ choices of $b_i(x)$ and $b_i(x)^m = -1$ for $(q^d-1)/2$ choices. The probability that a random $b(x)$ is equal to ± 1 is thus $2(1/2(1-q^{-d}))^n = 2^{1-n}(1-q^{-d})^n$.

The total probability that $b(x)$ is rejected is hence $1/q^n d + 2^{1-n}(1-q^{-d})^n$. Now $0 < q^{-d} < 1$, so $(1-q^{-d}) < 1$, hence $(1-d^{-d})^n < 1$ and so $2^{1-n}(1-q^{-d})^n < 2^{1-n}$.

So the probability that a randomly chosen $b(x)$ is suitable for use with Theorem 4.4.1 is $\geq 1/2$ and we expect to have to select no more than 2 before this occurs.

□

We now consider the case where the field order q is a power of 2. We cannot proceed as above in this case, as if q is a power of 2, and hence even, $q-1$ is odd and so $m = (q-1)/2$ is not an integer. Fortunately, the above process readily adapts to this important case. We consider two subcases:

If $q \equiv 1 \pmod 3$ then $q-1 \equiv 0 \pmod 3$, so $(q-1)/3$ is an integer. If we set $\rho = \alpha^{(q-1)/3}$, where α is a primitive element of $\mathbb{F}_q$, then ρ is a primitive third root of unity. It is clear that ρ^2 is also such a root, and that ρ and ρ^2 are the only such roots in the field.

We may now proceed largely as in the odd characteristic case. If we choose a random $b(x) \in R$ as before and compute $b(x)^m$ for this new m, we see that each $b_i(x) \neq 0$ satisfies $b_i(x)^m = (b_i(x)^{q^d-1})^{1/3} = 1^{1/3} \in \{1, \rho, \rho^2\}$. We can then make use of (4.10), (4.11), and (4.12) above, and, defining $D = \{i \mid a_i(x) = \rho\}$, $E = \{i \mid a_i(x) = \rho^2\}$, the extra equations:

$$\gcd(f(x), a(x) - \rho) = \prod_{i \in D} p_i(x),$$

$$\gcd(f(x), a(x) - \rho^2) = \prod_{i \in E} p_i(x),$$

which follow in the same way as the other three. Thus we can factor in fields of order $q \equiv 1 \pmod 3$.

If instead $q \equiv 2 \pmod 3$ these roots ρ do not exist. However, we may append these roots and then use the method above to factor $f(x)$ over the quadratic extension field $\mathbb{F}_q(\rho)$. We may then combine factors which are conjugate over $\mathbb{F}_q$ to recover the factorisation in the original field.

4.5 Further Reading

The polynomial factorisation algorithms of Berlekamp and Cantor-Zassenhaus discussed here are undoubtedly the best known and most widely used algorithms for the problem, but they are by no means the only algorithms or even the fastest.

An algorithm due to H. Niederreiter [53], based upopn a differential equation, is also well known. It relies heavily upon linear algebra, like Berlekamp's

algorithm. A number of modifications and improvements to this algorithm exist. A discussion of these may be found in [48].

The work of V. Shoup on polynomial factorisation is further reading of particular interest. [33] (with E. Kaltofen) presents a probabilistic factoring algorithm which is asymptotically the fastest known. For a fixed q, it factors a degree n polynomial in $\mathbb{F}_q[x]$ in time $O(n^{1.815})$. [78] (with J. von zur Gathen) presents an earlier probabilistic factoring algorithm. [69] presents a deterministic method which is asymptotically the fastest determinstic algorithm known.

Finally, we mention that the factorisation of polynomials over finite fields is also often used to compute the canonical factorisation of polynomials with integer coefficients, i.e. polynomials in $\mathbb{Z}$. A finite field factoring method can be used to factor such a polynomial modulo a prime p, i.e. in the field $\mathbb{F}_p$. A lemma due to Hensel then gives a factorisation modulo a power p^n of this prime. If p^n is large enough, the factorisation in $\mathbb{Z}$ may be "recovered". A discussion of this methodology can be found in [48].

Chapter 5

Index Calculus Algorithms for Finite Fields

All the discrete logarithm algorithms which we have seen so far have been so-called *generic algorithms*, which can be used to solve the discrete logarithm problem in any finite cyclic group. In this chapter we consider a class of non-generic algorithms, called index calculus algorithms, which can only be used to solve the DLP in particular groups. The multiplicative groups of finite fields are among the groups for which index calculus algorithms can be used. They are of particular interest because the loss of applicability to arbitrary groups allows a substantial decrease in complexity. While it is a proven result than any generic discrete logarithm algorithm must perform at least $O(\sqrt{n})$ group operations to solve the DLP in a finite cyclic group of order n (a complexity which is exponential in bitsize of the group order, with $\sqrt{n} = e^{1/2 \log n}$), index calculus algorithms allow us to compute logarithms in so-called *subexponential time*.

The term "index calculus" describes a family of discrete logarithm algorithms which are built around the same central idea, but in which the details of implementation may vary depending upon the fields being considered and application specific requirements. We sometimes refer to these methods collectively as "the" index calculus method, even though there is no canonical example of an algorithm following the template. The general idea of the index calculus method is generally considered to have originated in [49]. The first published method explicitly considering the computation of discrete logarithms is due to L. Adleman [1]. All of these authors published index calculus algorithms applicable to prime order fields $\mathbb{F}_p$. The first generalisation to prime power fields $\mathbb{F}_{p^n}$ was due to Hellman and Reyneri [32]. In 1984, Blake, Fuji-Hara, Mullin and Vanstone [8] published an important improvement to this method for arbitrary $\mathbb{F}_{p^n}$ and also an improvement specific to fields of characteristic 2, $\mathbb{F}_{2^n}$. The latter idea was then extended by D. Coppersmith [20] in the same year to give a drastically faster algorithm for $\mathbb{F}_{2^n}$.

The focus of our attention in this chapter will be somewhat biased toward the prime power order fields $\mathbb{F}_{p^n}$. This is because we have developed more theory regarding the representations of these fields than of prime order fields $\mathbb{F}_p$. For example, we have a result from Section 2.4 for counting irreducible

polynomials but no corresponding result for counting prime numbers. We have also considered polynomial factorisation in Chapter 4 but have not considered integer factorisation. The results from these considerations will be used in developing estimates of the asymptotic complexity of index calculus algorithms. We shall not develop such estimates for $\mathbb{F}_p$ algorithms. Towards the end of the chapter we discuss one such algorithm briefly, one in some detail and give reference to others.

5.1 Generic Description

In this section we discuss the general framework which all algorithms bearing the name "index calculus" have in common.

The index calculus algorithm relies crucially upon the property of discrete logarithms (1.1) given in Chapter 1, stating that the discrete logarithm of a product of field elements is equal to the sum of the logarithms of the individual elements modulo $q-1$. The key concept behind an index calculus algorithm for computing discrete logarithms is to use this property of logarithms to "build up" new logarithms from a database of known logarithms. A necessary step in achieveing this for a field $\mathbb{F}_q$ is the establishment of a so-called *factor base*: a subset $T = \{p_1, p_2, \ldots, p_t\} \subset \mathbb{F}_q$ which is "small" with the property that the following set is "large":

$$\{h \in \mathbb{F}_q | h = \prod_{i=1}^{t} p_i^{e_i}, p_i \in T, e_i \in \mathbb{Z}^+\}. \tag{5.1}$$

That is, the factor base is a small set of field elements such that a large proportion of the elements in $\mathbb{F}_q$ may be factorised into powers of elements from the factor base. The notions of small and large used here are relative and their importance will become clear. The need for a factor base is the reason why index calculus methods are not applicable to arbitrary cyclic groups. For some groups, such as the group of points on an elliptic curve, there are no known appropriate choices of a factor base.

With a factor base established, an index calculus method then generates a number of linear equations relating the discrete logarithms (to the desired base) of the elements of the factor base, i.e. equations of the form:

$$\sum_{i=1}^{n} a_i \log_g(t_i) \equiv b_i \pmod{q-1}.$$

These equations are usually refered to as *relations*. There are many different methods for generating relations, depening upon the field of interest, and some are more efficient than others. However, the property of logarithms given in Theorem 1.1.2 is always used. We shall see a number of methods for generating relations throughout this chapter. Once a number of linearly independent relations which is equal to the number of factor base elements has been computed, these relations contain enough information to uniquely determine the logarithm of each factor base element. This can be done in a number of ways. The logarithms of all the factor base elements form the database of known elements which we spoke of earlier.

Suppose that we ultimately wish to determine the discrete logarithm of h. With a well-chosen factor base, for most h we will be able to factorise h into powers of factor base elements, or else factorise $g^m h$ in this way for some integer m. Supposing that we can achieve this and we find that:

$$g^m h = \prod_{i=1}^{n} t_i^{e_i},$$

then (1.1) gives us the equation:

$$\log_g(h) \equiv \sum_{i=1}^{n} e_i \log_g(t_i) - m \pmod{q-1}.$$

With the $\log_g(t_i)$ terms known after solving the linear system arising from our collected relations, we may simply evaluate this to find the desired logarithm.

The importance of the factor base being "small" and the set (5.1) being "large" is now clear: The larger the factor base, the greater the time which must be spent generating relations and then solving or manipulating the resulting system of linear equations. The smaller the factor base, the less likely that elements will be able to be factorised into products of factor base elements as required.

We note that much of the work involved in the index calculus algorithm can be "recycled" for further computations. Once we have have chosen a factor base, we only need to complete the process of generating relations and solving the linear system a single time and can then compute the discrete logarithm of any field element which factors into powers of factor base elements for only the cost of that factorisation and the evaluation of a linear equation.

We now discuss some specific applications of this general framework to particular finite fields.

5.2 A Simple Index Calculus Method for $\mathbb{F}_{p^n}$

In this section, we shall present a simple index calculus algorithm for computing discrete logarithms in a field $\mathbb{F}_{p^n}$ of prime power order, using our polynomial representation. Our description of this algorithm shall be the most thorough in this chapter. In the following sections we shall consider a number of well known improvements to the various components of this basic algorithm. This algorithm was originally described by Hellman and Reyneri [32] for the special case of fields $\mathbb{F}_{2^k}$ which have characteristic 2, however it easily generalises to fields $\mathbb{F}_{p^n}$ for arbitary primes p, and we describe it in such generality. The generalisation to arbitrary p appears to have first been discussed by Adleman [1]. Because characteristic 2 fields $\mathbb{F}_{2^k}$ are widely used in practice due to their easy implementation, some of the improvements we shall view later are specific to these fields.

Throughout this algorithm, we represent $\mathbb{F}_{p^n}$ in the usual way, as the ring of polynomials $\mathbb{F}_p[x]$ modulo some irreducible polynomial $p(x)$ of degree k. We are interested in finding the logarithm of the element $h(x)$ to the base of the generator $g(x)$.

Choice of a Factor Base

We begin by selecting a factor base T. Recall that our requirements for T are (i) that $|T| = t$ be "small" and (ii) that a large proportion of the elements of $\mathbb{F}_{p^n}^*$ may be written as a product of elements of T. Our choice of factor base is guided by this second condition. We observe that our representation of $\mathbb{F}_{p^n}$ is contained within a larger algebraic structure $\mathbb{F}_p[x]$ which is a unique factorisation domain: every polynomial in $\mathbb{F}_p[x]$, and hence every polynomial in our representation of $\mathbb{F}_{p^n}$ may be uniquely expressed as a product of irreducible polynomnials. This motivates the following procedure for defining our factor base:

Defining a Factor Base

1. Select an integer b from the set $\{1, 2, \ldots, k-1\}$.
2. Set T equal to the set of all monic, irreducible polynomials $p_i(x) \in \mathbb{F}_p[x]$ of degree at most b.

The parameter b is referred to as a *smoothness parameter*. The reason for this is the following definition:

Definition 5.2.1. Smooth Polynomials

A polynomial $f(x) \in \mathbb{F}_q[x]$ is called *b-smooth* if every irreducible factor $f_i(x)$ of $f(x)$ satisfies $\deg(f_i(x)) \leq b$.

Throughout the chapter, we may say that a polynomial is simply "smooth". By this we mean b-smooth with respect to the value of b currently being considered, which should always be clear from context. We note in advance that we are able to use Lemma 2.4.3 to compute the size t of this factor base, and shall do so later.

Phase I: Generation of Relations

We generate linear equations between the logarithms of the irreducible polynomials in our factor base by randomly generating elements of $\mathbb{F}_{p^n}$ in such a way that their logarithm is known, and then attempting to factor these elements over the factor base. The properties of logarithms given in Theorem 1.1.2 then allow us to derive a relation. We proceed as follows:

Generating Relations

1. Select a random integer k, using a uniform distribution on the set $\{1, 2, \ldots, p-1\}$.
2. Compute $h_k(x) = g(x)^k \pmod{p(x)}$.
3. Test to see if $h_k(x)$ is smooth. If it is, compute the canonical factorisation of $h_k(x)$ and if:
$$h_k(x) = \prod_{i=0}^{t-1} p_i(x)^{e_i}, \tag{5.2}$$
then store the relation:
$$k \equiv \sum_{i=1}^{t-1} e_i \log_{g(x)}(p_i(x)) \pmod{p^n - 1},$$

which follows from (5.2) by (1.1). If $h_k(x)$ is not smooth, then return to step 1.

4. Continue until t or more relations have been found.

The steps involved in this method of generating relations may be performed using previously discussed algorithms. The exponentiations may be done using either exponentiation by squaring as discussed in Section 2.3.2 or addition chain exponentiation as discussed in Section 2.3.3. The smoothness testing and computation of canonical factorisations may be done using either Berlekamp's algorithm from Section 4.3 or the Cantor-Zassenhaus algorithm from Section 4.4. We can and later will use this previous material to estimate the amount of work involved in finding our relations in this manner.

Phase II: Solving the Linear System

Suppose now that we have generated (or believe we have generated) t linearly independent relations. We may represent this sytem of linear equations as a $t \times t$ matrix over $\mathbb{Z}_{p^n-1}$. The system may then be solved in any of the many well-known ways, for example using Gauss-Jordan elimination to put the matrix into reduced row echelon form.

It is worth noting that this matrix may often be "sparse", in the sense of having a non-negligible number of zero entries. In this case, performance may be improved by using some of the many techniques which have been developed especially to perform computations on sparse matrices. Again, space limitations prevent any discussion of these techniques. Commonly used methods for solving this system include D. H. Widemann's algorithm of 1986 [79] and a method described by B. LaMacchia and A. Odlyzko in [39], usually referred to as the Lanczos method.

Once the system has been solved, we know the discrete logarithm to the base of interest of any element of the factor base, and hence it is a trivial matter to compute the discrete logarithm of any field element for which we can find a factorisation over the factor base.

Phase III: Computing the Final Logarithm

Once we know the logarithm of all of the elements in the factor base, we may compute the logarithm of a particular field element as follows:

Computing a Logarithm

1. Select a random integer k, using a uniform distribution on the set $\{1, 2, \ldots, p-1\}$.

2. Compute $h^*(x) = h(x)g(x)^k \pmod{p(x)}$.

3. Test to see if $h^*(x)$ is smooth. If it is, compute the canonical factorisation of $h^*(x)$ and if:

$$h^*(x) = \prod_{i=0}^{t-1} p_i(x)^{e_i},$$

then compute $\log_{g(x)}(h(x))$ according to

$$\log_{g(x)}(h(x)) \equiv \sum_{i=1}^{t-1} e_i \log_{g(x)}(p_i(x)) - k \pmod{p^n - 1}.$$

If $h^*k(x)$ is not smooth, then return to step 1.

Complexity Analysis

We now prove a result regarding the complexity of the algorithm described above. Our analysis is not exhaustive, as this is a daunting task. Nevertheless, this algorithm is widely considered to have been analysed rigorously, and we give references which provide the required details. While the algorithm was first discussed by Hellman and Reyneri [32], our analysis is largely guided by the later work of Coppersmith [20] and Odlyzko [54], as the analysis in [32] has been improved upon by these works. Our result is specific to the use of the algorithm in fields of characteristic 2, $\mathbb{F}_{2^n}$. In practice, these are the only fields the algorithm is used for.

The result we seek to prove uses "little O" notation, defined similarly to big O:

Definition 5.2.2. "Little O" notation

A function $f(n)$ is said to be $o(g(n))$ ("little O $g(n)$") if:

$$\limsup_{n\to\infty} \frac{f(x)}{g(x)} = 0$$

We note that $o(1)$ terms, which we shall see often, tend to 0 as $n \leftarrow \infty$.

Our complexity result is the following:

Theorem 5.2.3. Complexity of the Index Calculus Algorithm for $\mathbb{F}_{2^n}$

The described algorithm , when used over $\mathbb{F}_{2^n}$, has asymptotic time complexity:

$$\exp\left((1 + o(1))\sqrt{n\log(n)}\right)$$

as $n, b \leftarrow \infty$ subject to $n^{1/100} \leq b \leq n^{99/100}$.

Proof:

We analyse the complexity of the algorithm phase by phase.

Phase I

We begin by deriving an expression for the size of the factor base, as this will tell us how many relations we need to generate. Recall from our work in irreducible polynomials in Chapter 2 (Lemma 2.4.3 and its corrolaries) that there are approximately $2^m/m$ irreducible polynomials of degree m over $\mathbb{F}_2$. Thus the size of our factor base is approximately:

$$t = \sum_{i=1}^{b} \frac{2^i}{i} < \frac{1}{b}\sum_{i=1}^{b} 2^i \simeq \frac{2^{b+1}}{b} \tag{5.3}$$

Now knowing how many relations we need to generate, we must estimate the work involved in generating each relation. Since relations are generated in a random manner, the amount of work involved is a random variable and are interested in the expected amount.

Each computation of a new $h_k(x)$ and subsequent attempt to derive a relation can be considered as a Bernoulli trial with some probability of "success" (i.e. of $h_k(x)$ being smooth and we can get a relation). If we can compute this probability of success, then we can find the expected number of trials before we find a relation. The probability of success is exactly the probability that a random polynomial in $\mathbb{F}_p[x]$ of degree at most $k-1$ is smooth. To find this probability requires knowing the number of smooth polynomials of a certain degree.

In 1984, I. F. Bake, R. Fuji-Hara, R. C. Mullin and S. A. Vanstone published a set of relations for counting smooth polynomials [8]. The author has identified an error in these published relations and has conferred with Mullin regarding it. The following relations have been agreed upon as correct: Let $N_l(m,k)$ denote the number of monic polynomials over $\mathbb{F}_q$ of degree $\leq m$ whose largest degree irreducible polynomial factors have degree $\leq k$ (i.e. the number of k-smooth polynomials of degree $\leq m$). Similarly, let $N_e(m,k)$ denote the number of monic polynomials over $\mathbb{F}_q$ of degree $\leq m$ whose largest degree irreducible polynomial factors have degree exactly k. Then we have:

1. For $m > 0$:
$$N_e(m,0) = N_l(m,0) = 1.$$

2. For $k \geq m > 0$:
$$N_l(m,k) = \frac{q^{m+1}-1}{q-1}.$$

3. For $k > m > 0$:
$$N_e(m,k) = 0.$$

4. For $m \geq k+1$:
$$N_l(m,k) = N_e(m,k) + N_l(m,k-1).$$

5. For $m \geq k \geq 1$:
$$N_e(m,k) = \sum_{i=1}^{\lfloor m/k \rfloor} \binom{N_e(k,k)+i-1}{i} N_l(m-ik,k-1).$$

6. For $m \geq 1$:
$$N_e(m,m) = N_l(m,m) - N_l(m,m-1).$$

We do not discuss here the derivation of this relations. This task is relatively straightforward.

These six relations provide sufficient information for us to evaluate the functions $N_l(m,k)$ and $N_e(m,k)$ for all possible m and k. However, they do not allow us to do so efficiently: Suppose we wish to use the index calculus algorithm to compute logarithms in $\mathbb{F}_{2^{2048}}$, using the polynomial representation of this field. If we choose a smoothness-bound b, then the number of b-smooth polynomials

in the representation of the field is $N_l(2047, b)$. We are interested in finding this number for many choices of b, to determine a suitable choice of that parameter. An implementation of the above relations by the author required over 36 hours of continuous computation to compute $N_l(2047, b)$ for $b = 1, 2, \ldots, 2047$, on a machine with a 2.0 GHz Intel®Pentium®-M Centrino®processor and 1 GB of RAM, running the NetBSD®3.0 operating system. The same implementation required less than an hour to compute $N_l(1023, b)$ for $b = 1, 2, \ldots, 1023$, suggesting that the computation time for this task increases quickly with m and that computing all the numbers of interest would be infeasible for larger fields. The results of these computations were used to produce Figure 5.1, which shows the proportion of polynomials used in the representation of $\mathbb{F}_{2^{2048}}$ which are b-smooth for all possible choices of b.

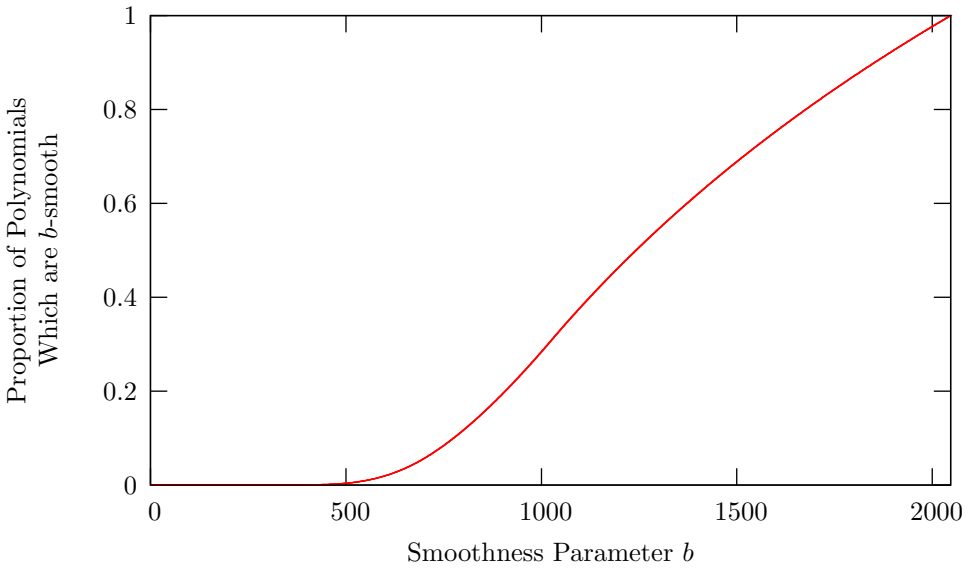

Figure 5.1: Proportion of Polynomials over $\mathbb{F}_2$ of Degree < 2048 Which are b-smooth for $b < 2048$.

The ability to numerically compute the number of smooth polynomials of a certain type is of no use in analysing the complexity of an algorithm. For this purpose, we must rely upon asymptotic results. Using generating functions and the so-called "saddle point method", Odlyzko has derived the following result regarding smooth polynomials over $\mathbb{F}_2$ [54].

Theorem 5.2.4. *Let $N(n, m)$ be the number of b-smooth polynomials of degree n. Then, as $n, b \to \infty$ such that $n^{1/100} \leq b \leq n^{99/100}$:*

$$N(n, b) = 2^n (n/b)^{(1+o(1))n/b},$$

and hence the probability that a random polynomial of degree n is b-smooth is:

$$P(n, b) = e^{(1+o(1))n/b \log(b/n)}. \tag{5.4}$$

Proof: Omitted.

A similar expression can be achieved for polynomials over arbitrary fields. However, these are more complicated and since $\mathbb{F}_2$ is by far the most used case for cryptographic applications we shall be satisfied with the above.

We have so far shown that in order to generate the required number of relations, we expect to have to try about the following number of random values of k:

$$(2^{b+1}/b)(n/b)^{(1+o(1))n/b}. \tag{5.5}$$

We are now interested in the amount of work required to derive, or attempt to derive, a relation from k. This is at least the amount of work required to test $h_k(x)$ for smoothness and at most the amount of work required to factor $h_k(x)$. The simplest way to test a polynomial $h_k(x)$ for smoothness is to simply factor it and inspect the degree of the irreducible factor with highest degree. If this is $\leq b$, then the polynomial is smooth. In fact, we do not have to do quite this much work. Recall the algorithm DistDeg (Algorithm 4.2.9). Given any squarefree polynomial $f(x)$ of degree n, this algorithm outputs $f_1(x), \ldots, f_n(x)$ which multiply to $f(x)$ and which have the property that $f_i(x)$ is a product of irreducible polynomials of degree i. It is clear that $f(x)$ is smooth if and only if $f_{b+1}(x) = \ldots = f_n(x) = 1$. Thus we need only compute a distinct degree factorisation of $h_k(x)$ to test it for smoothness. We may even modify DistDeg to abort as soon as it finds one non-trivial factor $f_i(x)$ for $i > b$, thus saving further time. If $h_k(x)$ is found to be smooth using this test, then we may refine the distinct degree factorisation to the canonical one using either Berlekamp's algorithm or the Cantor-Zassenhaus algorithm and derive a relation. We have shown that polynomial factorisation may be performed in polynomial time. The time taken to perform factorisations thus varies much less substantially with the parameter n than does the the number of factorisations we have to perform. In light of this, we treat the factorisations as taking constant time.

Ignoring polynomial time factors, the running time of phase I of the algorithm is:

$$(2^{b+1}/b)(n/b)^{(1+o(1))n/b}.$$

We may make our choice of the smoothness parameter b so as to minimise this quantity. This can be shown to occur for $b = c_1\sqrt{n\log(n)}$, where $c_1 = 0.8493$ (see [54]). With this choice of b, the running time becomes:

$$\exp\left((c_1\log(2) + \frac{1}{2c_1} + o(1))\sqrt{n\log(n)}\right). \tag{5.6}$$

This concludes our analysis of phase I of the algorithm.

Phase II

Phase II of the algorithm consists of solving a $t \times t$ system of linear equations over $\mathbb{Z}_{2^q-1}$. The time required for this depends upon the method used. Suppose that this system can be solved in t^ω operations for some ω. If we use Gauss-Jordan elimination, it is well known and fairly simple to see that $\omega = 3$. Using a method of Strassen [73] it is possible to achieve $\omega = 2.807$. A method of Coppersmith and Winograd [19] would give $\omega = 2.496$, but this method is generally considered impractical.

All of the above methods are algorithims which do not benefit from any potential sparseness of the system. It turns out that the system arising from an index calculus algorithm is typically quite sparse. Panario [56] shows that a polynomial of degree n over a finite field has an expected $\log(n)$ irreducible factors, counting multiplicity. We may thus expect the matrix corresponding to an index calculus system to have $\log(n)$ non-zero entries in each row, leaving $2^{b+1}/b - \log(n)$ zero entries. The matrix may thus be considered sparse, and techniques which take advantage of this may achieve running times with $\omega \simeq 2$. Using such techniques and our value of b chosen earlier the complexity of phase II is thus:

$$2^{2(b+1)}/b = \exp\left((c_2 + o(1))\sqrt{n\log(n)}\right), \tag{5.7}$$

which is of the same general form as (5.6).

Phase III

It is immediately clear that whatever time is spent by the algorithm in phase III shall be dominated by the first two phases; The time spent trying to find the desired logarithm is the same as the time spent trying to find a single relation in phase I.

Thus the time taken by the first two phases gives the asymptotic complexity of the whole algorithm, which is

$$\exp\left((c + o(1))\sqrt{n\log(n)}\right), \tag{5.8}$$

as required. The constant c is whichever of the constant terms in (5.6) and (5.6) is larger.

□

5.3 An Improved Method for Fields $\mathbb{F}_{p^n}$

An important and well-known improvement to the previous index calculus algorithm for $\mathbb{F}_{p^n}$ was published in 1984 by I. F. Bake, R. Fuji-Hara, R. C. Mullin and S. A. Vanstone [8]. This algorithm has become known as the "Waterloo variant", or "Waterloo algorithm", due to three of the paper's four authors being at the University of Waterloo, Canada at the time of publication. The improvement uses the Euclidean algorithm for polynomials to reduce the amount of work which must be done to establish relations between the factor base logarithms.

In the algorithm above, one generates relations by computing $h_k(x) = g(x)^k \pmod{p(x)}$ and attempting to factor the result over the factor base. A relation is yielded only if $h_k(x)$ is smooth. It is observed in practice (and stands to reason) that the probability of this factorisation being possible decreases as the degree of $h_k(x)$ increases. The Waterloo team use the following result to increase the probability that a useful factorisation is possible, and also decrease the amount of effort involved in the factorisation:

Theorem 5.3.1. *Let $h_k(x) \in \mathbb{F}_q[x]$ satisfy $(n-1)/2 < \deg(h_k(x)) < n$. Then there exist $r(x), t(x) \in \mathbb{F}_q[x]$, both of degree $\leq (n-1)/2$, satisfying:*

$$t(x)f(x) \equiv r(x) \pmod{p(x)}, \tag{5.9}$$

and these can be found in polynomial time.

We do not offer a complete proof of this result. Some details can be found in [8]. The essence of the proof is that polynomials $r(x), t(x)$ satisfying the above may be computed using the Euclidean algorithm for polynomials. Given input polynomials $a(x)$ and $b(x)$, the Euclidean algorithm computes polynomials $s_i(x), t_i(x), r_i(x)$ for $i = -1, 0, 1, 2, \ldots$ satisfying:

$$s_i(x)a(x) + t_i(x)b(x) = r_i(x), \text{for } i \geq -1, \tag{5.10}$$

or $t_i(x)b(x) \equiv r_i(x) \pmod{()a(x)}$. Thus if we apply the Euclidean algorithm with $a(x) = p(x)$ and $b(x) = f(x)$ every $t_i(x)$ and $r_i(x)$ in the generated sequence satisfy (5.9). That there exists in the sequence terms $t_i(x)$ and $r_i(x)$ which also satisfy the required condition on their degrees follows from some inductive arguments. That the polynomials can be found in polynomial time follows from the fact that the entire Euclidean algorith runs in polnomial time and hence so to must a modification of the algorithm which terminates part way through when a certain condition is met.

The Waterloo algorithm generates relations as follows. We generate $h_k(x) = g(x)^k$ as per the previous algorithm, and then use the Euclidean algorithm as above to find $r(x), t(x)$ so that $t(x)h_k(x) \equiv r(x) \pmod{p(x)}$. If $r(x)$ and $t(x)$ are both smooth, then we can derive the relation:

$$\log_{g(x)}(r(x)) - log_{g(x)}(t(x)) \equiv k \pmod{p^n - 1}.$$

The reason that this method is an improvement upon the previous one is that, as mentioned, the probability of a polynomial being smooth decreases as the degree of the polynomial increases. The polynomials $r(x)$ and $t(x)$ which we require to be smooth above are guaranteed to have degrees at most $(n-1)/2$. By contrast, $h_k(x)$ is likely to have a much higher degree, very close to $n - 1$.

Although the Waterloo algorithm is observed in practice to perform much better than our original algorithm, no rigorous result as to its complexity can be obtained, due to uncertainty in the exact relation between $h_k(x)$ and the pair $r(x), t(x)$. However, we can make a seemingly reasonable assumption and obtain an estimate as to the degree of improvement provided by this observation. We consider the case $q = 2^n$, so that we can use 5.4. The probability of successfully deriving a relation using the Waterloo method is at its least when, for a given $h_k(x)$ the resulting polynomials $r(x)$ and $t(x)$ are both of degree $(n-1)/2$. If we assume $r(x)$ and $t(x)$ act as random and independent polynomials selected from a uniform probability distribution over all polynomials of this degree, then the probability that both polynomials are smooth is approximately less than:

$$P(n/2, b)^{-2} \simeq \left(\frac{2b}{n}\right)^{(1+o(1))n/b}$$

By contrast, if we do not use the Euclidean algorithm and instead simply rely on $h_k(x)$ being smooth, our probability of success is at its least when $h_k(x)$ has degree $n - 1$. Our probability of success is then approximately less than:

$$P(n, b)^{-1} \simeq \left(\frac{b}{n}\right)^{(1+o(1))n/b}.$$

Thus using the Euclidean algorithm improves the probability of successfully generating a relation by a factor of approximately $2^{(1+O(1))n/b}$, which can be quite

substantial. For instance, the Waterloo team used their method to compute logarithms in $\mathbb{F}_{2^{127}}$ using $b = 17$, for which the improvement factor is approximately $2^{127/17} \simeq 177$, allowing a computation which would have previously taken just over a week to be performed in one hour. Note, however, that this improvement does not change the asymptotic complexity of the algorithm. It simply reduces the constant factors in this complexity by such an amount that run times are significantly reduced.

5.4 Improved Methods for Fields of Characteristic 2, $\mathbb{F}_{2^n}$

Finite fields of characteristic 2, i.e. those fields $\mathbb{F}_{2^n}$, are of supreme importance in practice, due to their straightforward and efficient implementation on binary computers. They posses an important theoretical property which other finite fields do not: squaring elements of $\mathbb{F}_{2^n}$ is a linear operation: $(\alpha+\beta)^2 = \alpha^2+\beta^2$. This fact was first exploited by the Waterloo team in the same paper which discussed their Euclidean algorithm improvement, through the use of so-called *systematic equations* to assist in the generation of some relations. This work was furthered by D. Coppersmith in 1984 to show that *all* the required relations could be generated in a similar way.

5.4.1 The Waterloo Work

We first discuss the original work of the Waterloo team, published in [8]. The following result is central:

Theorem 5.4.1. *Let $f(x) \in \mathbb{F}_q[x]$ be irreducible, with $\deg(f(x)) = n$, and $g(x) \in \mathbb{F}_q[x]$ be arbitrary. If $m(x)$ is a divisor of $f(g(x))$ then $\deg(m(x))$ is a multiple of n.*

Proof: We prove the result for irreducible divisors $m(x)$ of $f(g(x))$, since if every irreducible factor of a divisor has a degree which is a multiple of n, then so does the divisor itself. We label $f(g(x))$ as $h(x)$ for clarity. Let K be the splitting field of $h(x)$, $\alpha \in K$ a root of $m(x)$ and L the splitting field of $m(x)$. Since $m(x)|h(x)$ and $m(\alpha) = 0$, we must have $h(\alpha) = 0$ and so $g(\alpha)$ is in the splitting field of $f(x)$, which is $R = \mathbb{F}_{q^n}$. Further, $g(\alpha)$ is not contained in any proper subfield of $\mathbb{F}_{q^n}$, since every zero of $f(x)$ generates R. Since $g(\alpha) \in L$ we have $R \subset L$ and hence $n|[L : \mathbb{F}_q]$. Since $m(x)$ is irreducible with splitting field L we have $\deg(m(x)) = [L : \mathbb{F}_q]$ and so $n|\deg(m(x))$.

□

Suppose the field $\mathbb{F}_{2^n}$ is represented as $\mathbb{F}_2/\langle p(x)\rangle$. If the irreducible polynomial $p(x)$ has a particular property then we may use the above result in conjunction with an irreducible polynomial of low degree to obtain a relation. We illustrate with the following example from [8]:

Example

Consider $\mathbb{F}_{2^{127}}$ generated by $p(x) = 1 + x + x^{127}$. The polynomial $g(x) = 1 + x^2 + x^5$ is irreducible over $\mathbb{F}_2$. We have $x^{2^7} = x^{128} \equiv x + x^2 \pmod{p(x)}$,

and so:

$$\begin{aligned} g(x^{2^7}) = g(x + x^2) &= 1 + x^2 + x^4 + x^5 + x^6 + x^9 + x^{10} \\ &= (1 + x^2 + x^3 + x^4 + x^5)(1 + x^3 + x^5) = p_1(x)p_2(x). \end{aligned}$$

Now, squaring is a linear operator over $\mathbb{F}_2$ and so $g(x^{2^7}) = g(x)^{2^7}$, yielding:

$$2^7 \log(g(x)) = \log(p_1(x)) + \log(p_2(x)),$$

a relation between three polynomials of low degree.

By Theorem 5.4.1, the irreducible factors of $g(x^2 + x)$ in this example were required to be multiples of 5 less than 127 and had to sum to 10 (mod 127). The only possibilities were that $g(x^2 + x)$ was an irreducible polynomial of degree 10 or the product of two polynomials of degree 5. Thus this procedure, which required no exponentiation but only the factorisation of a low degree polynomial, was guaranteed to yield a relation among polynomials of low degree. By contrast, the previous procedure required an exponentiation as well as the factorisation of a probably high degree polynomial, and was not guaranteed to produce a useful relation.

An equation generated in the above manner is termed a *systematic equation*. If we allow $g(x)$ to range over all irreducible polynomials of low degree (where the concept of "low" depends upon our chosen smoothness parameter b), we may collect several relations between polynomials whose degrees must be multiples of the same low degree. Experimental evidence suggests that this method can yield a substantial number of relations at little cost. By way of example, there are 226 irreducible polynomials of degree at most 10 in the representation of $\mathbb{F}_{2^{127}}$ as $\mathbb{F}_2[x]/\langle x^{127} + x + 1\rangle$ - the Waterloo team found that 142 linearly independent systematic equations could be produced for them, almost 63% of the number required to find the logarithm of each of them. More than 71% of the required equations for a smoothness parameter of 9 can be found in the same way.

The defining property of $p(x) = 1 + x + x^{127}$ which was used in the above example was the fact that we could find a power of 2, $k = 2^7$, such that x^k was congruent to a low degree polynomial $\pmod{p(x)}$, in this case $x^2 + x$. For any given irreducible polynomial $p(x)$ which is used to construct a polynomial representation of a field of characteristic 2, the more terms x^{2^n} which are congruent to low degree polynomials modulo $p(x)$, the more effective the method of systematic equations. The Waterloo team considered the representation of fields $\mathbb{F}_{2^n}$ as vector spaces over $\mathbb{F}_2$ with basis $\{\alpha, \alpha^2, \ldots, \alpha^{k-1}\}$, where α is a root of a primitive polynomial $f(x)$ of degree k. Under this representation, they define the *square orbit* of α to be the set $SO(\alpha) = \{\alpha^{2^i} | 0 \leq i \leq k - 1\}$, and say that if $SO(\alpha)$ contains many low degree polynomials for a particular choice of $f(x)$, then the resulting representation exhibits *orbital weakness*. Systematic equations may be generated in any representation of the field exhibiting such weakness.

It may seem at first that the use of systematic equations may be avoided and hence attacks slowed down by using "orbitally strong" representations of prime power order fields when implementing cryptosystems. This is not the case. In 1974, N. Zierler presented an algorithm which performs conversions between discrete logarithms in two different representations of the field $\mathbb{F}_{2^n}$ [80]. Whatever field representation a cryptosystem is implemented with, an attacker may use

Zierler's algorithm to solve the DLP in an orbitally weak representation of the same field, using systematic equations to reduce the workload, and then convert the result back to the implemented representation. Thus the improvement offered by systematic equations is always of relevance when considering the DLP in fields of characteristic 2.

5.4.2 Coppersmith's Work

These ideas were later extended by Coppersmith [20], who showed how *all* of the required relations to determine the factor base logarithms could be produced systematically.

Coppersmith's method for generating relations proceeds as follows. We begin by defining some constants. Our choices for these constants shall be justified later.

Suppose we are given a represention for our field using a primitive polynomial $p(x)$ of degree n such that $p(x) = x^n + q(x)$, where $\deg(q(x)) < n^{2/3}$. Coppersmith suggests that such a representation can be expected to exist, based on heuristic arguments. Once again, Zierler's algorithm can be used to convert between such a representation and any other one. Select a smoothness parameter b, such that $b \geq c_1 n^{1/3} \log(n)^{2/3}$ for some small constant c_1. Choose an integer d which is near b. Let k be the power of 2 which is nearest $\sqrt{n/d}$. Let h be $\lceil n/k \rceil$. Let the polynomial $r(x)$ be chosen so that $r(x) \equiv x^{hk} \pmod{p(x)}$. From our choice of $p(x)$, $r(x)$ now satisfies $r(x) = q(x)x^{hk-n}$. Set $r = \deg(r(x))$.

To generate a relation, we begin by select random polynomials $a(x), b(x) \in \mathbb{F}_2$ such that $\deg(a(x)), \deg(b(x)) \leq d$ and $\gcd(a(x), b(x)) = 1$. We then set $c(x) = x^h a(x) + b(x)$ and compute $d(x) \equiv c(x)^k$. If $c(x)$ and $d(x)$ are both b-smooth and factor as, say:

$$c(x) = \prod_{i=1}^{m} p_i(x)^{e_i} \quad \text{and} \quad d(x) = \prod_{i=1}^{m} p_i(x)^{f_i},$$

then we have the relation:

$$\sum_{i=1}^{m} (ke_i - f_i) \log_{g(x)}(p_i(x)) = 0.$$

This method has a high probability of successfully generating a relation, and is capable of generating enough relations to completely determine our factor base logarithms. To see why this is the case, we need to consider the structure of the polynomials $c(x)$ and $d(x)$, using the constants defined earlier.

Firstly, it should be clear that the maximum possible degree of $c(x) = x^h a(x) + b(x)$ is the maximum possible degree of $x^h a(x)$, which is $h + d$, via our choice of $a(x)$. Next we consider the maximum possible degree of $d(x)$. Recalling that k was chosen to be a power of 2, so exponentiation to the k-th power is a linear operation in $\mathbb{F}_2$, we observe that:

$$\begin{aligned} d(x) &= c(x)^k \\ &= \left(x^h a(x) + b(x)\right)^k \\ &= x^{hk} a(x)^k + b(x)^k \\ &= r(x) a(x)^k + b(x)^k, \end{aligned}$$

so the maximum possible degree of $d(x)$ is the maximum possible degree of $r(x)a(x)^k$, which is $r + kd$, by our choices of $r(x)$ and $a(x)$. By our choice of parameters, these maximum degrees are both approximately $\sqrt{nd}$. This is substantially less than $(n-1)/2$, so the probability that $c(x)$ and $d(x)$ are both smooth and we get a relation is even greater than the probability of obtaining a relation using the Euclidean algorithm as shown by the Waterloo team. The parameters were chosen above so as to optimise this probability. Further, Coppersmith shows that the number of relatively prime polynomials $a(x)$ and $b(x)$ available for use in this method is sufficient to obtain *all* the required relations this way. Using a similar analysis to ours from section 5.2 (i.e. using Odlyzko's estimate (5.4)for $P(n, b)$), Coppersmith finds the total time complexity for this algorithm to be:

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

which is substantially faster than the Waterloo algorithm or our original algorithm.

5.5 A Simple Index Calculus Method for Fields $\mathbb{F}_p$

It is clear that the simple method for fields $\mathbb{F}_{p^k}$ which was presented in Section 5.2 can be modified to apply to fields of prime power order $\mathbb{F}_p$. This view is, in fact, somewhat backward: the original description of that algorithm given by Hellman and Reyneri [32] was a generalisation of an earlier algorithm due to Adleman [1] which applied to prime order fields. We describe the basic changes between the two algorithms and, while we do not derive a complexity estimate, give Adleman's estimate and some references to relevant material.

Only one major change to the algorithm has to be made - the factor base. We still select a smoothness parameter b, although this time it is taken from the set $\{2, 3, \ldots, p-1\}$. Our factor base is then the set of all primes $p \leq b$. The concept of a *smooth integer* is defined in the expected manner - an integer is b-smooth if all of its prime divisors are $\leq b$. The rest of the algorithm proceeds in the same way.

In the case of $\mathbb{F}_p$, the size of the factor base is equal to the number of primes $\leq b$. The function which counts the number of primes below some given bound is usually denoted π, with $\pi(x)$ being the number of primes $\leq x$. This function is very well studied and a number of approximations and other results are known. For instance, it is true that:

$$\pi(x) < 1.25506\frac{x}{\log(x)},$$

for $x > 1$. Further reading on the prime counting function can be found in Ribenboim's book [63]. A number of algorithms for evaluating $\pi(x)$ can be found in a paper by and J. Lagarias, V. Miller and Odlyzko [38].

The number of field elements which can be factorised over the factor base is equal to the number of smooth integers $\leq b$. The function which counts the number of b-smooth integers below some given bound for a given b is usually denoted Ψ, where $\Psi(x, b)$ is the number of b-smooth integers $\leq x$. This function is also well studied, mainly in terms of its asymptotic behaviour. D. J. Bernstein

has published an algorithm for the fast computation of arbitrarily tight bounds on the function Ψ [6]. This paper includes the URL for software written by Bernstein to compute these bounds on a personal computer, and also gives many references to earlier work on the subject of Ψ, both computational and theoretical.

In the case of $\mathbb{F}_p$, the many factorisations involved in the algorithm are factorisations of integers, which we have not considered in this thesis. Unlike the problem of polynomial factorisation, the factorisation of integers is very difficult - so difficult that the infeasability of large factorisations is used to provide security in some cryptosystems, just like the DLP! The RSA algorithm [64] is such a cryptosystem, and was in fact the first public key cryptosystem developed. Because of this application to cryptography, the problem of integer factorisation has been studied extensively and a number of algorithms for the task exist. Unfortunately, space limitations prevent us from giving even a brief overview. The interested reader is given the following references to the three well-known and widely used modern algorithms:

1. The *Quadratic Sieve*, invented by C. Pomerance in 1984 [62].
2. The *Elliptic Curve Factorisation Method*, invented by H. W. Lenstra Jr. in 1987 [42].
3. The *Number Field Sieve*, invented by A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse and J. M. Pollard in 1990 [41].

These are by no means the only known factorisation algorithms more efficient than trial division, but they are the fastest for general factorisations and the most likely to be useful in an index calculus algorithm for large prime order fields. We note briefly that while the running time when using either of the sieve-based methods to factor an integer is independent of the size of that integer's factors, the elliptic curve method will factor an integer quicker if it has small factors, as is the case when the integer is smooth.

Adleman finds the complexity of this method to be:

$$\exp\left(c\sqrt{\log(p)\log(\log(p))}\right),$$

for a constant c.

5.6 An Improved Method for Some Fields $\mathbb{F}_p$

The method described above is by no means the only index calculus method for finite fields $\mathbb{F}_p$. We detail one more efficient alternative method here, and give reference to another.

The method we will now consider was presented by D. Coppersmith, A. Odlzyko and R. Schroepell in 1986 [21], which works for fields of prime order $p \equiv 1 \pmod 4$. This method was used by B. A. LaMacchia and Odlzyko in 1991 to readily compute logarithms in the field $\mathbb{F}_p$ for a certain 192-bit prime p which was actually used by Sun Microsystems in the security component of their Network File System (NFS). Their work is described in [40]. Our examination of this method is the one place in this thesis where we depart from the representation of $\mathbb{F}_p$ as the ring of integers modulo p. Here we use the representation of the

field as a factor ring of the Gaussian integers. This representation allows a more efficient method for generating relations than that described above. We begin by quickly presenting the relevant theory on Gaussian integers, omitting the proof of a well known and easily derived result.

Field Representation Using the Gaussian Integers

Definition 5.6.1. The Gaussian Integers

The ring of *Gaussian integers* is the integral domain:

$$\mathbb{Z}[i] = \{a + bi | a, b \in \mathbb{Z}\}$$

Definition 5.6.2. The Norm of a Gaussian Integer

The *norm* of a Gaussian integer $z = a+bi$, denoted $N(z)$, is the non-negative integer:

$$N(z) = a^2 + b^2$$

Lemma 5.6.3. Multiplicativity of the Norm

If $z_1, z_2 \in \mathbb{Z}[i]$, then:

$$N(z_1 z_2) = N(z_1)N(z_2)$$

Proof: Omitted.

The following theorem was first claimed by Fermat in 1640, although no proof of his has been found. The first known proof is due to Euler in 1749. The proof we present here is adapted from one given by Fraleigh [26]. This proof combines with the previous lemma to give a field representation theorem which is the basis for this index calculus method.

Theorem 5.6.4. Prime Numbers as the Sum of Two Squares

Let p be a prime number which satisfies $p \equiv 1 \pmod 4$. Then there exist natural numbers a and b such that $p = a^2 + b^2$.

Proof: We begin by considering $\mathbb{Z}_p$, the multiplicative group of units modulo p. This a cyclic group of order $p-1$. Since $p \equiv 1 \pmod 4$, $p-1$ is divisble by 4. Suppose $p-1 = 4k$ and let α be a generator of $\mathbb{Z}_p$. Then α^k is an element of $\mathbb{Z}_p$ with multiplicative order 4 and consequently $\alpha^2 \equiv -1 \pmod p$, i.e. $p|\alpha^2 + 1$.

Considered as an element of $\mathbb{Z}[i]$, $\alpha^2 + 1$ has the factorisation $(\alpha + i)(\alpha - i)$. Suppose that p is irreducible in $\mathbb{Z}[i]$. Then, by our above factorisation of $\alpha^2 + 1$, p must divide either $\alpha + i$ or $\alpha - i$. Suppose that p divides $\alpha + i$. Then $\alpha + i = p(a + bi)$ for some $a, b \in \mathbb{Z}$. Equating the coefficients of i in this equation, we get that $pb = 1$, which is impossible. Similarly, if p divided $\alpha - i$ then we could arrive at the equation $pb = -1$, which is also impossible. So p is not irreducible.

Since p is not irreducible, there must exist Gaussian integers $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ such that $p = z_1 z_2$. Taking the norm of each side of this equation and using Lemma 5.6.3 we see that $p^2 = (a_1^2 + b_1^2)(a_2^2 + b_2^2)$, an equation in $\mathbb{Z}$. Since p is a prime number, it must be the case that $(a_1^2 + b_1^2) = (a_2^2 + b_2^2)$, i.e.

$p = a^2 + b^2$ for some a, $b \in \mathbb{Z}$. Clearly we may assume a and b to be positive and also clearly neither of them may be 0, so $a, b \in \mathbb{N}$.

□

It is actually possible to show a stronger result than this: if a prime p can be expressed as a sum of two squares then it must satisfy $p \equiv 1 \pmod 4$, and the expression of a prime of this form as the sum of two squares is unique. These two properties will not be used here. The interested reader may find a proof of the first property in Fraleigh [26], and of the second part in a paper by Brillhart [12].

We now prove the main result of this section. This proof is adapted from [23].

Theorem 5.6.5. Field Representation Theorem

Let p be a prime number satisfying $p \equiv 1 \pmod 4$. Then there exists a Gaussian integer $z \in \mathbb{Z}[i]$ such that:

$$R = \frac{\mathbb{Z}[i]}{\langle z \rangle} \cong \mathbb{Z}_p,$$

and so R can be used as a representation for $\mathbb{F}_p$.

Proof: We begin by identifying the relevant Gaussian integer z. By Theorem 5.6.4, there exist $a, b \in \mathbb{N}$ such that $p = a^2 + b^2$. We may set $z = a + bi$ or $z = b + ai$.

Define the map $\phi : \mathbb{Z}[i] \to \mathbb{Z}_p$ according to the rule:

$$\phi : x + yi \mapsto x - ab^{-1}y \pmod p.$$

We show first that this map is a homomorphism.

Let $z_1 = x_1 + y_1 i, z_2 = x_2 + y_2 i$. Then:

$$\begin{aligned}
\phi(z_1) + \phi(z_2) &= x_1 - ab^{-1}y_1 + x_2 - ab^{-1}y_2 \pmod p \\
&\equiv (x_1 + x_2) - ab^{-1}(y_1 + y_2) \pmod p \\
&= \phi((x_1 + x_2) + (y_1 + y_2)i) \\
&= \phi(z_1 + z_2),
\end{aligned}$$

i.e. ϕ preserves the additive structure of $\mathbb{Z}[i]$. To show that it also preserves the multiplicative structure, we need to make the observation that $b^2 \equiv -a^2 \pmod{p = a^2 + b^2}$, so $(ab^{-1})^2 \equiv -a^2 a^{-2} \equiv -1 \pmod p$. With this in mind:

$$\begin{aligned}
\phi(z_1)\phi(z_2) &= \phi(x_1 + y_1 i)\phi(x_2 + y_2 i) \\
&= (x_1 - ab^{-1}y_1)(x_2 - ab^{-1}y_2) \pmod p \\
&\equiv x_1 x_2 - ab^{-1}(x_1 y_2 + x_2 y_1) + (ab^{-1})^2 y_1 y_2 \pmod p \\
&\equiv (x_1 x_2 - y_1 y_2) - ab^{-1}(x_1 y_2 + x_2 y_1) \pmod p \\
&= \phi((x_1 x_2 - y_1 y_2) + (x_1 y_2 + x_2 y_1)i) \\
&= \phi(z_1 z_2).
\end{aligned}$$

So ϕ also preserves the multiplicative structure of $\mathbb{Z}[i]$ and hence is a homomorphism. Now we show that $\ker(\phi) = \langle z \rangle$.

We begin by noting that $\phi(z) = a - ab^{-1}b = a - a = 0$, so, since ϕ is a homomorphism, any multiple of z maps to zero and hence $\langle z \rangle \subseteq \ker(\phi)$. We now show the reverse inclusion, and hence equality. Suppose that $x = c+di \in \ker(\phi)$. We consider x as an element of $\mathbb{C}$ rather than $\mathbb{Z}[i]$ and write the division:

$$\frac{x}{z} = \frac{c+di}{a+bi} = y = \frac{ac+bd}{a^2+b^2} + \frac{ad-bc}{a^2+b^2}i. \tag{5.11}$$

Since $x \in \ker \phi$, we have $\phi(x) = c - ab^{-1}d \equiv 0 \pmod{p}$, an equation in $\mathbb{Z}_p$. Multiplying this equation by b and using the commutativity of $\mathbb{Z}_p$ we get:

$$bc - ad \equiv 0 \pmod{p}. \tag{5.12}$$

Multiplying this by -1 and expanding p, $ad - bc \equiv 0 \pmod{a^2+b^2}$. This implies that the imaginary part of y, as expressed in 5.11, is an integer. Similarly, multiplying (5.12) by ab gives $ab^2c - a^2bd \equiv 0 \pmod{p}$, and then multiplying by b^{-2} gives $ac - (ab^{-1})^2bd \equiv ac + bd \equiv 0 \pmod{p}$, where we have used our earlier observation that $(ab^{-1})^2 \equiv -1 \pmod{p}$. This congruence implies that the real part of y is also an integer, so $y \in \mathbb{Z}[i]$. Consequently x is a Gaussian integer multiple of z, so $x \in \langle z \rangle$ and hence $\ker(\phi) \subseteq \langle z \rangle$. Together with the reverse inclusion shown earlier, this gives $\ker(\phi) = \langle z \rangle$.

Since ϕ is a homomorhism with $\ker(\phi) = \langle z \rangle$, it follows from the well known homomorphism theorem for rings that the factor ring R is isomorphic to the the image $\phi(\mathbb{Z}_p)$. This image is clearly $\mathbb{Z}_p$ itself, as for any $a \in \mathbb{Z}_p$, we have $a \in \mathbb{Z}[i]$ with $\phi(a) = a$. So $R \cong \mathbb{Z}_p$.

□

Thus we have a new representation of $\mathbb{F}_p$. We will see that an efficient index calculus algorithm for $\mathbb{F}_p$ can be constructed using this representation. If we are given a DLP in $\mathbb{F}_p$ posed using the integer representation of $\mathbb{F}_p$, wishing to find $\log_\alpha(\beta)$ as usual, we may compute $\phi(\alpha)$ and $\phi(\beta)$ and the find $\log_{\phi(\alpha)}(\phi(\beta))$ in the Gaussian integer representation using this index calculus algorithm. It is clear that these two logarithms will be the same. On a practical note, in order to perform computations in $\mathbb{F}_p$ using the Gaussian integer representation we need to be able to find the unique a and b such that $p = a^2 + b^2$. The simple proof we offered of the existence of this decomposition, howerver, was non-constructive. Constructive proofs exist, and are based on the concept of continued fractions. An efficient algorithm for finding, given a prime $p \equiv 1 \pmod{4}$, a and b satisfying $p = a^2 + b^2$ is given by Brillhart [12].

Choice of Factor Base

Supposing we have found a Gaussian integer $z = a + bi$ such that $\mathbb{Z}[i]/\langle z \rangle$ is a representation of our field of interest, then we define a factor base as follows. Select a smoothness parameter B and include in the factor base:

1. All irreducible Gaussian integers ("Gaussian primes") z_i with norm satisfying $N(z_i) \leq B$,
2. All prime numbers $p_i \leq B$,

3. The imaginary part of z, b.

The reason for this choice of factor base will become clear soon when we discuss our method for obtaining relations between the factor base elements.

Generating Relations

Observe that, for $c_1, c_2 \in \mathbb{Z}$, we have:

$$\begin{aligned} c_1 b - c_2 a &= b(c_1 + c_2 i) - c_2(a + bi) && (5.13)\\ &= b(c_1 + c_2 i) - c_2 z && (5.14)\\ &\equiv b(c_1 + c_2 i) \pmod{z}. && (5.15) \end{aligned}$$

If the integer $c_1 b - c_2 a$ is smooth with respect to the primes $p_1, p_2, \ldots, p_m$ of our factorbase and the Gaussian integer $b(c_1 + c_2)$ is smooth with respect to the Gaussian primes $z_1, z_2, \ldots, z_n$, then (5.15) provides a relation between factorbase elements. The relative sizes of a and b, as well as c_1 and c_2 will influence the probability of successfully obtaining a relation in this way. If c_1 and c_2 can be chosen so that $c_1 b - c_2 a$ is a "small" integer and $(c_1 + c_2 i)$ is a Gaussian integer of "small norm", then the likelihood of success will be increased. By using the above result with several values of c_1 and c_2, we can obtain a number of relations in a more systematic and efficient way than our earlier random exponentiation and factorisation.

5.7 Further Reading

Coppersmith's algorithm for fields of characteristic 2 is the fastest index calculus algorithm known for these fields. However, the algorithms we have shown for other fields can be improved upon. Unfortunately, the fastest algorithms rely upon the number field sieve algorithm for factoring integers (mentioned earlier in this chapter, see [41]) and their description is well beyond the level of theory developed in this thesis. The fastest algorithms are both due to D. Gordon. For fields of prime order, see [30]; this algorithm is a generalisation of the Gaussian integer method from Section 5.6 to different number fields. For fields of prime power order, see [29].

We also mention an earlier index calculus algorithm due to Elgamal [25] which works for finite fields $\mathbb{F}_{p^2}$. This algorithm is slower than Gordon's later algorithm, but for large p it is faster than the Waterloo algorithm, making it the fastest algorithm available for these fields at the time of its publication.

A paper by T. Garefalakis and Panario [27] considers the possibility of using a factor base in index calculus algorithms for fields $\mathbb{F}_{p^n}$ other than the usual choice of all irreducible polynomials of degree below some bound. In particular, the analyse an index calculus algorithm where the factor base is all irreucible polynomials of degree between a lower and upper bound. They show that such an algorithm is asymptotically as fast as that using the standard factor base, however in practice there is little benefit in the change. The generalisations of the Waterloo variant and Coppersmith's method to this new factor base are also considered.

Finally, we mention that index calculus algorithms are not limited to finite fields, although this is certainly the situation in which they have been the most

well studied and often used. In a paper by R. Granger and F. Vercauteren [31] we see the development of an index calculus algorithm for solving the DLP on an *algebraic torus*. Algebraic tori are a certain kind of subgroup of $\mathbb{F}_q^*$ which have lately attracted attention as possible settings for efficient DLP based cryptography. In the references of [31], we see index calculus algorithms for abelian varieties and for hyperelliptic curves.

Appendix A

Computer Code

This chapter contains listings of computer code implementations of many of the algorithms discussed in this thesis.

A.1 The GNU Multiple Precision Library

The GNU Multiple Precision Library (GMP library) is "a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface". More information can be found at: `http://www.swox.com/gmp/` (the source of the preceeding quotation).

Multiplication of integers is performed in GMP via multiplying polynomials, as discussed in 2.2.1. The Karatsuba multiplication algorithm discussed in 2.2.3 and the Fast Fourier Transform multiplication algorithm discussed in 2.2.4 are used for this, depending upon which is fastest for the particular polynomials involved. A fast algorithm is available for modular exponentiation, as is a fast implementation of the Extended Euclidean Algorithm.

The integer arithmetic functions of the GMP library have be used to implement some of the algorithms discussed in this thesis for finite fields of prime order.

A.2 Pollard's ρ-method

The following program, written in C, provides an implementation of Pollard's ρ-method for computing discrete logarithms in finite fields of prime order. The GMP library provides fast integer arithmetic. The function `pollardrho` accepts as arguments a pointer to an `mpz_t` type variable and 3 `mpz_t` type variables - a group element h, a group generator g and a group order p. The discrete logarithm of h to the base g is computed and stored in the variable pointed to by the first argument. This function repeatedly calls `iterate` to update the sequence variables. The `main` program in this code listing simply accepts commandline inputs for g, h and p, calls `pollardrho` and prints the result.

```
#include <stdio.h>
#include <time.h>
#include <gmp.h>

/* Function prototypes */

int pollardrho(mpz_t *result, mpz_t h, mpz_t g, mpz_t p);
void iterate(mpz_t *x, mpz_t *a, mpz_t *b);

/* Global variables */

mpz_t h, g, p, pminus1;
gmp_randstate_t state;

/****************
 * Main program *
 ****************/

int main(int argc, char *argv[]) {

/* Initialise and set global variables */

mpz_init_set_str(h,argv[1],10);
mpz_init_set_str(g,argv[2],10);
mpz_init_set_str(p,argv[3],10);
mpz_init(pminus1);
mpz_sub_ui(pminus1,p,1);

/* Prepare random number generator and seed by time */

unsigned long int timeseed;
time(&timeseed);
gmp_randinit_default(state);
gmp_randseed_ui(state,timeseed);

/* Compute, store and display logarithm */

mpz_t result;
mpz_init(result);
pollardrho(&result,h,g,p);
gmp_printf("Logarithm of %Zd to base %Zd in Z*(%Zd) is %Zd.\n", \
        h, g, p, result); */

}

/*********************************
 * Pollard's rho-method function *
 *********************************/

int pollardrho(mpz_t *result, mpz_t h, mpz_t g, mpz_t p) {
```

```
/* Randomly set initial sequence terms */

mpz_t a1, b1, x1, a2, b2, x2, temp;
mpz_init(a1); mpz_init(b1), mpz_init(x1); mpz_init(temp);
mpz_urandomm(a1,state,p); mpz_urandomm(b1,state,p);
mpz_powm(x1,g,a1,p); mpz_powm(temp,h,b1,p);
mpz_mul(x1,x1,temp); mpz_mod(x1,x1,p); mpz_clear(temp);
mpz_init(a2); mpz_init(b2); mpz_init(x2);
mpz_set(a2,a1); mpz_set(b2,b1); mpz_set(x2,x1);

/* Use Floyd's algorithm to find sequence colliison */

do {
        iterate(&x1,&a1,&b1);
        iterate(&x2,&a2,&b2);
        iterate(&x2,&a2,&b2);
} while (mpz_cmp(x1,x2)!=0);

/* Construct linear congruence */

mpz_t a, b; /* a(log) = b (mod p-1) */
mpz_init(a); mpz_init(b);
mpz_sub(b,a2,a1); mpz_mod(b,b,pminus1);
mpz_sub(a,b1,b2); mpz_mod(a,a,pminus1);
mpz_clear(a1); mpz_clear(a2); mpz_clear(b1); mpz_clear(b2);

/* Ensure congruence is non-trivial and retry if not */

if(mpz_cmp_ui(a,0)==0) {
 pollardrho(result,h,g,p);
}

/* Solve linear congruence if possible and retry otherwise */

mpz_t gcd, soln;
mpz_init(gcd); mpz_init(soln);
mpz_gcdext(gcd,soln,NULL,a,pminus1);
if(mpz_divisible_p(b,gcd)) {
        /* Congruence has a solution, solve it */
        mpz_mul(soln,b,soln);
        mpz_div(soln,soln,gcd);
        mpz_mod(soln,soln,pminus1);

} else {
        /* Congruence has no solution, try again */
        pollardrho(result,h,g,p);
}

/* Perform trial exponentiation on candidates */
```

```
        mpz_t step, trial;
        mpz_init(step); mpz_init(trial);
        mpz_div(step,pminus1,gcd);
        mpz_mod(soln,soln,step);

        while(1) {
                mpz_powm(trial,g,soln,p);
                if(mpz_cmp(trial,h)==0) break;
                else mpz_add(soln,soln,step);
        }

        /* Store result */

        mpz_set(*result,soln);

}

/***********************************
 * Rho sequence updating subroutine *
 ***********************************/

void iterate(mpz_t *x, mpz_t *a, mpz_t *b) {

        if(mpz_congruent_ui_p(*x,0,3)) {
                mpz_mul(*x,*x,*x);
                mpz_mod(*x,*x,p);
                mpz_mul_ui(*a,*a,2);
                mpz_mod(*a,*a,pminus1);
                mpz_mul_ui(*b,*b,2);
                mpz_mod(*b,*b,pminus1);
        } else if(mpz_congruent_ui_p(*x,1,3)) {
                mpz_mul(*x,g,*x);
                mpz_mod(*x,*x,p);
                mpz_add_ui(*a,*a,1);
                mpz_mod(*a,*a,pminus1);
        } else if(mpz_congruent_ui_p(*x,2,3)) {
                mpz_mul(*x,h,*x);
                mpz_mod(*x,*x,p);
                mpz_add_ui(*b,*b,1);
                mpz_mod(*b,*b,pminus1);
        }

}
```

Bibliography

[1] L. Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*, Proceedings of the IEEE 20th Annual Symposium on Foundations of Computer Science (1979), 55–60.

[2] Iris Anshel, Michael Anshel, and Goldfeld Dorian, *An algebraic method for public-key cryptography*, Mathematical Research Letters **6** (1999), 287–291.

[3] Elwyn R. Berlekamp, *Factoring polynomials over finite fields*, Bell Systems Technical Journal **46** (1967), 1853–1859.

[4] ———, *Algebraic coding theory*, McGraw-Hill, 1968.

[5] ———, *Factoring polynomials over large finite fields*, Mathematics of Computation **24** (1970), 713–735.

[6] Daniel J. Bernstein, *Arbitrarily tight bounds on the distribution of smooth integers*, `http://cr.yp.to/papers/psi.pdf` (2000).

[7] ———, *Multidigit multiplication for mathematicians*, `http://cr.yp.to/papers/m3.pdf` (2001).

[8] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, *Computing logaithms in finite fields of characteristic two*, SIAM Journal of Algorithmic and Discrete Mathematics **5** (1984), 276–285.

[9] Ian F. Blake, XuHong Gao, Ronald C. Mullin, Scott A. Vanstone, and Tomik Yaghoobian, *Applications of finite fields*, Kluwer Academic Publishers, 1993.

[10] J.V. Brawle and Carlitz L., *Irreducibles and the composed product for polynomials over a finite field*, Discrete Mathematics **65** (1987), 115–139.

[11] Richard P. Brent, *An improved monte carlo factorization algorithm*, BIT **20** (1980), 176–184.

[12] John Brillhart, *Note on representing a prime as a sum of two squares*, Mathematics of Computation **26** (1972), 1011–1013.

[13] M. C. R. Butler, *On the reducibility of polynomials over a finite field*, Quarterly Journal of Mathematics, Oxford Series 2 **5** (1954), 102–7.

[14] David G. Cantor and Erich Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Informatica **28** (1991), 693–701.

[15] David G. Cantor and Hans Zassenhaus, *A new algorithm for factoring polynomials over finite fields*, Mathematics of Computation **36** (1981), 587–592.

[16] J. Cheon, J. Han, J. Kang, K. Ko, S. Lee, and C. Park, *New public-key cryptosystem using braid groups*, Advances in Cryptology - CRYPTO 2000.

[17] S. A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, Cambridge, MA: Harvard University, 1966.

[18] J.W. Cooley and J.W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of Computation **19** (1965), 297–301.

[19] D. Coppersmith and S. Winograd, *On the asymptotic complexity of matrix multiplication*, SIAM Journal of Computing **11** (1982), 472–492.

[20] Don Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Transactions on Information Theory **30** (1984), 587–594.

[21] Don Coppersmith, Andrew M. Odlzyko, and Richard Schroeppel, *Discrete logarithms in* GF(p), Algorithmica **1** (1986), 1–15.

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

[23] Greg Dresden and Dymàček, *Finding factors of factor rings over the gaussian integers*, American Mathematical Monthly **112** (2005), 602–611.

[24] Taher Elgamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **IT–31** (1985), 469–472.

[25] ———, *A subexponential-time algorith for computing discrete logarithms over* GF(p^2)., IEEE Transactions on Information Theory **31** (1985), 473–481.

[26] Jonh B. Fraleigh, *A first course in abstract algebra*, Addison Wesley Publishing Company, 2003.

[27] Theodoulos Garefalakis and Daniel Panario, *The index calculus algorithm using non-smooth polynomials*, Mathematics of Computation **70** (2001), 1253–1264.

[28] O. Goldreich, S. Goldwasser, and S. Halevi, *Public key cryptosystems from lattice reduction problems*, Advances in Cryptology - CRYPTO '97 **1294** (1997), 112–131.

[29] D. Gordon, *Discrete logarithms in* GF(p^n) *using the number field sieve*, (1991).

[30] ———, *Discrete logarithms in* GF(p) *using the number field sieve*, SIAM Journal of Discrete Mathematics **6** (1993), 124–138.

[31] R. Granger and F. Vercauteren, *On the discrete logarithm problem on algebraic tori*, Proceedings of Crypto 2005 **3621**.

[32] Martin E. Hellman and J. M. Reyneri, *Fast computation of discrete logarithms in GF(q)*, Advances in Cryptology - Proceedings of CRYPTO '82 (1983), 3–13.

[33] Erich Kaltofen and Victor Shoup, *Subquadratic-time factoring of polynomials over finite fields*, Mathematics of Computation **67** (1998), 1179–1197.

[34] A. Karatsuba, Doklady Akademia Nauk SSSR **145** (1962), 293–294.

[35] Donald E. Knuth, *The art of computer programming - volume 3 / sorting and searching*, Addison Wesley Publishing Company, 1973.

[36] Neal Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), 203–209.

[37] Lydia Kronsjö, *Computational complexity of sequential and parallel algorithms*, John Wiley and Sons, 1985.

[38] J. C. Lagarias, V. S. Miller, and A. M. Odlyzko, *Computing $\pi(x)$: The meissel-lehmer method*, Mathematics of Computation **44** (1985), 537–560.

[39] B. A LaMacchia and A. M. Odlyzko, *Solving large sparse linear systems over finite fields*, Advances in Cryptology - Proceedings of CRYPTO '90 (1990), 109–133.

[40] B. A. LaMachia and A. M. Odlyzko, *Computation of discrete logarithms in prime fields*, Designs, Codes and Cryptography **1** (1991), 47–62.

[41] A. K. Lenstra, H. W. Jr. Lenstra, M. S. Manasse, and J. M. Pollard, *The number field sieve*, Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing (1990), 564–572.

[42] H. W. Jr. Lenstra, *Factoring integers with elliptic curves*, Annals of Mathematics (1987), 649–673.

[43] Jr. Lenstra, H. W and C. P. Schnorr, Mathematics of Computation **43** (1984), 289–311.

[44] Rudolph Lidl and Harald Niederreiter, *Finite fields*, Cambridge University Press, 1997.

[45] Wenbo Mao, *Modern cryptography: Theory and practice*, Prentice Hall PTR, 2004.

[46] R. J. McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report **42–44** (1978), 114–116.

[47] Alfred Menezes, Paul C. Van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, Inc., 1997.

[48] Maurice Mignotte and Doru Ştefănescu, *Polynomials; an algorithmic approach*, Spring Verlag, 1999.

[49] J.C.P. Miller and Western A. E., *Tables of indices and primitive roots*, Royal Society Mathematical Tables **9** (1968).

[50] R. T. Moenck, *Fast computation of GCDs*, Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (1973), 142–151.

[51] Robert T. Moenck, *Practical fast polynomial multiplication*, Proceedings of the ACM Symposium on Symbolic and Algebraic Computation (1976).

[52] Peter Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44**, 519.

[53] Harald Niederreiter, *A new efficient factorization algorithm for polynomials over small finite fields*, Appl. Alg. Eng. Comm. Comp. (1993), 81–87.

[54] A. Odlyzko, *Discrete logarithms and their cryptographic significance*, Proceedings of Eurocrypt 1984 **209** (1985), 224–314.

[55] U.S. Department of Commernce / National Institute of Standards and Technolgoy, *Digital Signature Standard (DSS)*, `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf`.

[56] Daniel Panario, *What do random polynomials over finite fields look like?*, Proceedings of the Seventh International Conference on Finite Fields: Theory, Applications, and Algorithms (2004), 89–108.

[57] Stephen C. Pohlig and Martin E. Hellman, *An improved algorithm for computing logarithms over* GF(p) *and its cryptographic significance*, IEEE Transactions on Information Theory **IT-24** (1978), 106–110.

[58] J. M. Pollard, *The fast fourier transform in a finite field*, Mathematics of Computation **25** (1971), 265–74.

[59] ———, *A monte carlo method for factorization*, BIT **15** (1975), 331–334.

[60] ———, *Monte carlo methods for index computation* (mod p)., Mathmatics of Computation **32** (1978), 918–924.

[61] ———, *Kangaroos, monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447.

[62] Carl Pomerance, *The quadratic sieve factoring algorithm*, Proceedings of Eurocrypt 1984 (1984), 169–182.

[63] Paulo Ribenboim, *The new book of prime number records*, Springer, 2004.

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

[65] Bruce Schneier, *Applied cryptography*, John Wiley and Sons, Inc., 1996.

[66] Claus P. Schnorr, *Efficient signature generation by smart cards*, Journal of Cryptology **4** (1991), 161–174.

[67] A. Schönhage, *Schnelle multiplikation von polynomen über körpen der charakteristik 2*, Acta Informatica **7** (1977), 395–398.

[68] A. Schönhage and V. Strassen, *Schnelle multiplikation großer zahlen*, Computing **7** (1971), 281–292.

[69] Victor Shoup, *On the deterministic complexity of factoring polynomials over finite fields*, Information Processing Letters **33** (1990), 261–267.

[70] ———, *Fast construction of irreducible polynomials over finite fields*, Journal of Symbolic Computation **17** (1993), 371–391.

[71] ———, *Lower bonds for discrete logarithms and related problems*, Proceedings of Eurocrypt 1997 (1997).

[72] Igor E. Shparlinski, *Finite fields: Theory and computation*, Kluwer Academic Publishers, 1999.

[73] V. Strassen, *Gassian elimination is nhot optimal*, Numerical Mathematics **13** (1969), 354–356.

[74] Edlyn Teske, *On random walks for pollard's rho method*, Mathematics of Computation **70** (2001), 809–825.

[75] A. L. Toom, *The complexity of a scheme of functional elements simulating the multiplication of integers*, Doklady Academic Nauk SSSR **150** (1963), 496–498.

[76] P. V. Trifonov and S. V. Fedorenko, *A method for fast computation of the fourier transform over a finite field*, Problems of Information Transmission **39** (2003).

[77] Paul C. van Oorschot and Michael J. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28.

[78] Joachim von zur Gathen and Victor Shoup, *Computing frobenius maps and factoring polynomials*, Computational Complexity **2** (1992), 97–105.

[79] D. H. Widemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **32** (1986), 54–62.

[80] N. Zierler, *A conversion algorithm for logarithms on* $\mathrm{GF}(2^n)$, Journal of Pure and Applied Algebra **4** (1974), 353–356.