

On the Difficulty of Prime Root Computation in Certain Finite Cyclic Groups

Anna M. Johnston

Technical Report
RHUL-MA-2006-1
27 March 2006



Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
<http://www.rhul.ac.uk/mathematics/techreports>

**ON THE DIFFICULTY OF PRIME ROOT
COMPUTATION IN CERTAIN FINITE
CYCLIC GROUPS**

Anna M. Johnston

Royal Holloway and Bedford New College,
University of London

*Thesis submitted to
The University of London
for the degree of
Doctor of Philosophy
2006.*

The work contained in this thesis
was performed by
Anna M. Johnston

Acknowledgments

This work would not have been possible without the support of many mentors, both in the United States and in the United Kingdom. On the top of the list is my advisor Sean and my family: Mark, Ben and Sam. A close second is my management and colleagues at Sandia National Laboratories¹: Bruce Hendrickson, Cindy Phillips, David Womble, Mel Loran, Suzanne Rountree, and Bill Camp. Finally, many thanks to my fellow students Roger, Laurence, Thomas, Colin, Maura, Illana and Su-Jeong for keeping me sane.

¹Sandia National Laboratories, Albuquerque, New Mexico 87185-1110. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Abstract

Public key cryptography is a relatively new branch of cryptography, with the first cryptosystems published in the 1970's. Despite its youth, much of our modern information infrastructure relies on public key cryptography for efficient security.

Public key cryptosystems are built on computationally infeasible mathematical problems. The two most common problems are factoring large composite integers and computing discrete logarithms. It is well known that the problems of factoring a composite integer N and computing a square root modulo N are equivalent, with several cryptosystems designed from this hard problem. A lesser known equivalency exists between the discrete logarithm and q^{th} root problems. Public key cryptosystems have been based on the difficulty of the q^{th} root problem, including a key exchange provably secure against the man-in-the-middle attack.

If q is a prime integer and G is a finite cyclic group such that q^2 divides the order of G , then computing a q^{th} root in G appears to be equivalent to computing a discrete logarithm of an element of order q . No known q^{th} root algorithms for groups G exist which do not require discrete logarithm computation. Earlier work proved that if a 'well-behaved' q^{th} root algorithm existed, then discrete logarithms of elements of order q in these groups could be computed with $2 \log_2(q)$ calls to the q^{th} root algorithm.

All known q^{th} root algorithms for G can be reduced to a single algorithm, described in the thesis. It requires a discrete logarithm computation whenever q^2 divides the order of the group. However, one specialized algorithm is known which does not require a discrete logarithm computation: Cipolla's algorithm. This algorithm only works if G is the multiplicative group of a finite field. If

Cipolla's algorithm were more efficient than discrete logarithm computation this would show that with current technology the q^{th} root problem is easier than the discrete logarithm problem in finite fields. If it was 'well-behaved' then it would give an efficient discrete logarithm algorithm when combined with earlier work. However in its current form Cipolla's algorithm is computationally more expensive than finding a discrete logarithm using exhaustion and does not have the 'well-behaved' property needed to find discrete logarithms.

Cipolla's algorithm was originally designed to compute square roots when large powers of 2 divided the order of the group and has not been analyzed for larger primes. If the algorithm could be modified such that its computational requirements were less than a discrete logarithm or it were given the 'well-behaved' property needed to compute a discrete logarithm, then the balance between the two problems would be upset, at least in the multiplicative groups of finite fields.

We will show that Cipolla's algorithm does not upset the balance of the q^{th} root and discrete logarithm problems. If the computational requirements could be minimized with respect to q , then this implies pre-knowledge of the q^{th} root. While Cipolla's algorithm itself gives no further insight into the q^{th} root or discrete logarithm problems, the research indicates new directions for future work on these problems.

Contents

Acknowledgments	3
Abstract	4
Contents	6
List of Tables	9
List of Figures	10
Notation	11
1 Introduction	12
1.1 The q^{th} root and discrete logarithm problems	15
1.2 An unexplored algorithm	16
2 Number theory for finding roots and Cipolla's algorithm	19
2.1 Number theoretic structure of $(P^q - 1)$	19
2.2 The discrete logarithm function	23
2.3 The Chinese remainder theorem in finite cyclic groups	24
2.4 Residues and residuosity testing	28
2.4.1 Residuosity and discrete logarithms	30
2.5 q^{th} roots of unity	32
2.6 Conjugates, Norms, and Traces	33
2.7 Representation of \mathbb{F}_{P^q}	37
3 Discrete logarithm techniques	40
3.1 Discrete logarithm algorithms with square root run-time	42
3.1.1 Baby-step-giant-step	42

3.1.2	Pollard's randomized techniques	45
3.2	Index Calculus	49
4	Root finding techniques	53
4.1	Analysis of a square root algorithm	54
4.2	Root finding algorithms based on the order of the group	56
4.3	Cipolla's algorithm	59
4.4	Comparison of required multiplies for Cipolla's and standard q^{th} root algorithms	62
5	On the difficulty of the q^{th} root problem	67
5.1	The q^{th} root oracles	68
5.2	Using the well-behaved q^{th} root oracle to compute a discrete log- arithm	70
5.3	An example	74
6	Polynomials and elements for Cipolla's algorithm	77
6.1	Root generating elements and root mapping polynomials	78
6.2	Enumeration of Polynomials based on the norm of its roots	80
7	Breaking down Cipolla's Algorithm	82
7.1	Exponential structure of Cipolla's root finding algorithm	83
7.1.1	Formulas for base field elements and conjugates	86
7.1.2	An alternate enumeration	87
7.2	Similar root mapping polynomials	88
7.3	Representation and order of a root mapping polynomial equiva- lence class	91
7.4	Knowledge of set representative implies knowledge of q^{th} roots	96
8	Reducing computational requirements	98
8.1	Cipolla's algorithm for small order equivalence class	99

8.2	Prime order equivalence classes	100
8.3	Trace formulas for minimal order representative polynomials	103
8.4	Minimal order representative polynomials	105
8.4.1	Distributions in \mathbb{Z}_R	108
8.4.2	A partitioning of \mathcal{A}_i	109
8.5	Minimal representative polynomials for $q = 5$ and $q = 11$	118
9	Fourier transforms on root mapping polynomial and elements in \mathbb{F}_{P^q}	120
9.1	Simplified conjugate formula	122
9.2	Fourier transforms over \mathbb{F}_P on root mapping polynomials and elements in \mathbb{F}_{P^q}	123
9.3	Properties of Fourier transformed root mapping polynomials and their roots	126
9.4	Multiplication in \mathbb{F}_{P^q} using transformed elements	127
10	Conclusions	133
A	Tables	137
A.1	Examples of prime (q, P) pairs for which $q^2 \mid (P - 1)$	137
A.2	Examples of fields with minimal order root mapping polynomial equivalence class	137
B	Other Algorithms	139
B.1	Degree q polynomial irreducibility testing in \mathbb{F}_P	139
	Bibliography	142

List of Tables

2.1	Parameters for \mathbb{F}_{P^q}	20
4.1	Smallest n for a given q such that Cipolla's algorithm is cost effective assuming: we have an appropriate polynomial and that multiplications in $\mathbb{F}_P[x]/(f(x))$ are equivalent to multiplications in \mathbb{F}_P	65
4.2	Smallest n for a given q such that Cipolla's algorithm is cost effective.	65
7.1	Variables used for analysis of Cipolla's algorithm	84
8.1	Minimal order root mapping polynomial equivalence class restrictions	103
A.1	Small test primes with $q^2 P - 1$	138
A.2	Primes for which minimal order equivalence classes exist	138

List of Figures

2.1	$\hat{\Psi}$ function diagram	26
3.1	The Discrete Logarithm Problem	41
3.2	Pollard Rho Example	47

Notation

\mathbb{N}	the set of natural numbers $\{1, 2, 3, \dots\}$;
\mathbb{Z}	The ring of integers under standard addition and multiplication;
\mathbb{Z}_P	the ring of integers modulo P ;
$\mathbb{F}_P, \mathbb{F}_P^*$	the finite field of order P and its multiplicative subgroup;
$\langle a \rangle$	the cyclic group generated by a ;
$dl_g(w)$	the index or discrete logarithm of w base g : $g^{dl_g(w)} \equiv w$;
$N(\gamma), N_{\mathbb{F}_{P^q}}(f)$	the norm of $\gamma \in \mathbb{F}_{P^q}$ or of $f(x) \in \mathbb{F}_P[x]$ with respect to \mathbb{F}_{P^q} ;
$ord(G), ord(g)$	the order of the group G or the order of the element $g \in G$;
G, G_s	a finite cyclic group and one with given order s ;
$a b$	a divides b ;
$q^n \parallel b$	for prime q and integer $n > 0$, $q^n b$ and $q^{n+1} \nmid b$;
$lg(P)$	$lg(P) = \log_2(P)$;
$\mathfrak{C}(f), \mathfrak{C}(\eta)$	the q^{th} root returned using Cipolla's algorithm and irreducible degree q polynomial $f \in \mathbb{F}_P[x]$ or its root $\eta \in \mathbb{F}_{P^q}$;
$\mathcal{F}(A) = [\mathcal{F}_j(A)]_{j=0}^{n-1}$	the n dimension discrete Fourier transform on $A \in \mathbb{F}_P^q$ or $A(x) \in \mathbb{F}_P[x]$;

Chapter 1

Introduction

Public key cryptography is a relatively new branch of cryptography, with the first cryptosystems published in the 1970's. Despite its youth, much of our modern information infrastructure relies on public key cryptography for efficient security.

Public key cryptosystems are built on computationally infeasible mathematical problems. The two most common problems are factoring large composite integers and computing discrete logarithms. It is well known that the problems of factoring a composite integer N and computing a square root modulo N are equivalent, with several cryptosystems designed from this hard problem, as in [36] and [14]. A lesser known problem equivalency exists between the discrete logarithm and q^{th} root problems. Public key cryptosystems have been designed off of the q^{th} root problem, including a key exchange provably secure against the man-in-the-middle attack. This key exchange is known as the Secure Authenticated Key Exchange (SAKE) and is briefly described below. See [22] for a more detailed description.

SAKE Components

q : A large prime integer;

G_{q^2} : A group (or subgroup) for which discrete logarithms are 'hard' and which has order q^2 ;

u : $u \in G_{q^2}$, whose order is q^2 ;

x_i : Long term secret integer (modulo q^2) for user i ;

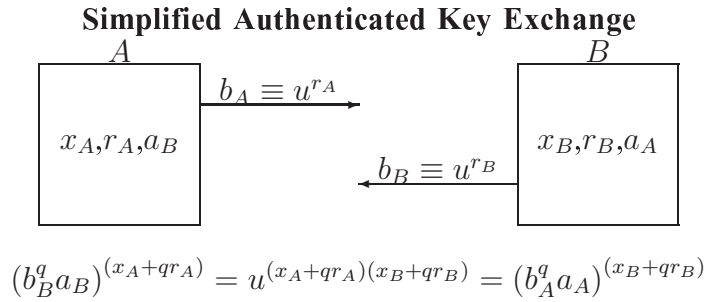
a_i : The public version of x_i , namely $a_i \equiv u^{x_i}$;

r_i : A one-time random secret integer (modulo q^2) for user i ;

b_i : The public version of r_i , namely $b_i \equiv u^{r_i}$;

Simplified Authenticated Key Exchange (SAKE)

1. Transmit the b_i values;
2. Receive b_j ;
3. Compute: $r_j \equiv b_j^q a_j \equiv u^{x_j + qr_j}$;
4. Compute the shared key: $c_j^{x_i + qr_i} \equiv u^{(x_i + qr_i)(x_j + r_j q)}$;



The q^{th} root problem is a very newly discovered computationally infeasible problem. It was first suggested for use in public key cryptosystems in 1999 in [4]. Like the square root problem, it is concerned with the difficulty of root computation. Unlike the square root problem, its difficulty is linked not to factoring but the discrete logarithm problem. Traits of the cryptosystem arise from the problem on which it is based. Factoring based algorithms are generally simpler to implement

and have straight forward encryption/decryption processes, as in RSA [37]. Discrete logarithm based algorithms have different qualities: simplified infrastructure, increased flexibility in choice of underlying group, and often improved confidence in its security.

In large systems with many users, discrete logarithm based systems can simplify key generation and updating. Factoring based algorithms generally require each user to have their own secret moduli, which determines the ring over which to operate. In a large system this requires the secure generation and distribution of many two prime composite integers. In discrete logarithm based systems, all users can operate over the same group. Furthermore, if a public/private key pair is compromised in a factoring based system, the factorization is also compromised and the ring must be replaced. If the cryptosystem is one of the few which allows for a shared modulus, a single compromise would require the re-keying of the entire system. In a discrete logarithm based system only the public/private key pair for the individual would need to be replaced – the group structure could remain.

Both factoring and discrete logarithm based cryptosystems are computationally expensive compared to their symmetric counterparts. The cost of discrete logarithm (and therefore q^{th} root) based systems can be greatly reduced by choosing an appropriate group. Elliptic curve cryptosystems are based on the discrete logarithm problem over a group of points on an elliptic curve [24], greatly reducing the required key size and often the computation.

Both factoring and discrete logarithm based algorithms have attacks based on the order of the group. An ill chosen group with poor order, such as those vulnerable to the attack in [33], severely weakens the security of the algorithm. Public knowledge of the group's order, as in most discrete logarithm based systems,

can reassure both secret key holders and public users of the systems security.

The q^{th} root problem opened new public key design possibilities while retaining the benefits of discrete logarithm based systems. Algorithms based on the q^{th} root problem work over the same types of finite cyclic groups that discrete logarithm based algorithms use.

1.1 The q^{th} root and discrete logarithm problems

The q^{th} root problem is closely connected to the discrete logarithm problem. In chapter 5 we prove that, in certain circumstances, the two problems are computationally equivalent. In other words, the ability to take discrete logarithms implies the ability to solve the q^{th} root problem and the ability to solve the q^{th} root problem implies the ability to compute discrete logarithms. This section defines the two problems.

The discrete logarithm problem is one of the foundational problems of public key cryptography and is thoroughly described in most books on cryptography such as [43] and [31], and a formal definition of the discrete logarithm function is in section 2.2. For elements $g, a \in G$ where G is a finite cyclic group, the notation $dl_g(a)$ will be used to denote the discrete logarithm of a to the base g : $g^{dl_g(a)} = a$.

The q^{th} root problem is not as well known as the discrete logarithm problem. The first reference to the q^{th} root problem as a basis for public key cryptographic algorithms is in [4].

Definition 1.1.1 (q^{th} roots and the q^{th} root problem): *Let q be a prime integer and G be a finite cyclic group such that $q^2 \mid \text{ord}(G)$. If $w, a \in G$ such that $a^q = w$ then a is a q^{th} root of w . The q^{th} root problem is to find a q^{th} root for a given $w \in G$.*

Let a q^{th} root exist for $w \in G$ where $w \neq 1$. In chapter 5 we show that $(w)^{\frac{1}{q}} \in \langle w \rangle$ exists only if $q^2 \nmid ord(G)$. If $q^2 \nmid ord(G)$ a q^{th} root can be found with a simple exponentiation. However, if $q^2 \mid ord(G)$, then $(w)^{\frac{1}{q}} \notin \langle w \rangle$ and a q^{th} root can not be found with an exponentiation. The most efficient q^{th} root technique currently known requires computation of discrete logarithms of elements with order q . This is described in more detail in chapter 4.

Computing q^{th} roots for the special case of $q = 2$ (i.e., square roots) has recieved great deal of attention. The most efficient square root algorithms over finite cyclic groups are essentially equivalent¹. Algorithm 4.1.1 generalizes these square root techniques, explaining the major steps and why they are needed. No matter how the algorithm is described, all require discrete logarithm computations in a subgroup of order 2 when (2^2) divides the order of the group. This fact is often disguised behind quadratic reciprocity testing and the simplicity of finding the discrete logarithms of elements of order two.

Similarly, there is really only one known q^{th} root algorithm on finite cyclic groups, generalized in algorithm 4.2.1. Substituting $q = 2$ into this algorithm produces algorithm 4.1.1. For $q > 2$ the discrete logarithm computation can not be disguised.

1.2 An unexplored algorithm

An equivalency proof of the q^{th} root and discrete logarithm problems was given in [4] and is also shown in chapter 5. It is easy to show that the ability to compute discrete logarithms implies the ability to find q^{th} roots. The difficult part is proving that the ability to compute q^{th} roots implies the ability to compute

¹Other algorithms for computing square roots exist when working over a finite field, such as [40]

discrete logarithms. In the equivalency proof this was done by first assuming the ability to compute q^{th} roots, then using this ability to compute discrete logarithms. This ability is defined by a q^{th} root oracle: an imaginary machine which computes q^{th} roots.

Unlike the discrete logarithm problem, the q^{th} root problem does not generate a well defined function. If a q^{th} root exists for $w \in G$, then there are q solutions to $(w)^{\frac{1}{q}}$. So the proof made assumptions about the type of q^{th} roots the oracle returned. The assumptions were based on general purpose q^{th} root algorithms currently known which do not require discrete logarithm computations, as in [2], [21], and chapter 4.

Although the efficient q^{th} root algorithms over arbitrary finite cyclic groups require discrete logarithm computations, one special purpose algorithm exists which does not: Cipolla's algorithm [7] and [2]. This algorithm finds q^{th} roots in the multiplicative group of a finite field, \mathbb{F}^* , by working in its extension field \mathbb{F} of degree q . It was originally designed to compute square roots ($q = 2$). The general purpose form of the algorithm computes q^{th} roots for any prime q dividing the order of the multiplicative group and is described in section 4.3.

Cipolla's is the only known q^{th} root algorithm which does not require computation of discrete logarithms. Unfortunately the algorithm is computationally more costly than computing a discrete logarithm, even using simple exhaustion, for all but the smallest primes q or if very large powers of q divide the order of the group (section 4.4). Furthermore, it produces a random oracle and does not satisfy the assumptions needed for the equivalency proof in [4]. If Cipolla's algorithm could be modified so that either the computational requirements were significantly reduced and/or it produced q^{th} roots which fit the needed q^{th} root oracle in the equivalency proof, then it could be used in several ways:

- If the algorithm could be modified such that the computational cost was less than that of current discrete logarithm techniques this would show that with current technology the q^{th} root problem over finite fields is computationally less expensive than the discrete logarithm problem.
- If the algorithm could also be modified to satisfy the requirements of the q^{th} root oracle, then it could be used in conjunction with the algorithm outlined in [4] and chapter 5 to compute discrete logarithms.

The goal of this research is to investigate Cipolla's algorithm, find out what modifications are possible and what impact these discoveries have on q^{th} root based cryptographic algorithms. We show that modifying Cipolla's algorithm in certain fields to minimize the computational requirements implies knowledge of a q^{th} root without using Cipolla's algorithm. This implies that Cipolla's algorithm can not be modified to assist with q^{th} root or discrete logarithm computation without the ability from another source to compute q^{th} roots. In other words Cipolla's algorithm has no impact on either the discrete logarithm or q^{th} root problems, or on the cryptography based on these systems.

Chapter 2

Number theory for finding roots and Cipolla's algorithm

This chapter covers the background number theory needed for the q^{th} root problem and Cipolla's algorithm. Cipolla's algorithm computes q^{th} roots, where q is prime, in a finite field \mathbb{F}_P with $q^2 \mid (P - 1)$.

The algorithm operates in the extension field \mathbb{F}_{P^q} . This section covers the theory and tools needed to examine both the q^{th} root problem and Cipolla's algorithm and to explore modifications to the algorithm.

This chapter describes well known concepts relevant to the q^{th} root problem. Section 2.1 deals with the unique, number theoretic traits of the order of $\mathbb{F}_{P^q}^*$, the group in which Cipolla's algorithm operates. The remaining sections describe more common number theoretic concepts but described in unique ways which enable them to be applied to the q^{th} root problem.

2.1 Number theoretic structure of $(P^q - 1)$

Cipolla's algorithm [7] was originally designed to compute square roots in a finite field, \mathbb{F}_P , and worked in a second degree extension field of \mathbb{F}_P . The extended version of Cipolla's algorithm computes q^{th} roots, where q is a prime integer, in a

q	prime integer, $q > 2$
P	order of base field \mathbb{F}_P
n	$n > 1$ and $q^n \parallel (P - 1)$
s	$s = \frac{(P-1)}{q^n}$
K	$K = \frac{(P^q-1)}{q(P-1)}$

Table 2.1: Parameters for \mathbb{F}_{P^q}

finite field where q divides the order of the multiplicative group. This algorithm works in a degree q extension field and is described in section 4.3. Since the algorithm operates in the multiplicative group of the extension field, the first step in understanding Cipolla's algorithm is to understand the form or factorization of the order of this group: $(P^q - 1)$. The following will be shown in this section: Let q be prime with $P = (q^n s + 1)$, $n > 1$ and $\gcd(q, s) = 1$ (table 2.1). Then

1. $P^q - 1 = q^{n+1} s K$ with $\gcd(K, P - 1) = 1$;
2. $K \bmod (P - 1) \equiv \begin{cases} 1 + s 2^{n-1} & \text{for } q = 2 \\ 1 & \text{otherwise} \end{cases}$
3. If $r \mid K$ then $r \equiv 1 \pmod{q}$.

Many of the properties hold for $n = 1$ but the q^{th} root problem is of interest only for $n > 1$.

Theorem 2.1.1. *If q is prime, $n \geq 1$ and $P = (q^n s + 1)$ with $\gcd(q, s) = 1$, then $(P^q - 1) = q^{n+1} s K$ with*

$$K \bmod (P - 1) \equiv \begin{cases} 1 + s 2^{n-1} & \text{if } q = 2 \\ 1 & \text{otherwise} \end{cases} . \quad (2.1.1)$$

Proof. We are given that $(P-1) = q^n s$ and know that $(P^q - 1) = (((P - 1) + 1)^q - 1)$.

Expanding this equation gives:

$$\begin{aligned}
P^q - 1 &= (-1) + \sum_{i=0}^q \binom{q}{i} (P-1)^i \\
&= (P-1) \sum_{i=1}^q \binom{q}{i} (P-1)^{i-1} \\
&= (P-1)q \left(q^{n(q-1)-1} s^{q-1} + \sum_{i=1}^{q-1} \frac{1}{q} \binom{q}{i} (P-1)^{i-1} \right)
\end{aligned}$$

therefore

$$\begin{aligned}
K &= \frac{(P^q - 1)}{q(P-1)} = q^{n(q-1)-1} s^{q-1} + \sum_{i=1}^{q-1} \frac{1}{q} \binom{q}{i} (P-1)^{i-1} \\
&= \left(q^{n(q-1)-1} s^{q-1} + 1 + (P-1) \sum_{i=2}^{q-1} \frac{1}{q} \binom{q}{i} (P-1)^{i-2} \right)
\end{aligned}$$

Therefore

$$K \bmod (P-1) \equiv \begin{cases} 1 + s2^{n-1} & \text{if } q = 2 \\ 1 & \text{otherwise} \end{cases} .$$

□

It is clear from this theorem that $\gcd(K, (P-1)) = 1$, as is shown in the following corollary.

Corollary 2.1.2. *Let q, P, n, s, K be defined as in table 2.1. Then $\gcd(K, P-1) = 1$.*

Proof. This follows directly from theorem 2.1.1: $K \equiv 1 \pmod{s}$ and since $n > 1$, $K \equiv 1 \pmod{q}$. Therefore $\gcd(K, s) = \gcd(K, q) = 1$ and $\gcd(K, q^n s) = \gcd(K, P-1) = 1$. □

At this point we know that $P^q - 1 = q^{n+1} s K$ with q, s, K pairwise relatively prime and that $K \equiv 1 \pmod{q^n s}$ for all $q > 2$. One final property of K will be shown: all divisors of K must be congruent to 1 modulo q . This implies that the smallest factor of K possible is $(2q + 1)$.

Theorem 2.1.3. *Let P, q, K be defined as table 2.1. If r is a prime divisor of K then:*

$$r \equiv 1 \pmod{q}. \quad (2.1.2)$$

Proof. Since $r \mid K \mid (P^q - 1)$ and $\gcd(K, P - 1) = 1$, we know that $P^q \equiv 1 \pmod{r}$ and $P \not\equiv 1 \pmod{r}$. So P is a non-trivial q^{th} root of unity modulo r . This implies that $\gcd(q, \phi(r)) > 1$. q, r are prime and $\gcd(q, \phi(r)) > 1$ together implies that $q \mid (r - 1)$ and therefore $r \equiv 1 \pmod{q}$. \square

Notice that although this theorem specifies r as a prime divisor of K , a trivial extension allows r to be any divisor of K .

For polynomial irreducibility testing (appendix B.1) and for proving certain relationships between conjugate elements in \mathbb{F}_{P^q} (lemma 9.1.1) more information about the relationship between q, K and $(P^i - 1)$ for $0 < i < q$ is needed. The following lemma details these relationships.

Lemma 2.1.4. *Let $a, b \in \mathbb{N}$ and $\gcd(a, b) = 1$. Then:*

$$\gcd\left(\frac{P^a - 1}{P - 1}, \frac{P^b - 1}{P - 1}\right) = 1 \quad (2.1.3)$$

and if $q \mid (P - 1)$ then

$$\frac{P^a - 1}{P - 1} \equiv a \pmod{q}$$

Proof. Corollary 3.7 in [29] states that $\gcd(x^a - 1, x^b - 1) = x^{\gcd(a,b)} - 1$. Since $\gcd(a, b) = 1$, $\gcd(P^a - 1, P^b - 1) = P - 1$. Therefore

$$\gcd\left(\frac{(P^a - 1)}{(P - 1)}, \frac{(P^b - 1)}{(P - 1)}\right) = 1$$

We know that $\frac{(P^a-1)}{(P-1)} = \sum_{i=0}^{a-1} P^i$ and $q|(P-1)$ implies $P \equiv 1 \pmod{q}$. Therefore

$$\begin{aligned} \frac{P^a - 1}{P - 1} &= \sum_{i=0}^{a-1} P^i \\ &\equiv a \pmod{q} \end{aligned}$$

□

2.2 The discrete logarithm function

The discrete logarithm function, the inverse of exponentiation in a finite cyclic group, can be found in numerous sources, such as [31] and [38]. The function is formally defined here because of its importance to the q^{th} root problem.

Definition 2.2.1 (discrete logarithm function): *Let G be a finite cyclic group, $g \in G$ be a generator with $\text{ord}(g) = R$, and $a \in G$ such that $a = g^x$ for some $x \in \mathbb{Z}_R$. The discrete logarithm function with base g , $dl_g: G \rightarrow \mathbb{Z}_R$ is defined as*

$$dl_g(a) = x. \tag{2.2.1}$$

Notice that the range of the function dl_g is defined here to be \mathbb{Z}_R instead of \mathbb{Z} . Any integer $y \in (x + R\mathbb{Z})$ is a valid solution to $dl_g(g^x)$ and any integer $y \notin (x + R\mathbb{Z})$ is not a solution. The discrete logarithm function is well defined when the range is set to \mathbb{Z}_R : Suppose $dl_g(a) = x$ and $dl_g(a) = y$. Then

$$\begin{aligned} g^x &= g^y \\ \implies g^{x-y} &= 1 \\ \implies x - y &| R \\ \implies x &\equiv y \pmod{R} \end{aligned}$$

2.3 The Chinese remainder theorem in finite cyclic groups

The Chinese remainder theorem (CRT) is a powerful tool used extensively in this research. A generalized proof of the CRT over ideals in a commutative ring is given in [25] and [19]. The CRT in the integers, combined with the exponentiation (or its inverse, the discrete logarithm function), can be used to reduce operations in a finite cyclic group into operations on its subgroups with co-prime order. This technique was used in [33] to simplify discrete logarithm computation and is often used to minimize computation in RSA – at least for the holder of the secret key.

It is well known that any finite cyclic group G of order R is isomorphic to the group \mathbb{Z}_R^+ . Furthermore if R can be factored into co-prime parts, $R = pq$, then the ring \mathbb{Z}_R is isomorphic to the ring $(\mathbb{Z}_p \times \mathbb{Z}_q)$. Combining these two isomorphisms gives us a group isomorphism from G to the subgroups of G of order p and q . However multiplicative operations in \mathbb{Z}_R and how they relate to the group G are central to root finding. For example, when $q \nmid R$, the isomorphic equivalent to computing a q^{th} root G is multiplication by $q^{-1} \in \mathbb{Z}_R$. For this reason, we extend the finite cyclic group to a finite ring. This new ring is isomorphic to \mathbb{Z}_R , uses the Diffie-Hellman oracle [9], and enables the q^{th} root problem to be reduced to the subrings of co-prime order.

Lemma 2.3.1. *Let G be a finite cyclic group of order R with $G = \langle g \rangle$, and define the binary operations (\oplus, \otimes) on G by:*

$$\begin{aligned} g^x \oplus g^y &= g^{x+y} \\ g^x \otimes g^y &= g^{xy}. \end{aligned}$$

Then $\widehat{G} = \langle G, \oplus, \otimes \rangle$ is a commutative ring with unity g .

Proof. Define $\theta_g: \mathbb{Z}_R \rightarrow G$ by $\theta_g(x) = g^x$. Since \mathbb{Z}_R is a commutative ring with unity 1, if θ_g is an isomorphism then \widehat{G} is a commutative ring with unity $g^1 = g$. The function θ_g is a ring isomorphism if it is a bijection and homomorphic under addition and multiplication. Since $G = \langle g \rangle$ is a finite cyclic group of order R , $\theta_g: \mathbb{Z}_R^+ \rightarrow G$ is a group isomorphism (bijection and homomorphic). It remains to be shown that θ_g is also homomorphic under multiplication.

Let $x, y \in \mathbb{Z}_R$. Then

$$\begin{aligned}\theta_g(xy) &= g^{xy} \\ &= \theta_g(x) \otimes \theta_g(y)\end{aligned}$$

Therefore θ_g is homomorphic under multiplication and θ_g is a ring isomorphism. Therefore \widehat{G} is a commutative ring with unity g . \square

The function θ_g is exponentiation, therefore its inverse is the discrete logarithm function: $\theta_g^{-1} = dl_g$ as defined in 2.2.1.

The second ring isomorphism needed is defined by the CRT. Let $R \in \mathbb{Z}$ be $R = pq$ with $\gcd(p, q) = 1$. The CRT function $\Psi: \mathbb{Z}_R \rightarrow (\mathbb{Z}_p \times \mathbb{Z}_q)$ is an isomorphism (corollary 2.27 [20]).

A composition of the exponentiation (θ_g) and CRT (Ψ) isomorphisms defines an isomorphism between \widehat{G} and its subrings of relatively prime order ($\widehat{\Psi}$). With this isomorphism operations in \widehat{G} , in particular computing q^{th} roots, can be performed using operations in the subgroups of relatively prime order.

Let the order of G be $R = pq$ where p, q are co-prime and $G_p = \langle g_p \rangle$, $G_q = \langle g_q \rangle$ be the subgroups G of order p and q respectively. Then the mapping $\theta_{g_p, g_q}: (\mathbb{Z}_p \times \mathbb{Z}_q) \rightarrow (\widehat{G}_p \times \widehat{G}_q)$ defined by

$$\theta_{g_p, g_q}(x, y) = (\theta_{g_p}(x), \theta_{g_q}(y)) = (g_p^x, g_q^y)$$

$$\begin{array}{ccc}
\widehat{G} & \xrightarrow{\theta_g^{-1}} & \mathbb{Z}_R \\
\widehat{\Psi} \downarrow & & \downarrow \Psi \\
(\widehat{G}_p \times \widehat{G}_q) & \xleftarrow{\theta_{g_p, g_q}} & (\mathbb{Z}_p \times \mathbb{Z}_q)
\end{array}$$

Figure 2.1: $\widehat{\Psi}$ function diagram

is also a ring isomorphism. Composing the isomorphisms $\theta_g^{-1}, \Psi, \theta_{g_p, g_q}$ gives the isomorphism $\widehat{\Psi}: \widehat{G} \rightarrow (\widehat{G}_p \times \widehat{G}_q)$ as in figure 2.1:

$$\widehat{\Psi}(a) = \theta_{g_p, g_q} \cdot \Psi \cdot \theta_g^{-1}(a) = (a^q, a^p) \quad (2.3.1)$$

The isomorphism $\widehat{\Psi}$ can be used in the q^{th} root problem, reducing the problem to finding roots in subgroups of prime order. Leaving the general group and concentrating on the group of order q^2s of interest, let $G = \langle g \rangle$ be the finite cyclic group of $\text{ord}(G) = q^2s$ where $\text{gcd}(q, s) = 1$ and $n > 1$, and $w \in G$ be the element we wish to find the q^{th} root of. If $g_s = g^{q^2}$ and $g_{q^2} = g^s$, then the subgroups $G_s = \langle g_s \rangle$ and $G_{q^2} = \langle g_{q^2} \rangle$ are subgroups of G with orders s and q^2 respectively. Finding a q^{th} root in G_s is trivial, as an inverse for q exists modulo s : $(w)^{\frac{1}{q}} = w^{q^{-1} \bmod s}$. The problem is not so trivial in G_{q^2} . No inverse for q exists in \mathbb{Z}_{q^2} . Simple exponentiation alone can not solve the problem. The best method currently known for computing a q^{th} root in this subgroup is to compute a discrete logarithm.

For computational purposes exact formulas for Ψ, Ψ^{-1} are given here as well as derivation of the generators for G_p, G_q . The CRT functions used in the examples are

$$\Psi(x) = (x \bmod p, x \bmod q). \quad (2.3.2)$$

and its inverse $\Psi^{-1}: (\mathbb{Z}_p \times \mathbb{Z}_q) \rightarrow \mathbb{Z}_R$ defined by

$$\Psi^{-1}(x_p, x_q) = q(x_p q^{-1} \bmod p) + p(x_q p^{-1} \bmod q) \bmod R. \quad (2.3.3)$$

These formulas can be found in [1].

Let $g_p = g^q$, $g_q = g^p$. The order of the subrings $\widehat{G}_p = \langle g_p \rangle$ and $\widehat{G}_q = \langle g_q \rangle$ are

$$\begin{aligned} \text{ord}(G_p) &= \frac{\text{ord}(g)}{\text{gcd}(\text{ord}(g), q)} = \frac{pq}{q} = p \\ \text{ord}(G_q) &= \frac{\text{ord}(g)}{\text{gcd}(\text{ord}(g), p)} = \frac{pq}{p} = q \end{aligned}$$

therefore they are generators of the co-prime subrings.

The following example demonstrates use of $\widehat{\Psi}$ to compute $(12)^{\frac{1}{3}}$ in \mathbb{Z}_{19} .

Example 2.3.2 – Compute $(12)^{\frac{1}{3}}$ in \mathbb{F}_{19} : The group used is $G = \mathbb{F}_{19}^* = \langle 2 \rangle$, which has order $18 = 2 \cdot 3^2$. Let $p = 2$, $q = 3$, $n = 2$, and $g_p = 2^{q^n} = 18$; $g_q = 2^p = 4$. Compute $(12)^{\frac{1}{3}}$:

1. Use $\widehat{\Psi}$ to map the problem into subgroups of order 9 and 2:

$$\widehat{\Psi}\left((12)^{\frac{1}{3}}\right) = \left(\left(12^9\right)^{\frac{1}{3}}, \left(12^2\right)^{\frac{1}{3}}\right) = \left(\left(18\right)^{\frac{1}{3}}, \left(11\right)^{\frac{1}{3}}\right);$$
2. Solve the root problem for G_2 : $3^{-1} \equiv 1 \pmod{p}$; $(18)^{\frac{1}{3}} \equiv 18$;
3. Since q divides the $dl_{g_q^n}(11)$, compute the discrete logarithm:

$$dl_{g_q^n}(11) \equiv 2 \pmod{3}; \therefore dl_{g_q^n}(11) \equiv 2 + 3x \pmod{9} \text{ for } 0 \leq x < 3;$$
4. Solve the q^{th} root problem in G_9 : $(11)^{\frac{1}{3}} \in \{g_q^2, g_q^5, g_q^8\} = \{16, 17, 5\}$;
5. The inverse values needed for the inverse of the CRT are: $q^{-1} \equiv 1 \pmod{p}$, $p^{-1} \equiv 5 \pmod{q}$;
6. Patch the two results together with the inverse CRT function:

$$\widehat{\Psi}^{-1}(18, 5) = 18^1 \times \{16, 17, 5\}^5 \equiv \{15, 13, 10\} \pmod{19}.$$

So any element of the set $\{15, 13, 10\}$ are q^{th} roots of 12 in \mathbb{F}_{19} .

2.4 Residues and residuosity testing

The focus of this research is on the q^{th} root problem in finite cyclic groups, particularly over finite fields. The concepts of residues and residuosity are used throughout this paper and are discussed in great detail in [28]. Unless otherwise stated, all groups are assumed to be finite cyclic groups.

The terms *residue* and *residuosity* most often refer to residues of order 2: quadratic residues or quadratic residuosity. A definition of quadratic residues is given in section 6.5 of [18]. The generalized concept of residues, that is of order greater than two, are defined in chapter 4.7.3 of [38]. Many techniques exist for exploring quadratic residuosity, such as quadratic reciprocity laws and Jacobi symbols, which do not generalize to residuosity of orders greater than 2. The definitions and algorithms given here are generalized for residues of prime order q in finite cyclic groups.

Definition 2.4.1 (q^{th} residue/non-residue): Let $q \in \mathbb{Z}$ with $q > 1$, G a finite cyclic group with $q \mid \text{ord}(G)$. $w \in G$ is a q^{th} residue (or q^{th} -power residue) in G if there exists an element $a \in G$ such that $a^q \equiv w$. If no such a exists then w is a q^{th} non-residue.

So a q^{th} residue is any element in G for which a q^{th} root exists.

Definition 2.4.2 (q^{th} residuosity): The q^{th} residuosity of an element w is its status as either a q^{th} residue or non-residue. Two elements, $w_1, w_2 \in G$, have the same q^{th} residuosity if w_1, w_2 are both q^{th} residues or both q^{th} non-residues in G . They have differing q^{th} residuosity if one element is a q^{th} residue and the other is a q^{th} non-residue.

Definition 2.4.1 leads to the following equivalent statements on q^{th} residues

in finite cyclic groups, a corollary concerning multiplication by q^{th} residues, and an algorithm for testing residuosity.

Lemma 2.4.3. *Let $G = \langle g \rangle$ be a finite cyclic group and $q \in \mathbb{Z}$ with $q \geq 2$ and $q \mid \text{ord}(G)$. If $w \in G$ then the following statements are equivalent:*

1. w is a q^{th} residue;
2. $\text{ord}(w) \mid \frac{\text{ord}(G)}{q}$
3. $dl_g(w) \equiv 0 \pmod{q}$.

Proof. From definition 2.4.1, $w \in G$ is a q^{th} residue if and only if there exists $a \in G$ such that $a^q = w$. Since g is a generator of G , there exists $t \in \mathbb{Z}$ such that $g^t = a$, therefore $g^{qt} = w$ and $dl_g(w) = qt \equiv 0 \pmod{q}$. Therefore w is a q^{th} residue if and only if $dl_g(w) \equiv 0 \pmod{q}$. We know that

$$\text{ord}(w) = \frac{\text{ord}(g)}{\text{gcd}(\text{ord}(g), dl_g(w))}$$

therefore $dl_g(w) \equiv 0 \pmod{q}$ if and only if

$$\text{gcd}\left(\frac{\text{ord}(g)}{q}, \frac{dl_g(w)}{q}\right) \text{ord}(w) = \frac{\text{ord}(g)}{q}$$

which implies that $dl_g(w) \equiv 0 \pmod{q}$ if and only if $\text{ord}(w) \mid \frac{\text{ord}(G)}{q}$. Therefore the three statements w is a q^{th} residue, $\text{ord}(w) \mid \frac{\text{ord}(G)}{q}$, and $dl_g(w) \equiv 0 \pmod{q}$ are equivalent. \square

Corollary 2.4.4 (Multiplication by q^{th} residue does not change residuosity).

Let $w \in G$ be a q^{th} residue and $a \in G$. Then the elements a, wa have the same q^{th} residuosity.

Proof. Let $G = \langle g \rangle$, $a \equiv g^{t_a}$ and $w \equiv g^{q^t}$. From lemma 2.4.3 a is a q^{th} residue if and only if $dl_g(a) = t_a \equiv 0 \pmod{q}$. Since $wa \equiv g^{qt+t_a}$, $dl_g(wa) = qt + t_a \equiv t_a \pmod{q}$, therefore wa has the same residuosity as a . \square

Residuosity testing on $w \in G$ determines whether or not w is a q^{th} residue in G . If w passes the q^{th} residuosity test then $ord(w) \mid \frac{ord(G)}{q}$, and w is a q^{th} residue. If w is a q^{th} residue then q^{th} roots exist for w .

Algorithm 2.4.5: q^{th} residuosity test

Input: $w \in G$

Output: an answer to the question ‘is w a q^{th} residue’

1: Compute $x \equiv w^{\frac{ord(G)}{q}}$

2: If $x \equiv 1$, w is a q^{th} residue; otherwise it is not.

2.4.1 Residuosity and discrete logarithms

Quadratic residuosity can also be used to determine the discrete logarithm, reduced modulo 2, of an element. If $G = \langle g \rangle$, $2 \mid ord(G)$ and $a \in G$, then from lemma 2.4.3 a is a quadratic residue if and only if $dl_g(a) \equiv 0 \pmod{2}$. If a is a quadratic non-residue, then $dl_g(a) \not\equiv 0 \pmod{2}$ which implies $dl_g(a) \equiv 1 \pmod{2}$. Therefore the discrete logarithm base g , modulo 2, of a is 0 if a is a quadratic residue and 1 if and only if it is a quadratic non-residue.

Example 2.4.6 – quadratic residuosity determines $dl_g \pmod{2}$: Let $G = \mathbb{Z}_{17}^*$ with generator $g = 3$. Quadratic residuosity testing in G implies raising an element to the 8 power. $7^8 \equiv 16 \pmod{17}$ so the $dl_g(7) \equiv 1 \pmod{2}$. $2^8 \equiv 1 \pmod{17}$ so the $dl_g(2) \equiv 0 \pmod{2}$. The discrete logarithms are $dl_g(7) \equiv 11 \pmod{16}$; $dl_g(2) \equiv 14 \pmod{16}$.

Square root algorithms which work in groups whose order is divisible by 4 use this fact either explicitly or implicitly. If 2^n is the largest power of 2 dividing the order of the group, with $n > 1$, then the discrete logarithm modulo 2^n is found using 2^i -th residuosity testing, where $0 < i < n$. Only the upper $(n - 1)$ bits of this discrete logarithm are needed as the low order bit must be 0 (i.e., the element must be a quadratic residue) for the root to exist.

Example 2.4.7 – square roots using residuosity testing: Let $G = \mathbb{Z}_{17}^*$ with generator $g = 3$, and G_t be the subgroups of G of order t . Find the square root of $w_0 = 15 \pmod{17}$. Let:

$$w_i \equiv g^{\sum_{j=i+1}^3 2^j x_j} \pmod{17}$$

$$a_i \equiv g^{\sum_{j=0}^{i-1} 2^j x_{j+1}} \pmod{17}$$

be the current remaining quadratic residue and square root values with $w_0 = 15$, $a_0 = 1$. The final values will be $w_2 = 1$, $a_2 = (w_0)^{\frac{1}{2}}$.

1. x_0 must be zero, otherwise no square root exists. Test this with residuosity testing: $w^8 = 15^8 \equiv 1 \pmod{17}$, so $x_0 = 0$
2. Compute x_1 by computing the quadratic residuosity of $w_0 \in G_8$:
 $w_0^4 \equiv 15^4 \equiv 16 \pmod{17}$, so $x_1 = 1$. Remove x_1 from w_0 :
 $w_1 \equiv w_0 g^{-2} = 15 \times 2 \equiv 13$; Add x_1 to a_0 : $a_1 \equiv a_0 \times g = 3$.
3. Compute x_2 by computing the quadratic residuosity of $w_1 \in G_4$:
 $w_1^2 = 13^2 \equiv 1 \pmod{17}$ so $x_2 = 1$. Remove x_2 from w_1 : $w_2 \equiv w_1 g^{-4} \equiv 1 \pmod{17}$. Add x_2 to a_1 : $a_2 \equiv a_1 g^2 \equiv 10 \pmod{17}$.
4. Compute x_3 by computing the quadratic residuosity of $w_2 \in G_2$:
 $w_2 \equiv 1$ so $x_3 = 0$.

The q^{th} root of 15 is $\pm a_2 \equiv \pm 10 \pmod{17}$.

For prime q greater than 2, q^{th} residuosity testing does not produce the discrete logarithm modulo q . If $G = \langle g \rangle$ and $a \in G$, then q^{th} residuosity testing only determines if $dl_g(a) \equiv 0 \pmod{q}$. If q is large and a is a q^{th} non-residue, residuosity testing gives very little information. Residue testing can not help compute q^{th} roots for these larger q . Instead, discrete logarithms are computed. If q^n is the largest power of the prime q dividing the order of the group, that is $q^n \parallel ord(G)$, $n - 1$ discrete logarithms, using a base of order q , must be found. These algorithms are described in chapter 4.

2.5 q^{th} roots of unity

Another important tool for this research is a primitive q^{th} root of unity.

Definition 2.5.1 (q^{th} root of unity and primitive q^{th} root of unity):

A q^{th} root of unity in a finite cyclic group G is an element $h \in G$ such that $h^q = 1$. h is a primitive q^{th} root of unity if $h^i \neq 1 \forall 0 < i < q$.

The element 1 is always a q^{th} root of unity for any q . If q is prime and $h \neq 1$ is a q^{th} root of unity, then it must also be a primitive q^{th} root of unity. Notice that if $h \in G$ is a primitive q^{th} root of unity, then $G_q = \langle h \rangle$, the subgroup of G of order q .

The following lemma is well known and can be found in [1]. It applies to q^{th} roots of unity in the multiplicative group of a finite field.

Lemma 2.5.2. *Let q be a prime integer and $h \neq 1$ be q^{th} root of unity in a field \mathbb{F} . Then:*

$$\sum_{i=0}^{q-1} h^{it} \equiv 0 \pmod{q} \forall 0 < t < q \quad (2.5.1)$$

If $w \in G$ is a q^{th} residue then it is well known that it has exactly q distinct roots. Multiplying a given q^{th} root by any q^{th} root of unity produces another q^{th} root.

Lemma 2.5.3 (There are q distinct roots for every q^{th} residue). *If $w \in G$ is a q^{th} residue then there are q distinct solutions to $(w)^{\frac{1}{q}}$.*

Proof. Let $n_r(w)$ be the number of q^{th} roots for w . By definition, a q^{th} residue has at least one solution a to $a^q = w$, or $a = (w)^{\frac{1}{q}}$. For any $h \in G_q$, $(ah)^q = a^q(1)$. If $b \in G$ is a q^{th} root of w , then $(a^{-1}b)^q = w^{-1}w = 1$. Therefore $a^{-1}b = h \in G_q$, and $b = ah$. This implies that the set of q^{th} roots of w is exactly the coset $\{ah \mid h \in G_q\}$ and $n_r(w) = q$ for all q^{th} residues $w \in G$. \square

q^{th} roots of unity also play an important role in chapter 9. Using q^{th} roots of unity in functions closely resembling Fourier transforms we can:

1. easily compute all conjugates of a given element;
2. represent q irreducible polynomials for use in Cipolla's algorithm with a single set of q elements in \mathbb{F}_P .

2.6 Conjugates, Norms, and Traces

Cipolla's algorithm for finding q^{th} roots in \mathbb{F}_P , described in section 4.3, operates in the extension field \mathbb{F}_{P^q} . Conjugates, norms and traces of elements in \mathbb{F}_{P^q} are crucial to understanding the algorithm. General definitions can be found in [20] and [26]; definitions and theory over finite fields can be found in [29]. The following definitions are specialized for the extension field, \mathbb{F}_{P^q} .

Definition 2.6.1 (Minimal polynomial): *The minimal polynomial for $\alpha \in \mathbb{F}_{P^q}$ is the monic polynomial, $f(x)$, of lowest degree $n \geq 1$ over \mathbb{F}_P for which $f(\alpha) \equiv 0$.*

Definition 2.6.2 (Degree of $\alpha \in \mathbb{F}_{P^q}$): *The degree of α over \mathbb{F}_{P^q} is defined to be the degree of its minimal polynomial*

Lemma 2.6.3. *Every element $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$, where q is prime, has degree q .*

Proof. Let $\alpha \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$. The degree of α must divide q (theorem 1.86 in [29]), and since q is prime $\deg(\alpha) \in \{1, q\}$. An element $\alpha \in \mathbb{F}_{P^q}$ is the root of a degree 1 polynomial if and only if $\alpha \in \mathbb{F}_P$. Therefore every element in $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$ must have degree q . □

Definition 2.6.4 (Conjugates): *If $\alpha \in \mathbb{F}_{P^q}$ has minimal polynomial $f(x)$ of degree m , then the conjugates of α are the distinct elements $\{\alpha^{P^j} \mid 0 \leq j < m\}$. These elements are the m roots of $f(x)$ (theorem 2.14 in [29]).*

Any element in $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$ has degree q , and therefore has q conjugates, including itself. Since the conjugates of $\alpha \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$ are the distinct roots its minimal polynomial $f(x)$, we know that

$$f(x) = \prod_{j=0}^{q-1} (x - \alpha^{P^j}). \quad (2.6.1)$$

The following defines norms and traces for both elements in \mathbb{F}_{P^q} and irreducible degree q polynomials over \mathbb{F}_P . The norm and trace of an irreducible degree q polynomial over \mathbb{F}_P is simply the norm and trace of one of its roots.

Definition 2.6.5 (Norm): The norm of $\alpha \in \mathbb{F}_{P^q}$ is the function $N : \mathbb{F}_{P^q} \rightarrow \mathbb{F}_P$ defined by:

$$N(\alpha) \equiv \prod_{i=0}^{q-1} \alpha^{P^i} \quad (2.6.2)$$

The norm of an irreducible polynomial $f(x)$ over \mathbb{F}_P with root $\alpha \in \mathbb{F}_{P^q}$, with respect to \mathbb{F}_{P^q} , is $N_{\mathbb{F}_{P^q}}(f(x)) = N(\alpha)$.

Definition 2.6.6 (Trace): The trace of $\alpha \in \mathbb{F}_{P^q}$ is the function $Tr : \mathbb{F}_{P^q} \rightarrow \mathbb{F}_P$ defined by:

$$Tr(\alpha) \equiv \sum_{i=0}^{q-1} \alpha^{P^i} \quad (2.6.3)$$

The trace of an irreducible polynomial $f(x)$ over \mathbb{F}_P , with respect to \mathbb{F}_{P^q} , is the trace of any one of its roots in \mathbb{F}_{P^q} : if $\alpha \in \mathbb{F}_{P^q}$ such that $f(\alpha) = 0$ then $Tr(f) = Tr(\alpha)$.

For elements in $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$ the sum and product of all conjugates of an element have special properties. The following theorems describe a collection of these properties. Variants on these theorems and the proofs can be found in chapter 3 of [29]. The first theorem describes properties of roots of degree q monic irreducible polynomials. The second theorem describes properties of norms and traces in general.

Theorem 2.6.7. Let $f(x) = \sum_{i=0}^q c_i x^i$ be a degree q monic irreducible polynomial over \mathbb{F}_P with root α . Then:

1. $N(\alpha) = (-1)^q c_0$
2. $Tr(\alpha) = (-1)c_{q-1}$

Theorem 2.6.8. Let $\alpha \in \mathbb{F}_{P^q}$ Then:

1. the norm is

$$N(\alpha) = \alpha^{\left(\frac{P^q-1}{P-1}\right)}$$

2. The norm and trace are mappings from \mathbb{F}_{P^q} to \mathbb{F}_P ;

3. The norm and trace are multiplicative and additive respectively:

$$N(\alpha_1\alpha_2) = N(\alpha_1)N(\alpha_2)$$

$$Tr(\alpha_1 + \alpha_2) = Tr(\alpha_1) + Tr(\alpha_2)$$

The exponent used in the norm function has a particularly nice form when $q|(P-1)$. Recall from section 2.1 that if $(P-1) = q^n s$ where $\gcd(q, s) = 1$, then $(P^q - 1) = q^{n+1} s K$ where $\gcd(q^n s, K) = 1$. The exponent used in the norm function for these fields is $\frac{P^q-1}{(P-1)} = qK$. Therefore if $N(\eta) = \eta^{qK} = w$ then

$$(w)^{\frac{1}{q}} = \eta^K. \tag{2.6.4}$$

This is the heart of Cipolla's q^{th} root algorithm, described in section 4.3.

Although nice formulas exist for computation of the norm, few exist for computing the trace. A formula for the trace function for a special class of polynomials is given in section 8.3. This formula is crucial in demonstrating that Cipolla's algorithm has no impact on the q^{th} root problem.

The concept of a reciprocal polynomial is used in the development of this trace formula. A definition of a reciprocal polynomial over \mathbb{F}_2 can be found in [30].

Definition 2.6.9 (Reciprocal of an irreducible polynomial): *Let $f(x)$ be the minimal polynomial for $\alpha \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$. Then the **reciprocal** of f is \bar{f} , the minimal polynomial for α^{-1} .*

The coefficients for the reciprocal polynomial can be derived directly from the coefficients of $f(x)$.

Lemma 2.6.10. *Let the minimal polynomial for $\alpha \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$ be $f(x) = \sum_{i=0}^q c_i x^i$.*

Then the reciprocal polynomial of $f(x)$ is

$$\bar{f}(x) = (c_0)^{-1} \sum_{i=0}^q c_{q-i} x^i \quad (2.6.5)$$

Proof. Evaluating $\bar{f}(x)$ at α^{-1} gives us

$$\begin{aligned} \bar{f}(\alpha^{-1}) &\equiv \alpha^{-q} (c_0)^{-1} \sum_{i=0}^q c_{q-i} \alpha^{q-i} \\ &\equiv \alpha^{-q} (c_0)^{-1} f(\alpha) \\ &\equiv 0 \end{aligned}$$

Furthermore, \bar{f} is monic since $c_0^{-1} c_{q-q} = 1$. Therefore \bar{f} is the minimal polynomial for α^{-1} and the reciprocal polynomial for f . \square

2.7 Representation of \mathbb{F}_{P^q}

All fields of order P^q are isomorphic therefore any field of order P^q can be used to implement and analyze Cipolla's algorithm. However, choosing an appropriate representation for \mathbb{F}_{P^q} simplifies and clarifies the algorithm. There are several ways to represent elements in an extension field, as described in chapter 2.5 of [29]. Two methods will be described: one for strictly theoretical purposes and the second for both theoretical and computational purposes. The first representation method uses the fact that $\mathbb{F}_{P^q}^*$ is a finite cyclic group and $\widehat{\mathbb{F}_{P^q}^*}$ is isomorphic to \mathbb{Z}_{P^q-1} . If $\gamma \in \mathbb{F}_{P^q}$ is a generator, then every element, $\alpha \in \mathbb{F}_{P^q}^*$ can be expressed as $\alpha = \gamma^i$ for some $i \in \mathbb{Z}_{P^q-1}$, and $\mathbb{F}_{P^q} = \{0\} \cup \{\gamma^i \mid 0 \leq i < (P^q - 1)\}$. In section 7.1 this representation is used to detail the inner workings of Cipolla's algorithm and how q^{th} residues are mapped to their respective q^{th} roots.

The second method, used for computational and theoretical purposes, is the residue class ring representation of \mathbb{F}_{P^q} . Let $f(x) = \sum_{i=0}^q c_i x^i \in \mathbb{F}_P[x]$ be a monic

irreducible polynomial of degree q . Then $\mathbb{F}_{P^q} = \mathbb{F}_P[x]/(f(x))$ and operators in the field will be straight forward polynomial multiplication and addition, reduced with $x^q \equiv -\sum_{i=0}^{q-1} c_i x^i$. Vector notation will often be used to represent elements in $\mathbb{F}_P[x]/(f(x))$. If $\beta = \sum_{i=0}^{q-1} a_i x^i \in \mathbb{F}_P[x]/(f(x))$, then the vector notation for β is $\beta = [a_{q-1} \ a_{q-2} \ \dots \ a_1 \ a_0]$.

All fields of order P^q are isomorphic, therefore the choice of the field representation used for computation or analysis has no effect on the theoretical results. However, if the representation $\mathbb{F}_P[x]/(f(x))$ is used, the choice of the monic irreducible polynomial $f(x)$ of degree q does have an effect on computation and some polynomials illuminate the structure of the field better than others. For example, there are three irreducible polynomials in $\mathbb{F}_3[x]$: $f_1(x) = x^2 + x + 2$, $f_2(x) = x^2 + 2x + 2$, $f_3(x) = x^2 + 1$. Irreducible binomial polynomials, such as f_3 can minimize computation costs and, as is shown in chapter 9, enable alternate views on conjugates and determining q^{th} roots. Irreducible polynomials of this form only exist when $q|(P-1)$. If $a \in \mathbb{F}_P$, $(x^q - a)$ is irreducible if and only if the order of a is divisible by q^n .

Lemma 2.7.1 (Irreducibility of $(x^q - a)$). *Let q, P, n, s, K be defined as in table 2.1 (page 20). Then $f(x) = x^q - a$, $a \in \mathbb{F}_P^*$, is irreducible over \mathbb{F}_P if and only if q^n divides the order of a .*

Proof. Let $\gamma \in \mathbb{F}_{P^q}$ be a generator, $f(x) = (x^q - a)$, and recall from theorem 2.1.1 that $(P-1) = q^n s$ and $(P^q - 1) = q^{n+1} s K$. The order of γ^{qK} is $\text{ord}(\gamma^{qK}) = \frac{q^{n+1} s K}{qK} = q^n s$, so $\gamma^{qK} \in \mathbb{F}_P$ is a generator of \mathbb{F}_P and there exists $t \in \mathbb{Z}_{q^n s}$ such that $a = \gamma^{qKt}$. The order of a is $\text{ord}(a) = \text{ord}(\gamma^{qKt}) = \frac{q^n s}{\text{gcd}(q^n s, t)}$, so $q^n | \text{ord}(a)$ if and only if $\text{gcd}(q, t) = 1$.

Since $f(\gamma^{Kt}) = (\gamma^{Kt})^q - a = 0$, we know that γ^{Kt} is a root of $f(x)$. Furthermore,

$$\gamma^{Kt} \in \begin{cases} \mathbb{F}_P & \iff \text{ord}(\gamma^{Kt}) \mid q^n s & \iff \text{gcd}(q, t) = q \\ \mathbb{F}_{P^q} \setminus \mathbb{F}_P & \iff \text{ord}(\gamma^{Kt}) \nmid q^n s & \iff \text{gcd}(q, t) = 1 \end{cases}$$

and since q is prime the degree for the minimal polynomial, $m(x)$, for γ^{Kt} is:

$$\deg(m(x)) = \begin{cases} 1 & \gamma^{Kt} \in \mathbb{F}_P \\ q & \gamma^{Kt} \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P \end{cases} .$$

Since $m(\gamma^{Kt}) = f(\gamma^{Kt}) = 0$ and $m(x)$ is the minimal polynomial for γ^{Kt} , we know that $m(x) \mid f(x)$. Therefore if $\gamma^{Kt} \in \mathbb{F}_P$, $f(x)$ is divisible by $m(x)$ with $\deg(m(x)) < \deg(f(x))$ and is not irreducible. If $\gamma^{Kt} \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$, $\deg(m(x)) = \deg(f(x))$ and since both $m(x), f(x)$ are monic this implies that $m(x) = f(x)$ and $f(x)$ is irreducible. Therefore $f(x) = (x^q - a)$ is irreducible if and only if $q^n \mid \text{ord}(a)$

□

Chapter 3

Discrete logarithm techniques

Although the discrete logarithm or index computation problem, described in figure 3.1, has been studied for centuries, techniques for solving it, other than exhaustion, have only been discovered in recent decades. The best general purpose discrete logarithm techniques run in square root time. Special purpose algorithms run much faster but are restricted to groups whose underlying structure can be mapped to a unique factorization domain, such as finite fields.

The Diffie-Hellman key exchange [9] was the first published public key cryptosystem and is based on the discrete logarithm problem. Many more discrete logarithm based public key cryptosystems followed, such as the ElGamal [13], Schnorr [39], and Feige-Fiat-Shamir [14] signature schemes. The DSA (Digital Signature Algorithm) [11] is based on the discrete logarithm problem as are many other key exchanges, including single pass authenticated key exchanges as in [27] and [22]. The difficulty of both factoring and discrete logarithm problems is still unknown, even though most public key systems are based on one of these two problems. Confidence in the difficulty of these problems arises from the intense scrutiny they have received in the past few decades. This section gives an overview of the discrete logarithm algorithms designed during this period.

All techniques for finding discrete logarithms are based on the commutative

G	a finite cyclic group
γ, α	elements in G such that γ generates α
t	a non-negative integer such that $\gamma^t = \alpha$
Given γ, α find t	

Figure 3.1: The Discrete Logarithm Problem

properties of cyclic groups. The techniques can be divided into two classes. The first type are the square root algorithms. These work for any cyclic group. The algorithms can be improved and parallelized using the ring isomorphism $\widehat{\Psi}$ in section 2.3, as was done in [33].

The second type of discrete logarithm algorithm is index calculus. Index calculus works over the multiplicative group of a quotient ring, D/I , where D is a Euclidian domain, such as $\mathbb{Z}/P\mathbb{Z}$. It is a two-phase algorithm: phase one generates a look-up table, phase two uses the look-up table to compute the discrete logarithm. Phase one, the generation of the look-up table, is the computationally expensive phase. This phase is generally solved using a sieving technique such as [32], and many special purpose sieving algorithms have been designed to work with \mathbb{F}_P where P has special form. Once phase one is finished and the look-up table is built, any number of discrete logarithms can be found using the same look-up table. The speed of phase two, finding individual discrete logarithms, depends on the size of the look-up table generated in phase one. The majority of the computational effort in the most efficient algorithms is in phase one.

Example 3.0.2 – Discrete logarithm via exhaustion: For this example let $G = \mathbb{F}_{3^2}^*$, represented by $\mathbb{F}_3[\gamma]$ where γ is a root of $f(x) = x^2 + x + 2$. Find the discrete logarithm of $\alpha = \gamma + 2$ base γ .

t	0	1	2	3	4	5	6	7
γ^t	1	γ	$2\gamma + 1$	$2\gamma + 2$	2	2γ	$\gamma + 2$	$\gamma + 1$

From the table $dl_\gamma(\alpha) = 6$.

3.1 Discrete logarithm algorithms with square root run-time

The class of algorithms described in this section have several common attributes. First, they work over any finite cyclic group. Second, if the order of the group is n , run time for the algorithms is generally $O(\sqrt{n})$.

Among the square root algorithms are two further subdivisions: deterministic and random. The deterministic algorithm was first described in [41] and is known as baby-step-giant-step or meet-in-the-middle. Its generalized form, described here, solves the discrete logarithm and many other problems, as in [44]. Randomized versions were first designed by Pollard [34]. They drastically reduce the storage requirements of the deterministic algorithm, using traits of the birthday problem and the coalescing nature of random mappings.

The run time for many of the algorithms in this section can be reduced using the isomorphism $\widehat{\Psi}$ as described in section 2.3, and working in subgroups of G . This improvement is also described in [33].

3.1.1 Baby-step-giant-step

The baby-step-giant-step or meet-in-the-middle algorithm is described in [41] and [10] and is one of the most simple and elegant of all cryptanalytic algorithms. It not only computes discrete logarithms, but can be applied to a variety of other problems.

The algorithm solves the following problem: Let E_x be an indexed, invertible function. Given a pair, (β, a) , find a value x for which $E_x(\beta) = a$. In some cases there may be multiple values, x , which solve this problem.

The baby-step-giant-step algorithm works over three sets of indexed, easily

invertible functions:

$$\Upsilon = \{E_x : S \rightarrow S \mid x \in X\}$$

$$\Upsilon_1 = \{K_{x_1} : S \rightarrow S \mid x_1 \in X_1\}$$

$$\Upsilon_2 = \{H_{x_2} : S \rightarrow S \mid x_2 \in X_2\}$$

and a surjective mapping on the index sets, $f : (X_1 \times X_2) \rightarrow X$, such that if $f(x_1, x_2) = x$ then $H_{x_2} \circ K_{x_1} = E_x$. The following are a few examples of possible sets and functions f :

- Let $\langle \gamma \rangle = G$ be a finite cyclic group of order n and $m = \lceil \sqrt{n} \rceil$.

$$\Upsilon = \{E_x(\beta) = \beta\gamma^x \mid \beta \in \langle \gamma \rangle; x \in \mathbb{Z}_n\}$$

$$\Upsilon_1 = \{K_{x_1}(\beta) = \beta\gamma^{x_1} \mid \beta \in \langle \gamma \rangle; x_1 \in \mathbb{Z}_m\}$$

$$\Upsilon_2 = \{H_{x_2}(\beta) = \beta(\gamma^m)^{x_2} \mid \beta \in \langle \gamma \rangle; x_2 \in \mathbb{Z}_m\},$$

and $f(x_1, x_2) = x_1 + mx_2 \pmod n$.

- Let $\langle \gamma \rangle = G$ be a finite cyclic group of order $n < 2^m$, and let $W(x) : \mathbb{Z}_{2^m} \rightarrow \mathbb{Z}_m$ be the Hamming weight function on x .

$$\Upsilon = \{E_x(\beta) = \beta\gamma^x \mid \beta \in \langle \gamma \rangle; x \in \{x \in \mathbb{Z}_{2^m} \mid W(x) < 2k\}\}$$

$$\Upsilon_1 = \{K_{x_1}(\beta) = \beta\gamma^{x_1} \mid \beta \in \langle \gamma \rangle; x_1 \in \{x_1 \in \mathbb{Z}_{2^m} \mid W(x_1) < k\}\}$$

$$\Upsilon_2 = \{H_{x_2}(\beta) = \beta\gamma^{x_2} \mid \beta \in \langle \gamma \rangle; x_2 \in \{x_2 \in \mathbb{Z}_{2^m} \mid W(x_2) < k\}\},$$

and $f(x_1, x_2) = x_1 + x_2 \pmod{2^m}$.

- Let H, K be m -bit block ciphers using n -bit keys. Then

$$\Upsilon = \{E_x(\beta) = H_{x_2}(K_{x_1}(\beta)) \mid x = (x_1, x_2) \in \mathbb{F}_{2^n} \times \mathbb{F}_{2^n}\}$$

$$\Upsilon_1 = \{K_{x_1} : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^m} \mid x_1 \in \mathbb{F}_{2^n}\}$$

$$\Upsilon_2 = \{H_{x_2} : \mathbb{F}_{2^m} \rightarrow \mathbb{F}_{2^m} \mid x_2 \in \mathbb{F}_{2^n}\}$$

with $f(x_1, x_2) = (x_1, x_2)$.

Since f is a surjection, for every $x \in X$ there exists at least one pair, $(x_1, x_2) \in (X_1 \times X_2)$, such that $f(x_1, x_2) = x$. Given $E_x(\beta) = \alpha$, we know that $H_{x_2}(K_{x_1}(\beta)) = \alpha$. Therefore, $K_{x_1}(\beta) = H_{x_2}^{-1}(\alpha)$. The algorithm searches for a pair, (x_1, x_2) , such that $K_{x_1}(\beta) = H_{x_2}^{-1}(\alpha)$. If it finds such a pair then $E_{f(x_1, x_2)}(\beta) = \alpha$.

The most common way to find a feasible pair, (x_1, x_2) , is with exhaustion. Using the input pair (β, α) , the two sets

$$\{(x_1, K_{x_1}(\beta)) \mid x_1 \in X_1\} \quad \{(x_2, H_{x_2}^{-1}(\alpha)) \mid x_2 \in X_2\}$$

are generated and searched to find (x_1, x_2) such that $K_{x_1}(\beta) = H_{x_2}^{-1}(\alpha)$. Therefore the run-time for this algorithm is $O(\max(|X_1|, |X_2|))$. Since f is a surjection, $|X_1| \times |X_2| \geq |X|$, so the optimal size for X_1, X_2 is $\sqrt{|X|}$. When this can be accomplished, the algorithm runs in $O(\sqrt{|X|})$ time.

The first example of possible baby-step-giant-step sets given above is used to solve the general purpose discrete logarithm problem. Let $G = \langle \gamma \rangle$ be a finite cyclic group. In this problem we want to find the discrete logarithm, base γ , in G . The baby-step-giant-step algorithm computes a discrete logarithm using the following sets and function:

$$\Upsilon = \{E_x(\beta) = \beta\gamma^x \mid \beta \in \langle \gamma \rangle; x \in \mathbb{Z}_n\}$$

$$\Upsilon_1 = \{K_{x_1}(\beta) = \beta\gamma^{x_1} \mid \beta \in \langle \gamma \rangle; x_1 \in \mathbb{Z}_m\}$$

$$\Upsilon_2 = \{H_{x_2}(\beta) = \beta(\gamma^m)^{x_2} \mid \beta \in \langle \gamma \rangle; x_2 \in \mathbb{Z}_m\},$$

and $f(x_1, x_2) = x_1 + mx_2 \pmod n$, where $\text{ord}(\gamma) = n$ and $m = \lceil \sqrt{n} \rceil$. The input pair we are given is $(1, E_x(1) = \gamma^x = \alpha)$, and the two sets which must be generated are $\{(x_1, \gamma^{x_1}) \mid 0 \leq x_1 < m\}$ and $\{(x_2, \alpha(\gamma^{-m})^{x_2}) \mid 0 \leq x_2 < m\}$. To find the discrete logarithm of α base γ , find a pair, (x_1, x_2) , such that $\gamma^{x_1} = \alpha(\gamma^m)^{x_2}$. The discrete logarithm will be $x = x_1 + mx_2 \pmod n$.

This technique is used in the following example.

Example 3.1.1 – Baby-step-giant-step Algorithm: Find the discrete logarithm of $\alpha = 3$ base $\gamma = 2$ modulo 101 with $n = 100$ and $m = 10$.

i	0	1	2	3	4	5	6	7	8	9
2^i	1	2	4	8	16	32	64	27	54	7
$3(2^{-10})^i$	3	94	50	18	59	98	7	51	83	42

The matching pair found at $(x_1, x_2) = (9, 6)$ which gives $dl_2(3) = 9 + 10 \times 6 = 69$.

One of the main drawbacks of the baby-step-giant-step algorithm, compared to other square root algorithms, is the large memory requirement of $2m$. Although these requirements can be reduced the randomized techniques require even less. On the other hand, this technique guarantees a solution; the randomized algorithms do not.

3.1.2 Pollard’s randomized techniques

Pollard’s Rho and Kangaroo traps [34] are two randomized square root discrete logarithm algorithms. Parallel improvements to these collision search techniques can be found in [45]. The coalescing properties of random mappings are used to find the discrete logarithm in about the same run time as the baby-step-giant-step algorithm, but with almost no storage requirements. Baby-step-giant-step, on a group G of order $ord(G) = n$, required storage for around $(2\sqrt{n})$ elements in G and their corresponding function indices. Random mapping techniques require storage for only a hand full of these elements and indices.

Random mappings coalesce into cycles, forming a shape something like ρ , as shown in figure 3.2. This gives the name to Pollard’s first algorithm. Let $\langle \gamma \rangle = G$ be the finite cyclic group over which a discrete logarithm is sought with $ord(G) = n$. Given α , the goal is to find an $x \in \mathbb{Z}_n$ such that $\gamma^x = \alpha$.

Pollard Rho is based on a sequence, $\delta_i \in G$, guided by a random mapping $f : (\mathbb{Z}_n \times \mathbb{Z}_n) \rightarrow (\mathbb{Z}_n \times \mathbb{Z}_n)$. If $f(u_{i-1}, v_{i-1}) = (u_i, v_i)$, the sequence δ_i is defined by $\delta_i = \gamma^{u_i} \alpha^{v_i}$ for $i > 0$. The assumption is that δ_i will eventually reach a point i such that $\delta_i = \delta_{2i}$. Starting at a known point, (u_0, v_0) , (δ_i, δ_{2i}) are computed until $\delta_i = \delta_{2i}$. This point generates the following equation:

$$\begin{aligned}\gamma^{u_i} \alpha^{v_i} &= \delta_i = \delta_{2i} = \gamma^{u_{2i}} \alpha^{v_{2i}} \\ \gamma^{u_i + xv_i} &= \gamma^{u_{2i} + xv_{2i}}\end{aligned}$$

This gives us the following equation in \mathbb{Z}_n :

$$u_i - u_{2i} \equiv x(v_{2i} - v_i) \pmod{n} \quad (3.1.1)$$

Solving for x in equation (3.1.1) gives us the discrete logarithm – or at least part of it.

Occasionally an equation is found which generates multiple solutions for x . The discrete logarithm function (see 2.2.1) is well defined, which implies that only one of these solutions is the discrete logarithm. In this case only the discrete logarithm of α modulo $\frac{n}{t}$ is found. The discrete logarithm modulo t will still need to be computed. If $\gcd(n, v_{2i} - v_i) = t$, the discrete logarithm of α base γ modulo $\frac{n}{t}$ is

$$x \equiv \frac{u_i - u_{2i}}{t} \left(\frac{v_{2i} - v_i}{t} \right)^{-1} \pmod{\frac{n}{t}} \quad (3.1.2)$$

Example 3.1.2 – Pollard Rho: Using the parameters from example 3.1.1, let $G = \mathbb{F}_{101}^*$, $\gamma = 2$ and $\alpha = 3$. Let $\delta_i \in \mathbb{Z}_{101}$ be represented by $\delta_i \in \{0, 1, \dots, 100\}$. The function f will be determined by the residue class modulo 3 of δ_i in this reduced form.

$$(f(u_i, v_i), \delta_i) = \begin{cases} ((u_i + 1, v_i), 2\delta_i) & \text{if } \delta_i \equiv 0 \pmod{3} \\ ((u_i, v_i + 1), 3\delta_i) & \text{if } \delta_i \equiv 1 \pmod{3} \\ ((3u_i, 3v_i), \delta_i^3) & \text{if } \delta_i \equiv 2 \pmod{3} \end{cases}$$

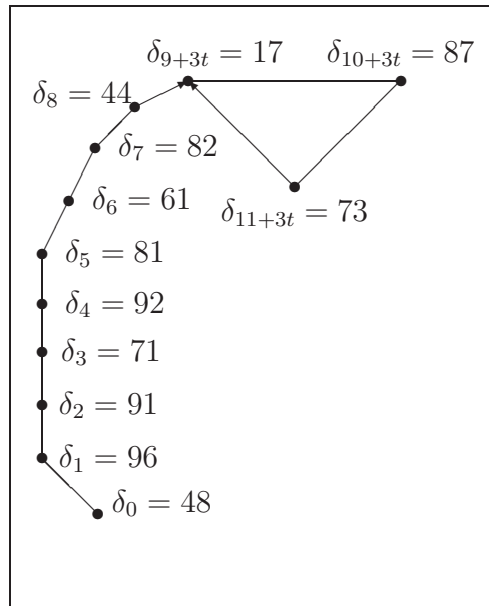


Figure 3.2: Pollard Rho Example

A starting point of $(u_0, v_0) = (4, 1)$ is randomly generated. Start the sequences δ_i, δ_{2i} with $\delta_0 = 48$. The following chart show the progress of δ_i, δ_{2i} , with it's eventual match at $i = 9$, as does figure 3.2:

i	(u_i, v_i)	δ_i	δ_{2i}	(u_{2i}, v_{2i})
0	(4, 1)	48	48	(4, 1)
1	(5, 1)	96	91	(6, 1)
2	(6, 1)	91	92	(12, 4)
3	(6, 2)	71	61	(25, 8)
4	(12, 4)	92	44	(25, 10)
5	(24, 8)	81	87	(0, 40)
6	(25, 8)	61	17	(1, 41)
7	(25, 9)	82	73	(3, 82)
8	(25, 10)	44	87	(6, 66)
9	(50, 20)	17	17	(7, 67)
10	(0, 40)	87	73	(15, 34)

Since $\delta_i = \delta_{2i}$ at $i = 9$, the $(u_9, v_9), (u_{18}, v_{18})$ pairs gives:

$$50 - 7 \equiv x(67 - 20) \pmod{100}$$

$$43 \equiv 47x \pmod{100}$$

$$69 \equiv x \pmod{100}$$

and the discrete logarithm of 3 base 2 is 69.

Pollard's second algorithm, Kangaroo traps, uses a slightly different model. Instead of stepping a single random mapping in lock step and finding a match point, different start points of the random mapping are used, one (or more) 'tame' point starting at a known exponent and one (or more) 'wild' point starting at the unknown discrete logarithm. The tame points are stepped a sufficient number of steps (around $\sqrt{\text{ord}(\gamma)}$) to place them on the one of the cycles of the random mapping. That end point (a trap) is stored. The wild point is then stepped and its value checked against the value of the trap. The hope is that it will, at some point in its travels, be equivalent to a stored value (fall into the trap). If this happens then an equation once again exists for the discrete logarithm.

Let $f(u_{i-1}) = u_i, \delta_i = \gamma^{u_i}$. Wild traps will have the variable x in its representation of u_i . Starting at one or more chosen points u_0 , the function is stepped and the pair (δ_n, u_n) computed ($n \approx \sqrt{\text{ord}(\gamma)}$). Now start at $u_0 = x$ or $u_0 = x + c$ for some known integer c and compute the wild pairs, comparing them with the traps. Recall that the wild value for u_i will be a function on x .

Multiple traps and multiple wild kangaroos are often launched as there are generally many disjoint cycles. The individual traps are independent of each other, therefore can be set in parallel. The wild kangaroos are also independent of each other, but each wild kangaroo will need the entire set of set traps to check itself against.

Example 3.1.3 – Kangaroo Traps: Find the discrete logarithm of 3 base 2 modulo 101. For the random mapping use:

$$f(x) \equiv \begin{cases} x + 10 & \text{if } 0 \leq \delta_i < 25 \\ 3x & \text{if } 25 \leq \delta_i < 50 \\ x + 71 & \text{if } 50 \leq \delta_i < 75 \\ 7x & \text{if } 75 \leq \delta_i < 101 \end{cases}$$

$$\Downarrow$$

$$\delta_{i+1} \equiv \begin{cases} 14\delta_i & \text{if } 0 \leq \delta_i < 25 \\ \delta_i^3 & \text{if } 25 \leq \delta_i < 50 \\ 12\delta_i & \text{if } 50 \leq \delta_i < 75 \\ \delta_i^7 & \text{if } 75 \leq \delta_i < 101 \end{cases}$$

Set two traps starting at $u_0 = 0$ and $u_0 = 32$ and step them each 10 steps:

trap 1	trap 2
(95, 20)	(92, 88)

Now set the wild kangaroo free. Start it with two offsets: $\kappa_0 = x$, $\kappa_1 = x + 5$. The wild initial pairs are $(3, x)$ and $(96, x + 5)$. After four steps the second wild kangaroo falls into the second trap:

$$(92, 47x + 45) \sim (92, 88)$$

This generates the equation

$$88 \equiv 47x + 45 \pmod{100}$$

and solving for x gives $x \equiv 69 \pmod{100}$.

3.2 Index Calculus

Index calculus, as described in [8], [17], and [31], is a special purpose discrete logarithm technique, working only in certain types of groups. It is very important however, as it is currently the fastest known technique for finding discrete logarithms over $G = \mathbb{F}_P^*$, where P is a large prime.

Index calculus works best over a group $G = (D/I)^*$, the multiplicative group of the quotient ring, D/I , where D is a Euclidean domain. The two main traits of Euclidean domains used by index calculus are:

Unique Factorization: Every element $s \in D$ can be represented uniquely, up to multiplication by units, as the product of irreducible elements.

Euclidean valuation: An ordering of irreducible elements in D is necessary. The necessary ordering is guaranteed in a Euclidean domain by its Euclid valuation. A Euclidean valuation (chapter 33 in [15] or definition 3.8 in [20]) is a mapping $\nu : D \setminus \{0\} \rightarrow \mathbb{N}$, such that for all $a, b \in D \setminus \{0\}$:

1. $\nu(a) \leq \nu(ab)$;
2. $\exists q, r \in D$ such that $a = bq + r$ with $\begin{cases} r = 0 \\ \text{or} \\ r \neq 0, \nu(r) < \nu(b) \end{cases}$

The ordering used is $b < a$ if and only if $\nu(b) < \nu(a)$.

These traits allow for the concept of *smooth numbers*. Smooth numbers are those which can be factored into irreducible numbers, all less than a given bound, B . Index calculus first creates a factor base of irreducible numbers all less than B . With this factor base the discrete logarithms of smooth numbers can easily be found. All that is needed to compute a discrete logarithm, $dl_\gamma(\alpha)$, where $\langle \alpha \rangle = \mathbb{F}_{p^q}^*$, is to map α into one of these smooth numbers, factor it, and solve the simple equation resulting from that factorization.

The first phase of the algorithm, creation of the factor base, is one-time work but computationally intensive. The run time of the algorithm is mostly based on the run time of the first phase. Algorithms such as the number field sieve [17] or its predecessor the Gaussian integer sieve [8] use fast sieving techniques for the first phase. These sieves generate equations with the discrete logarithm of the

factor base primes as variables. Once enough have been generated the system of equations is solved modulo the order of the group to obtain the discrete logarithm of factor base elements.

The second phase of index calculus computes the actual discrete logarithm, $dl_\gamma(\alpha)$. Assuming α is not already smooth, techniques such as multiplying α by known powers of the base or adding elements of the ideal are used to transform α into a smooth element. Once a smooth form of α has been found it can be factored. Let the factor base be $\{p_i \mid \nu(p_i) < \nu(B)\}$ for some bound $B \in D$. If multiplication of α by γ^c is smooth then:

$$\alpha\gamma^c \equiv \prod_{i=0}^{n-1} p_i^{e_i} \quad (3.2.1)$$

$$dl_\gamma(\alpha) \equiv -c + \sum_{i=0}^{n-1} e_i dl_\gamma(p_i) \quad (3.2.2)$$

Example 3.2.1 – Index Calculus: A slightly larger problem was needed to properly demonstrate index calculus. Let $G = \mathbb{F}_P$ where $P = 251941$. Find the discrete logarithm of $\alpha = 101$, base $\gamma = 11$ (a generator), using a smoothness bound $B = 30$. Because the problem is so small, the first phase of the problem is done by generating random numbers, factoring them and checking for smoothness.

Phase 1: The small primes in the factor base are $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$; the discrete logarithms for them are $\{x_2, x_3, x_5, x_7, x_{13}, x_{17}, x_{19}, x_{23}, x_{29}\}$.

The following smooth numbers were found through trial and error:

$$\begin{aligned}
11^1 &\equiv 11 & 11^{124947} &\equiv 2^2 \times 5 \times 7 \times 11 \\
11^{16003} &\equiv 2 \times 3^4 \times 5^2 & 11^{240664} &\equiv 2^3 \times 3 \times 5 \times 13^{-1} \times 17 \times 23 \\
11^{137993} &\equiv 2^3 \times 11^2 \times 13 & 11^{115186} &\equiv 2^2 \times 3 \times 7 \times 13 \times 19 \\
11^{53937} &\equiv 2^5 \times 3^5 \times 5^{-1} \times 7 & 11^{13487} &\equiv 2^4 \times 3 \times 5^{-1} \times 29 \\
11^{231833} &\equiv 5^4 \times 7 \times 11 & 11^{206010} &\equiv 2^5 \times 3 \times 17 \times 29 \\
11^{57647} &\equiv 2^2 \times 3 \times 5^{-1} \times 7^3 \times 23
\end{aligned}$$

generating the equation set:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 & -1 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 5 & 5 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & -1 & 3 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ x_5 \\ x_7 \\ x_{11} \\ x_{13} \\ x_{17} \\ x_{19} \\ x_{23} \\ x_{29} \end{bmatrix} = \begin{bmatrix} 1 \\ 124947 \\ 16003 \\ 240664 \\ 137993 \\ 115186 \\ 53937 \\ 13487 \\ 231833 \\ 206010 \\ 57647 \end{bmatrix}$$

Solving this set of equations gives us the factor base:

x_2	x_3	x_5	x_7	x_{11}
246875	22086	200212	186804	1
x_{13}	x_{17}	x_{19}	x_{23}	x_{29}
153186	249316	69120	243311	138371

Phase 2: To find an individual discrete logarithm, multiply the value by random powers of the base.

$$101 \times 11^{76431} \equiv 2^2 3^4 5^1 29^1$$

$$dl_\gamma(101) \equiv 2dl_\gamma(2) + 4dl_\gamma(3) + dl_\gamma(5) + dl_\gamma(29) - 76431 \pmod{251940}$$

$$dl_\gamma(101) \equiv 108$$

Chapter 4

Root finding techniques

There are two categories of root finding algorithms on finite cyclic groups. The first and best known exploits the isomorphism between the group and \mathbb{Z}_R , where R is the order of the group. These algorithms can be used on any finite cyclic group, but if q^2 divides the order of the group they require discrete logarithm computation. The second category of algorithms does not require discrete logarithm computation, but operates only on the multiplicative group of a finite field, \mathbb{F}_P , using \mathbb{F}_{P^q} to find the root.

The first category of algorithms, based on the group's isomorphism to \mathbb{Z}_R , are the only known algorithms which operate over any finite cyclic group. Although there are many published algorithms in this category, all of them can be reduced to the same basic computations. Discrete logarithm computations, implicitly or explicitly, are included in these computations whenever q^2 divides the order of the group. A generalized description of these algorithms and an example of a square root algorithm which explicitly computes a discrete logarithm are described in sections 4.1 and 4.2.

The second category of q^{th} root algorithms operate only in the multiplicative group of a finite field and make use of its degree q extension field to compute roots. This is the one category of algorithms which does not require discrete

logarithm computation.

Although algorithms for computing q^{th} roots are described here only for prime q , the homomorphic property of root computation makes it easy to extend them to compute composite roots: $\left((w)^{\frac{1}{b}}\right)^{\frac{1}{a}} = (w)^{\frac{1}{ab}}$.

4.1 Analysis of a square root algorithm

Most published q^{th} root algorithms on a finite cyclic group G , including square root algorithms, are based on the isomorphism between \widehat{G} (section 2.3 on page 24) and $\mathbb{Z}_{ord(G)}$. These algorithms can be found in many sources including [31], [41], [2], [24] and [21]. Often only the square root version of the algorithm ($q = 2$) is described. In these algorithms the discrete logarithm computation is often hidden behind the link between residuosity testing and discrete logarithm computation in subgroups of order 2^n (see section 2.4.1).

The following square root algorithm is taken from [31], page 100, and generalized to work over any finite cyclic group, G . It is similar to Tonelli's algorithm in [3] on page 156, but with one less exponentiation and with the processes in a different order. In this form the algorithm and why it works is easy to explain in relation to its isomorphism to $\mathbb{Z}_{ord(G)}$. This also simplifies the explanation of the extension of the algorithm to $q > 2$.

Algorithm 4.1.1: Square root algorithm found in [31]

Input: $w, b \in G$ such that w is a quadratic residue, b is a quadratic non-residue, with $ord(G) = 2^n s$ where $\gcd(2, s) = 1$

Output: $a \in G$ such that $a^2 = w$

$$1: \quad a_0 = w^{(s+1)/2}$$

$$2: \quad u_0 = b^s$$

3: for $i = 1$ to $n - 1$ do:

$$i: \quad a_i = \begin{cases} a_{i-1}u_{i-1} & \left| \begin{array}{l} (a_{i-1}^2 w^{-1})^{2^{n-1-i}} = -1 \\ \text{otherwise} \end{array} \right. \\ a_{i-1} & \end{cases}$$

ii: Set $u_i = u_{i-1}^2$

4: Return $a = a_{n-1}$.

To understand how this algorithm operates, let $g \in G$ be a generator such that

$$\begin{aligned} b^s &= g^{-s} \\ w &= g^{2^x} \\ x &\equiv \sum_{i=0}^{n-2} 2^i e_i \pmod{2^{n-1}} = dl_{g^{2^s}}(w^s) \quad \text{with } e_i \in \mathbb{Z}_2. \end{aligned}$$

For q^{th} roots with $q > 2$, replace 2 with q in these formulas.

1. Define an approximate root as a root multiplied by a 2^n root of unity, with the error term being that 2^n root of unity. Step 1 computes an approximate root: $a_0 = w^{(s+1)/2} = g^x (g^s)^x$. Notice that $(s+1)/2 \equiv 2^{-1} \pmod{s}$, so this approximation step alone generates a square root for $n < 2$. If $n = 1$, as in the case in \mathbb{F}_P where $P \equiv 3 \pmod{4}$, then $(s+1)/2 = (P+1)/4$. Expanding this inverse formula to $q \geq 2$ gives $\frac{1+s(-s^{-1} \pmod{q})}{q} \equiv q^{-1} \pmod{s}$. Thus a formula to obtain a similar approximation for q^{th} roots is

$$(g^{qx})^{\frac{1+s(-s^{-1} \pmod{q})}{q}} = g^x (g^s)^{x(-s^{-1} \pmod{q})}.$$

2. Each step of the algorithm refines this approximation by removing one bit of the error term, with $a_i = g^x (g^s)^{\sum_{j=i}^{n-2} 2^j e_j}$.
3. Step 3.1 computes e_{i-1} :

$$\begin{aligned} (a_{i-1}^2 w^{-1})^{2^{n-1-i}} &= g^{(2x+2s \sum_{j=i-1}^{n-2} 2^j e_j - 2x)2^{n-1-i}} \\ &= g^{s \sum_{j=i-1}^{n-2} 2^{n-i+j} e_j} \\ &= g^{s 2^{n-1} e_{i-1}} \end{aligned}$$

Therefore if $e_{i-1} = 0$, the result will be 1; if $e_{i-1} = 1$, the result will be -1, and

$$e_{i-1} = \begin{cases} 1 & (a_{i-1}^2 w^{-1})^{2^{n-1-i}} = -1 \\ 0 & \text{otherwise} \end{cases}$$

For $q > 2$, $e_{i-1} \in \mathbb{Z}_q$, and its value can not be determined by this simple test. The difficulty of this discrete logarithm computation is determined by the size of q .

4. The value $u_{i-1} = g^{-s2^{i-1}}$ computed in step 2 and updated in step 3.2 is the inverse of $g^{s2^{i-1}}$. If the value of $e_{i-1} = 1$ then multiplying this value by the current approximation removes e_{i-1} from the exponent of the error term.

Algorithm 4.1.1 is a specialized version of algorithm 4.2.1, with $q = 2$.

4.2 Root finding algorithms based on the order of the group

The square root algorithm 4.1.1, its q^{th} root generalization 4.2.1, and similar algorithms found in [24], [2], [21] are all based on the isomorphism between \widehat{G} and $\mathbb{Z}_{\text{ord}(G)}$. Let $G = \langle g \rangle$ be a finite cyclic group of order R and $q > 1$ a prime integer. Recall that \widehat{G} , derived from G as in section 2.3, is isomorphic to \mathbb{Z}_R . Finding a q^{th} root in G is equivalent to a multiplicative operation in \widehat{G} . The operation in \mathbb{Z}_R isomorphic to finding a q^{th} root in \widehat{G} is division by q . If $q \nmid R$ then $q^{-1} \in \mathbb{Z}_R$ and

$$(w)^{\frac{1}{q}} \equiv w \otimes g^{q^{-1}} \equiv g^{dl_g(w)q^{-1}}$$

for any $w \in G$. When $q \mid R$ however, $q^{-1} \notin \mathbb{Z}_R$. In this case division by q is possible only for elements in the subgroup $q\mathbb{Z}_R \subseteq \mathbb{Z}_R$, and then only by leaving \mathbb{Z}_R and performing the division in the integers. If $q^n \parallel R$ then dividing $x \in (q^i\mathbb{Z}_R) \setminus (q^{i+1}\mathbb{Z}_R)$ by q returns an element in $q^{i-1}\mathbb{Z}_R \setminus q^i\mathbb{Z}_R$ for $1 \leq i < n$,

That is, division of x by q returns an element not included in the smallest ideal $q^i \mathbb{Z}_R$ containing x . It is this ideal jumping division which requires a discrete logarithm computation when the isomorphic operation of computing a q^{th} root in \widehat{G} is performed. This becomes clear when the q^{th} root computation is broken into subgroups of G with co-prime orders.

The square root algorithm in section 4.1 and its generalized q^{th} root algorithm had a trivial portion, the approximation, and a difficult portion, computation of the ‘error term’. By using the CRT isomorphism from section 2.3 the difficult portion is separated from the easy portion.

Let $R = q^n s$ with $n \geq 1$, $\gcd(q, s) = 1$, and G_{q^n}, G_s be the subgroups of G of order q^n and s respectively. Using the CRT, q^{th} roots will be computed in $(\widehat{G}_{q^n} \times \widehat{G}_s)$ instead of \widehat{G} , with $\widehat{\Psi}(w) = (w_{q^n}, w_s)$ and $\widehat{\Psi}(g) = (g_{q^n}, g_s)$ where $G = \langle g \rangle$ (equation (2.3.1)).

The trivial part of the root computation occurs in \widehat{G}_s . Since $q^{-1} \in \mathbb{Z}_s$,

$$(w_s)^{\frac{1}{q}} = w_s \otimes g_s^{q^{-1}} = w_s^{q^{-1}}.$$

The difficult portion occurs in \widehat{G}_{q^n} . Division of $x \in (q\mathbb{Z}_{q^n})$ by q is not defined in \mathbb{Z}_{q^n} even though a $y \in \mathbb{Z}_{q^n}$ such that $qy = x$ exists. A solution is found by performing an operation over \mathbb{Z} . Likewise, a solution to $(w_{q^n})^{\frac{1}{q}}$ is not possible with operations only within \widehat{G}_{q^n} . Operations in an extension field (covered in the following section) or operations in \mathbb{Z} , performing division with a discrete logarithm and exponentiation (as done here) are used in every q^{th} root algorithm currently known.

Let $dl_{g_{q^n}}(w_{q^n}) = x_{q^n}$ with $q^t \parallel x_{q^n}$ for some $0 < t \leq n$, and write x_{q^n} as

$$x_{q^n} = q^t c$$

for some $c \in \mathbb{Z}_{q^{n-t}}^*$. Except for the trivial case ($t = n$, $c = 0$ and $(w_{q^n})^{\frac{1}{q}} =$

$w_{q^n} = 1$), a q^{th} root of w_{q^n} is computed with the following discrete logarithm and exponentiation. Notice the different bases used on the discrete logarithm and exponentiation:

$$c = dl_{g_{q^n}^{q^t}}(w_{q^n})$$

$$(w_{q^n})^{\frac{1}{q}} = \left(g_{q^n}^{q^{t-1}}\right)^c.$$

The order of the two bases differ by a factor of q , producing the division by q necessary in \mathbb{Z} to obtain the q^{th} root. All solutions to $\left(\frac{x_{q^n}}{q}\right)$ in \mathbb{Z}_{q^n} are in $\{q^{t-1}c + kq^{n-1} \mid 0 \leq k < q\}$ so all q^{th} roots of w_{q^n} are in $\left\{g_{q^n}^{q^{t-1}c+kq^{n-1}} \mid 0 \leq k < q\right\}$.

The following algorithm summarizes this form of q^{th} root computation.

Algorithm 4.2.1: Computing a q^{th} root for prime integer q

Input: $w, g_{q^n} \in G$ with $ord(G) = q^n s$, $\gcd(q, s) = 1$, $q | ord(w)$ and $ord(g_{q^n}) = q^n$

Output: $a \in G$ such that $a^q = w$

1: Split w with $\widehat{\Psi}$

$$\widehat{\Psi}(w) = (w^s, w^{q^n}) = (w_{q^n}, w_s)$$

2: Compute the q^{th} root in G_s :

i: Compute $q^{-1} \bmod s$.

ii: Compute $w_s^{1/q} \equiv w_s^{(q^{-1} \bmod s)}$.

3: Compute the q^{th} root in G_{q^n} :

i: Compute the order of w_q : $ord(w_q) = q^{n-t}$.

ii: Compute $c = dl_{g_{q^n}^{q^t}}(w_q)$.

iii: $w_q^{1/q} = g_{q^n}^{q^{t-1}c}$.

4: Combine the results using equation (2.3.3)

$$a = w^{1/q} \equiv (w_s^{1/q})^{q^{-n} \bmod s} \left(w_{q^n}^{1/q} \right)^{s \bmod q^n}.$$

4.3 Cipolla's algorithm

Cipolla's algorithm is the only known technique for solving the q^{th} root problem which does not require discrete logarithm computation. Its original design in [7] computed square roots in a finite field with odd characteristic. Extending this technique to find q^{th} roots, where q is prime, produces a root finding algorithm which works over any finite field, \mathbb{F}_P , in which $q | ord(\mathbb{F}_P^*)$.

Cipolla's algorithm is based on properties of the norm function. Let q, P, K be defined as in table 2.1 (page 20), and $\eta \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$ with minimal polynomial $f(x) = \sum_{i=0}^q c_i x^i$. The two norm equations

$$N(\eta) = \eta^{\frac{P^q-1}{P-1}}$$

$$N(\eta) = (-1)^q c_0.$$

are of particular importance to Cipolla's algorithm. From theorem 2.1.1 we know that $\frac{P^q-1}{P-1} = qK$. Therefore $(\eta^K)^q = (-1)^q c_0$ and $((-1)^q c_0)^{\frac{1}{q}} = \eta^K$.

To compute a q^{th} root of w in \mathbb{F}_P , where $q \mid P-1$, Cipolla's algorithm first finds an irreducible polynomial of degree q over \mathbb{F}_P : $f(x) = \sum_{i=0}^q c_i x^i$ with $c_0 = (-1)^q w$ and $c_q = 1$. This is generally done by randomly generating $f(x)$ with c_0, c_q fixed to their proper values and testing for irreducibility (see [16], [29], and section B.1). Working in $\mathbb{F}_P[x]/(f(x))$, a q^{th} root is computed with $(w)^{\frac{1}{q}} = x^K \pmod{f(x)}$.

Definition 4.3.1 (Cipolla's function): *Let $f \in \mathbb{F}_P[x]$ be an irreducible polynomial of degree q , norm w and root $\eta \in \mathbb{F}_{P^q}$. Then Cipolla's function on f or η is defined as:*

$$\mathfrak{C}(f) = x^K \pmod{f(x)} = \mathfrak{C}(\eta) = \eta^K \quad (4.3.1)$$

Cipolla's algorithm comes directly from the function defined above.

Algorithm 4.3.2: Cipolla's q^{th} root algorithm

Input: $w \in \mathbb{F}_P$ such that $w^{(P-1)/q} = 1$

Output: $(w)^{\frac{1}{q}}$

- 1: Find an monic irreducible polynomial, $f(x) \in \mathbb{F}_P[x]$, of degree q and constant term $(-1)^q w$.

- 2: Let $\eta \in \mathbb{F}_{P^q}$ be a root of $f(x)$.
- 3: Compute $\mathfrak{C}(f) = (w)^{\frac{1}{q}} \equiv \eta^K \equiv x^K \pmod{f(x)}$

While Cipolla's algorithm is straight forward, the computational requirements for any cryptographically sized values are larger than using exhaustive discrete logarithm computation. The average number of multiplications in \mathbb{F}_{P^q} required for this exponentiation using a standard multiply/square exponentiation is $\frac{3}{2} \lg(K)$ ([23], section 4.6.3). Since $K = \frac{P^q - 1}{q(P-1)}$, this number is approximately

$$\text{average multiplies} = ((q - 1) \lg P - \lg q).$$

Furthermore, these cost estimates do not take into consideration the search for an appropriate irreducible polynomial (see section B.1).

For cryptographically sized values of q the number of multiplies in \mathbb{F}_{P^q} is approximately $(q - 1) \lg P$. Using the digital signature algorithm (DSA, page 8 in [11]) as an example, the size of q would be on the order of $159 < \lg q < 160$. In comparison, the number of multiplications in \mathbb{F}_P needed to find the discrete logarithm of an element of order q using a simple exhaustive search is q . If $P = q^n s + 1$, then the maximal number of these discrete logarithms necessary to compute the q^{th} root with algorithm 4.2.1 method is q^{n-1} . For large q Cipolla's algorithm takes on the order of $(\lg P)/(n - 1)$ times the work of algorithm 4.2.1. Since $\lg P > n \lg q > n > n - 1$, this value is greater than one. Therefore algorithm 4.2.1, even using simple exhaustion to find discrete logarithms, is more efficient than Cipolla's algorithm.

A second problem with Cipolla's algorithm is storage. For example, if $P \approx 2^{1024}$ and $q \approx 2^{160}$, a single element in \mathbb{F}_{P^q} requires 2^{170} bits.

4.4 Comparison of required multiplies for Cipolla's and standard q^{th} root algorithms

The computational cost for the q^{th} root algorithm 4.2.1 is dependant on the number of times q divides the order of the group. For example, in \mathbb{F}_P , where $P = q^n s + 1$, requires more multiplies as $n \rightarrow \infty$. If n is increased and s decreased so that P was approximately the same size, the cost of algorithm 4.2.1 would increase. To get an idea of how expensive Cipolla's algorithm is, we examine the worst case for this algorithm and see how large n must be, with respect to q , in order for Cipolla's algorithm to be cost effective.

For odd q let $P = 2q^n + 1$. This form allows for the largest n with respect to P . The number of multiplies over \mathbb{F}_P for algorithm 4.2.1 is:

1. Steps (1) and (2) force the element into the two subgroups and compute the root in the subgroup of order s , relatively prime to q . This initial setup cost will be ignored as it is very small compared to the discrete logarithm computations when q is large.
2. Step (3) computes t , where $(n - t)$ is the actual number of times q divides the order of the element. This value t must be at least one or the element is not a q^{th} residue and no q^{th} root exists. Step (3.ii) computes the discrete logarithm in the subgroup of order q^{n-t} , generally performed by computing $(n - t)$ discrete logarithm computations on elements of order q . We will see that the boundary size for n given q changes little if we use $d\ell m(q) = q$ or $d\ell m(q) = (q)^{\frac{1}{2}}$.

$$\text{number of multiplies} = \frac{3}{2} \lg(q^{n-2-t}) + d\ell m(q)$$

3. The final step (4) removes the 'error' term: $\frac{3}{2} \lg(q^{n-2})$

Therefore the total number of multiplies for this technique (ignoring small constant multiplications) is approximately

$$\begin{aligned}
\text{multiplications} &= \frac{3}{2}(n-2)\lg(q) + \sum_{i=0}^{n-2} \left(\frac{3}{2}\lg(q^{n-2-i}) + dlm(q) \right) \\
&= \frac{3}{2}(n-2)\lg(q) + (n-1)dlm(q) + \frac{3}{2}\lg(q) \sum_{i=0}^{n-2} (n-2-i) \\
&= \frac{3}{2}(n-2)\lg(q) + (n-1)dlm(q) + \frac{3}{4}\lg(q)(n-1)(n-2) \\
&= n^2 \frac{3}{4}\lg(q) + n \left(dlm(q) - \frac{3}{4}\lg(q) \right) - \left(dlm(q) + \frac{3}{2}\lg(q) \right)
\end{aligned} \tag{4.4.1}$$

As $q \rightarrow \infty$, this cost will be approximately $(n-1)dlm(q)$, the cost of $(n-1)$ discrete logarithm computations.

For Cipolla's algorithm, the main cost is in finding the polynomial and the exponentiation. In theorem 6.2.1 we show that there are $\frac{P^q-1}{q(P-1)} - 1$ degree q monic irreducible polynomials with given norm. This implies that approximately $1/q$ of these randomly chosen polynomials are prime, therefore an irreducible polynomial will be found, on average, in $q/2$ tries. The irreducibility test and q^{th} root computation can be combined (see appendix B.1); the number of \mathbb{F}_P multiplies for a multiply in $\mathbb{F}_P[x]/(f(x))$ using a degree q polynomial f is q^2 so the total number of multiplies in \mathbb{F}_P expected for Cipolla's algorithm is:

$$\begin{aligned}
[\# \text{ multiplications}] &= \frac{3}{4}q^3 \lg(K) \\
&= \frac{3}{4}q^3 (\lg(P^q - 1) - n \lg(q) - 1 - \lg(q))
\end{aligned} \tag{4.4.2}$$

To give Cipolla's algorithm the extreme benefit of the doubt, assume we already have the necessary irreducible polynomial and are able to reduce the cost of a multiplication in $\mathbb{F}_P[x]/(f(x))$ to that of a multiplication in \mathbb{F}_P . With this assumption

the number of multiplications in \mathbb{F}_P for Cipolla's algorithm is:

$$\begin{aligned} [\# \text{ multiplications}] &= \frac{3}{2} \lg(K) \\ &= \frac{3}{2} (\lg(P^q - 1) - n \lg(q) - 1 - \lg(q)) \end{aligned}$$

Since $\lg(P^q - 1) \approx \lg((2q^n)^q) = q + nq \lg(q)$, we have the approximate number of multiplies required for Cipolla's algorithm is

$$[\# \text{ multiplications}] \approx n \frac{3}{2} \lg(q) (q - 1) + \frac{3}{2} (q - 1 - \lg(q)) \quad (4.4.3)$$

The dominating factor of this cost is q or q^3 if we account for the real cost of the algorithm.

Given q then we can solve for the smallest n for which Cipolla's algorithm is cost effective by subtracting equation (4.4.3) from (4.4.1) and solving for n . Subtracting equation (4.4.3) from (4.4.1) is $An^2 + Bn + C$ where:

$$\begin{aligned} A &= \frac{3}{4} \lg(q) \\ B &= dlm(q) - \frac{3}{4} \lg(q) (2q - 1) \\ C &= - \left(dlm(q) + \frac{3}{2} (q - 1) \right) \end{aligned}$$

Plugging these values into the quadratic formula gives us the values for n given q in table 4.1. The values for n given $dlm(q) = q$ and $dlm(q) = (q)^{\frac{1}{2}}$ are given in the table. Notice that in order for Cipolla's algorithm to be faster than algorithm 4.2.1, even using exhaustion for the discrete logarithm computation and ignoring much of the computational cost of Cipolla's algorithm, requires that $n > q$ for every odd prime q . If the added computational cost of Cipolla's algorithm is used, the size of n increases significantly. Table 4.2 gives the smallest n for which Cipolla's algorithm requires less multiplications than algorithm 4.2.1 using equation (4.4.2) for the number of multiplications in Cipolla's algorithm.

q	n	
	using an exhaustive search $d\ell m(q) = q$	using an algorithm from section 3.1 $d\ell m(q) = (q)^{\frac{1}{2}}$
3	4	5
5	8	9
7	11	13
11	18	21
13	21	25
23	39	44
101	182	200
1999	3755	3992

Table 4.1: Smallest n for a given q such that Cipolla's algorithm is cost effective assuming: we have an appropriate polynomial and that multiplications in $\mathbb{F}_P[x]/(f(x))$ are equivalent to multiplications in \mathbb{F}_P .

q	n	
	using an exhaustive search $d\ell m(q) = q$	using an algorithm from section 3.1 $d\ell m(q) = (q)^{\frac{1}{2}}$
3	53	54
5	499	500
7	2056	2058
11	13307	13310
13	26361	26364
23	267669	267674
101	10^8	10^8
1999	10^{13}	10^{13}

Table 4.2: Smallest n for a given q such that Cipolla's algorithm is cost effective.

These bounds on n are the reason why Cipolla's algorithm is really only used for computing square roots. For the larger of these small values of q , the size of P at which Cipolla's algorithm is cost effective are larger than any currently used public key prime. In table 4.1, if $q \geq 1999$ then $\lg(P) > 41,000$, and for the more realistic view of Cipolla's algorithm, if $q \geq 11$ then $\lg(P) > 46,000$. For cryptographically secure values of q , with $q > 2^{160}$, the smallest value for n for which Cipolla's algorithm is less expensive than standard methods is on the order of $2q$, forcing $P > 2^{2^{167}}$.

Chapter 5

On the difficulty of the q^{th} root problem

The q^{th} root algorithms described in chapter 4 for groups in which q^2 divided its order, were either more costly than or required a discrete logarithm computation. This chapter explains why q^{th} root computation under these conditions are difficult and proves the equivalence of these two problems. This chapter extends previous work done in [4] with refined definitions, a generalized algorithm description, and a detailed description of problems which arise in a practical implementation of the equivalence proof.

The difficulty of computing q^{th} roots can be isolated in the subgroup of order q^n , where $n > 1$. The remainder of this research will focus on finite cyclic groups of order q^n .

This section will prove that the q^{th} root problem in definition 1.1.1 is, under certain assumptions, equivalent to the discrete logarithm problem. There are two parts to proving this equivalence:

1. Prove that a solution to the discrete logarithm problem generates a solution to the q^{th} root problem;
2. Prove that a solution to the q^{th} root problem generates a solution to the

discrete logarithm problem.

The first part of the equivalence proof is straight forward. Let $G = \langle g \rangle$ be a finite cyclic group with $\text{ord}(g) = q^n$. If $w \in G$ has a q^{th} root and $dl_g(w) = qx$, then $\frac{dl_g(w)}{q} \equiv x \pmod{q^{n-1}}$ and the roots of w are $(w)^{\frac{1}{q}} = g^{x+q^{n-1}i}$ for $0 \leq i < q$. It is not quite as easy to prove that a solution to the q^{th} root problem generates a solution to the discrete logarithm problem.

The first problem with the second half of the proof is to define the q^{th} root function which will enable us to compute a discrete logarithm. This is done using a q^{th} root oracle.

5.1 The q^{th} root oracles

The discrete logarithm function is well defined and therefore it is easy to show that the ability to compute discrete logarithm implies the ability to compute q^{th} roots. However the q^{th} root mapping is not well defined. If $q^n \parallel \text{ord}(G)$, the number of q^{th} root mappings is $q^{q^{n-1}}$ as in definition 5.1.1: each q^{th} residue can be mapped to any one of q roots and there are (q^{n-1}) residues. Furthermore, all known q^{th} root techniques are computationally equivalent or more costly than a discrete logarithm computation, and therefore no known model exists on which to base the oracle.

Definition 5.1.1 (General q^{th} root mapping): Let $G_{q^n} = \langle g \rangle$ be an group of order q^n with $n > 1$. A q^{th} root oracle

$$\Omega: \langle g^q \rangle \rightarrow \langle g \rangle$$

is a mapping from a q^{th} residue to a q^{th} root and is defined as

$$\Omega(g^{qt}) \equiv g^{t+q^{n-1}\delta(t)} \quad (5.1.1)$$

where $0 \leq t < q^{n-1}$ and δ is any function $\delta: \{0, 1, \dots, q^{n-1} - 1\} \rightarrow \mathbb{Z}_q$.

A well defined q^{th} root oracle is needed for the second half of the equivalency proof. All well defined q^{th} root functions are described using equation (5.1.1) and one of the $q^{q^{n-1}}$ δ functions. Since no generalized q^{th} root algorithm in G_{q^n} exists which does not use discrete logarithm computations, the proof which follows modeled the oracle on the type of q^{th} roots returned in known algorithms for groups other than G_{q^n} . These algorithms operate on groups where $q \parallel ord(G)$, or $ord(G) = qs$ where $\gcd(q, s) = 1$. The δ function in these algorithms is linear: $\delta(t) = ct$ for some $c \in \mathbb{Z}_q$. A q^{th} root of g^{qt} can be computed by:

$$\begin{aligned} (g^{qt})^{\frac{1}{q}} &= (g^{qt})^{\frac{1+s(-s^{-1} \bmod q)}{q}} \\ &= g^{t+s\delta(t)} \end{aligned}$$

where $\delta(t) \equiv (-s^{-1})t \bmod q$.

Oracles on cyclic groups of order q^n with linear form of δ are called well-behaved and if they existed, could be used to compute a discrete logarithm in the group.

Definition 5.1.2 (Well-behaved q^{th} root oracle): *A well-behaved q^{th} root oracle is a q^{th} root oracle with a linear function δ :*

$$\delta(t) = ct \bmod q$$

for all $0 \leq t < q^{n-1}$, and where c is a constant.

If a well behaved Ω is used to find the q^{th} root of g^{qt} then

$$\Omega(g^{qt}) \equiv g^{t+q^{n-1}ct} \tag{5.1.2}$$

The number of mappings $\delta: \{0, 1, \dots, q^{n-1} - 1\} \rightarrow \mathbb{Z}_q$ is $q^{q^{n-1}}$. There are only q well-behaved oracles. This implies that the probability of finding a well-behaved

oracle, at random, is $q/q^{q^{n-1}} = q^{1-q^{n-1}}$. If you only needed the mapping for r of the elements instead of all q^{n-1} , or $\delta: \{a_0, a_1, \dots, a_{r-1}\} \rightarrow \mathbb{Z}_q$, the chances of that partial mapping being well-behaved is slightly better: q^{1-r} . In section 5.2 discrete logarithms are taken using $\lg(q)$ calls to a well-behaved oracle. So the probability of getting a well-behaved oracle for a single discrete logarithm computation would be $q^{1-\lg q}$.

5.2 Using the well-behaved q^{th} root oracle to compute a discrete logarithm

This section describes an algorithm to compute discrete logarithms given a well-behaved q^{th} root oracle in a group of order q^n ($n > 1$) using $O(n \lg(q))$ operations. The oracle is used to solve the smaller problem of computing a discrete logarithm in a subgroup of order q .

The larger discrete logarithm problem on an element of order q^n is broken up into n sub-problems, each requiring a discrete logarithm computation in the subgroup of order q . Let $G = \langle g \rangle$ be a group of order q^n , $a \in G$, and the unknown value $dl_g(a) = t$ be in reduced form with $0 \leq t < q^n$:

$$a = g^t.$$

Since $0 \leq t < q^n$, it can be written as $t = \sum_{i=0}^{n-1} t_i q^i$ with $0 \leq t_i < q$. Let $a_j = g^{\sum_{i=j}^{n-1} q^i t_i}$ for $0 \leq j < n$. If a_j is known, then

$$a_j^{q^{n-1-j}} = g^{q^n (\sum_{i=j+1}^{n-1} q^{i-(j+1)} t_i) + q^{n-1} t_j} = g^{q^{n-1} t_j},$$

and $t_j = dl_{g^{q^{n-1}}} (a_j^{q^{n-1-j}})$, where $ord(g^{q^{n-1}}) = q$. If $j < (n-1)$, then a_{j+1} is

$$a_{j+1} = a_j g^{-q^j t_j} = g^{\sum_{i=(j+1)}^{n-1} q^i t_i}.$$

The initial $a_0 = a$, therefore each t_j can be found with a discrete logarithm computation on an element of order q .

Given a well behaved q^{th} root oracle, $\Omega: G_q \rightarrow G_{q^2}$, and generator u of G_{q^2} , the discrete logarithm of $w \in G_q$ base u^q can be computed using $2 \lceil \lg(q) \rceil$ calls to Ω . Let $dl_{u^q}(w) = t$ be the unknown discrete logarithm we wish to compute, and let t in reduced form have known bound R : $0 \leq t < R \leq q$. We will show two calls to the oracle divide this known range in half. After k steps the range is reduced in size to $\lceil \frac{R}{2^k} \rceil$. If the initial bound on t is q , then after $\lceil \lg(q) \rceil$ steps only one value is left in the range and t is fully determined.

Halving the known range of a discrete logarithm is accomplished using a linear function on \mathbb{Z}_{q^2} combined with the linearity of the oracle's δ function. Let $f: \mathbb{Z}_{q^2} \rightarrow \mathbb{Z}_{q^2}$ be a linear function, $\bar{f}: G_{q^2} \rightarrow G_{q^2}$ be defined by $\bar{f}(u^t) = u^{f(t)}$, and represent $f(t)$ in reduced form (i.e., $0 \leq f(t) < q^2$) by $f(t) = s_0 + qs_1$ with $0 \leq s_0, s_1 < q$. Applying $\Omega \cdot \bar{f}$ and $\bar{f} \cdot \Omega$ to u^{qt} give the following results:

$$\begin{aligned} \Omega \cdot \bar{f}(u^{qt}) &= \Omega(u^{q(s_0+qs_1)}) \\ &= u^{s_0+q\delta(s_0)} \end{aligned} \tag{5.2.1}$$

$$\begin{aligned} \bar{f} \cdot \Omega(u^{qt}) &= \bar{f}(u^{t+q\delta(t)}) \\ &= u^{s_0+qs_1+qf \cdot \delta(t)} \end{aligned} \tag{5.2.2}$$

Since δ is linear, $f \cdot \delta(t) = \delta \cdot f(t) = \delta(s_0) + q\delta(s_1)$ and equation (5.2.2) reduces to $\bar{f} \cdot \Omega(u^{qt}) = u^{s_0+qs_1+q\delta(s_0)}$. Equations (5.2.1) and (5.2.2) are equivalent if and only if $s_1 = 0$, or that $f(t)$ in reduced form is less than q . If the known range of the discrete logarithm, t , is $0 \leq t < \frac{q}{R}$ for some integer R , then letting $f(t) = 2Rt$ bounds $f(t)$ to $0 \leq f(t) < 2q$ or that $s_1 \in \{0, 1\}$. Therefore comparing these two equations enables the range of t to be halved:

- $\Omega \cdot \bar{f}(u^{qt}) = \bar{f} \cdot \Omega(u^{qt})$: $0 \leq t < \frac{q}{2R}$ and t is in the lower half of the range;

- $\Omega \cdot \bar{f}(u^{qt}) \neq \bar{f} \cdot \Omega(u^{qt})$: $\frac{q}{2R} \leq t < \frac{q}{R}$ and t is in the upper half of the range.

Let $f_k: \mathbb{Z}_{q^2} \rightarrow \mathbb{Z}_{q^2}$ be defined by $f_k(t) = 2^{k+1}t$. The algorithm uses variables T_k , w_k and t_k to compute $dl_{u^q}(w) = t$, where T_k represents the current known portion of the discrete logarithm and t_k the remaining unknown portion. At step k of the algorithm the values T_k , w_k will be known with $t = T_k + t_k$, $w_k = u^{qt_k}$, and $0 \leq t_k < \lceil \frac{q}{2^k} \rceil$. The algorithm starts with $T_0 = 0$ and $w_0 = w = u^{qt_0}$ with $0 \leq t_0 < q$. At each step equations (5.2.1) and (5.2.2) determine which half of the range t_k is in. If $0 \leq t_k < \lceil \frac{q}{2^{k+1}} \rceil$, then $T_{k+1} = T_k$, $w_{k+1} = w_k$, and all conditions hold. If $\lceil \frac{q}{2^{k+1}} \rceil \leq t_k < \lceil \frac{q}{2^k} \rceil$, the conditions are satisfied by setting $w_{k+1} = w_k u^{-q \lceil \frac{q}{2^{k+1}} \rceil}$, $T_{k+1} = T_k + \lceil \frac{q}{2^{k+1}} \rceil$, and t_{k+1} is forced into the range $0 \leq t_{k+1} < \lceil \frac{q}{2^{k+1}} \rceil$. At $k = \lceil \lg(q) \rceil$ the remaining unknown portion of the discrete logarithm is $0 \leq t_k < 1$ so $dl_{u^q}(w) = T_k$.

Algorithm 5.2.1: Computing discrete logarithms via a root finding oracle

Input: $u \in G$ where $\text{ord}(u) = q^2$, $w = (u^q)^t$, with $0 \leq t < q$

Output: $dl_{u^q}(w)$

1: Let $w_0 = w$, $t_0 = t$, and $T_0 = 0$.

2: For $k = 1$ to $k = \lceil \lg(q) \rceil$:

i: Compute:

$$\begin{aligned} A &= \bar{f}_{k-1} \cdot \Omega(w_{k-1}) = u^{2^k t_{k-1} + q \delta(2^k t_{k-1})} \\ B &= \Omega \cdot \bar{f}_{k-1}(w_{k-1}) = u^{(2^k t_{k-1} \bmod q) + q \delta(2^k t_{k-1})} \end{aligned}$$

ii: Let $d = \begin{cases} 0 & \text{if } A = B \\ \lceil \frac{q}{2^k} \rceil & \text{if } A \neq B \end{cases}$:

$$T_k = T_{k-1} + d$$

$$t_k = t_{k-1} - d$$

$$w_k = w_{k-1} u^{-qd}$$

$$3: \quad dl_{u^q}(w) = T_{\lceil \lg(q) \rceil}.$$

Theorem 5.2.2. *Algorithm 5.2.1 computes the discrete logarithm of w in $2\lceil \lg(q) \rceil$ oracle calls.*

Proof. Let $dl_{u^q}(w) = t$, with initial values of $t_0 = t$, $T_0 = 0$, $w_0 = w$. This implies that $t = t_0 + T_0$. Therefore at iteration $k > 0$,

- $t = t_k + T_k$ since $t_k + T_k = t_{k-1} - d + T_{k-1} + d = t_{k-1} + T_{k-1}$;
- $w_k = u^{-qt_k}$ since $w_k = w_{k-1}u^{-qd} = u^{q(t_{k-1}-d)} = u^{qt_k}$

By induction, $0 \leq t_k < \frac{q}{2^k}$ for all $0 \leq k$:

1. $k = 0$: Since $t_0 = dl_{u^q}(w) \in \mathbb{Z}_q$, in its reduced form $0 \leq t_0 < \frac{q}{2^0}$ and $w_0 = w = u^{qt_0}$.
2. Assume $w_k = u^{qt_k}$ is known with $0 \leq t_k < \frac{q}{2^k}$ for $k = 0, 1, \dots, j$ and define $\bar{f}_j(w_j) = w_j^{2^{j+1}}$. Let

$$A = \bar{f}_j \cdot \Omega(w_j)$$

$$B = \Omega \cdot \bar{f}_j(w_j)$$

By definition $t_{j+1} = t_j - d$ where $d = \begin{cases} 0 & | A = B \\ \lceil \frac{q}{2^{j+1}} \rceil & | A \neq B \end{cases}$. Since $0 \leq 2^{j+1}t_j < 2q$, $2^{j+1}t_j$ can be represented as $2^{j+1}t_j = s_0 + s_1q$ with $0 \leq s_0 < q$ and $s_1 \in \{0, 1\}$. Since Ω is a well behaved oracle,

$$\begin{aligned} A &= \bar{f}_j(u^{t_j+q\delta(t_j)}) = u^{s_0+qs_1+q\delta(s_0)} \\ B &= \Omega((u^q)^{s_0}) = u^{s_0+q\delta(s_0)} \end{aligned}$$

- $A = B \Leftrightarrow s_1 = 0$ which implies $2^{j+1}t_j < q$ and $t_{j+1} = t_j$, therefore $0 \leq t_{j+1} < \frac{q}{2^{j+1}}$.

- $A \neq B \Leftrightarrow s_1 = 1$ which implies $q \leq 2^{j+1}t_j$ and $t_{j+1} = t_j - \lceil \frac{q}{2^{j+1}} \rceil$.

Since $t_j \in \mathbb{Z}$ and $t_j < \frac{q}{2^j}$ this implies that $\lceil \frac{q}{2^{j+1}} \rceil \leq t_j$ and

$$\begin{aligned} 0 \leq t_j - d &< \frac{q}{2^j} - \lceil \frac{q}{2^{j+1}} \rceil \\ 0 \leq t_{j+1} &< \frac{q}{2^{j+1}} \end{aligned}$$

3. Therefore $0 \leq t_k < \frac{q}{2^k}$ holds for $k = 0, 1, \dots$

At $k = \lceil \lg(q) \rceil$, the bound on t_k is $0 \leq t_k < 1$ or that $t_k = 0$. Therefore $dl_u^q(w) = T_k + t_k = T_{\lceil \lg(q) \rceil}$. Each of the $\lceil \lg(q) \rceil$ iterations made two calls to the well-behaved oracle, therefore algorithm 5.2.1 computes the discrete logarithm of w with $2 \lceil \lg(q) \rceil$ well-behaved oracle calls. \square

The probability that this algorithm computes the discrete logarithm using a random q^{th} root oracle is $q^{-\lg(q)}$. Each step of the algorithm makes two calls to the oracle. These two calls must be from the same well behaved oracle, but since there is no comparison of these oracle calls between different steps a different well-behaved oracle can be used at each step. The probability that two calls to a random oracle return well-behaved results is $q^{1-2} = q^{-1}$. This is much lower than the $\frac{1}{2}$ probability of choosing the correct bit at random.

5.3 An example

The following example illustrates how this technique for computing discrete logarithms works. In this example $q = 19$ and $P = 10831$. An oracle is created with the appropriate properties, demonstrated here as a look-up table. Using this look-up table as the oracle, the discrete logarithm of an element is found.

Example 5.3.1 – Discrete logarithm via root finding: Let $P = 10831$, $q = 19$, $u = 7240$ an element of order q^2 . Find the discrete

logarithm of $w = 618$.

1. Let the oracle Ω be defined as $\Omega(u^{qt}) = u^{t+q(5t)}$:

t	u^{qt}	$\Omega(u^{qt})$	t	u^{qt}	$\Omega(u^{qt})$
0	1	1	10	4399	5187
1	7035	5635	11	2798	6707
2	4386	7564	12	4003	4586
3	8822	3155	13	505	10157
4	1140	4754	14	107	7642
5	4960	3727	15	5406	9445
6	6949	336	16	3569	9872
7	5912	8766	17	1657	704
8	10711	7050	18	2839	2894
9	618	9473			

2. The initial range is $0 \leq t_0 < 19$, $w_0 = w$, and $T_0 = 0$. The first calls to the oracle are:

$$\Omega(618)^2 = 2894$$

$$\Omega(618^2) = 2894$$

Since the oracle calls are identical, $t_1 = t_0$, $T_1 = T_0 = 0$, $w_1 = 618$, and $0 \leq t_1 < 19/2$.

3. The second call to the oracle returns:

$$\Omega(618)^4 = 2873$$

$$\Omega(618^4) = 704$$

Since the oracle calls return different answers, $19/4 \leq t_1 < 19/2$.

So since $\lceil \frac{q}{4} \rceil = 5$:

$$\begin{aligned} w_2 &= w_1 u^{-q^5} \\ &= 618 \cdot 7240^{-95} \\ &= 1140 \\ T_2 &= 5 \end{aligned}$$

4. The range is now $0 \leq t_2 < 19/4$. The oracle calls return:

$$\Omega(1140)^8 = 9877$$

$$\Omega(1140^8) = 10175$$

Once again the calls return different answers so the range of t_2 is $19/8 \leq t_2 < 19/4$. So since $\lceil \frac{q}{8} \rceil = 3$:

$$\begin{aligned} w_3 &= w_2 u^{-q^3} \\ &= 1140 \cdot 7240^{-57} \\ &= 7035 \end{aligned}$$

$$T_3 = T_2 + 3 = 8$$

5. The current range is $0 \leq t_3 < 19/8$. The call to the oracle returns the same value:

$$\Omega(7035)^{16} = 9872$$

$$\Omega(7035^{16}) = 9872$$

so the range is $0 \leq t_3 < 19/16$. This implies that $0 \leq t_3 < 2$ is either zero or one. Another iteration would return $t_3 = 1$, but it's easy to see that t_3 is not zero, so it must be one. The final value of $T_4 = T_3 + 1 = 9$.

Chapter 6

Polynomials and elements for Cipolla's algorithm

Leaving general finite cyclic groups, the remainder of this research will focus on finite fields and the analysis of Cipolla's algorithm. Cipolla's algorithm, described in section 4.3, is the only known q^{th} root algorithm independent of the discrete logarithm problem. It only works on the multiplicative groups of finite fields, does not have the 'well-behaved' property needed for algorithm 5.2.1, and has greater computational requirements than the worst discrete logarithm algorithm (i.e., exhaustion). For these reasons, Cipolla's algorithm in its current form neither supports nor weakens the hypothesis that the discrete logarithm and q^{th} root problems are computationally equivalent.

The goal of this research was to determine if modifications to the algorithm exist which either convert it to a well-behaved q^{th} root algorithm or sufficiently reduce the computational requirements. Cipolla's algorithm, described in section 4.3, computes the q^{th} root of $w \in \mathbb{F}_P$ by finding an irreducible polynomial, $f \in \mathbb{F}_P[x]$, of degree q whose norm is w . If $\eta \in \mathbb{F}_{P^q}$ is a root of f then Cipolla's algorithm computes a q^{th} root by exponentiation: $\mathfrak{C}(f) = (w)^{\frac{1}{q}} = \eta^K$ where $K = \frac{(P^q-1)}{q(P-1)}$ (see section 2.1). Exponentiation algorithms are discussed in [23], [5], [1], and a new technique specifically for performing multiplication in \mathbb{F}_{P^q}

is described in section 9.4. However, the size of the exponent is so large that even with the best exponentiation techniques, the algorithm is computationally infeasible. Any modifications to Cipolla's algorithm, either to improve efficiency or to add structure, must be based on how η , or equivalently its minimal polynomial f ,) is chosen. Currently f is found by randomly generating monic degree q polynomials with constant term $(-1)^q w$, and testing for irreducibility (see [29]). Given w , can an intelligent method for choosing f or one of its roots be found such that this q^{th} root function is well behaved or has greatly reduced run time? We show how the size of the exponent can be reduced in section 8.1 using the ground work developed in this chapter.

The analysis of Cipolla's algorithm begins in this chapter by categorizing and enumerating the desired subset of elements in \mathbb{F}_{P^q} , and their associated minimal polynomials, capable of computing q^{th} roots for residues in G_{q^n} .

6.1 Root generating elements and root mapping polynomials

Not all elements in \mathbb{F}_{P^q} will be capable of generating a q^{th} root for q^{th} residues in \mathbb{F}_P . Furthermore, as stated in chapter 5, we are only interested in computing the roots of elements in $G_{q^{n-1}}$, whose roots are in G_{q^n} , where the $G_{q^{n-1}} \subset G_{q^n} \subset \mathbb{F}_P^*$ are the subgroups of order q^{n-1} and q^n respectively. This section defines the elements in \mathbb{F}_{P^q} capable of generating q^{th} roots of elements in $G_{q^{n-1}}$, enumerates them and their corresponding minimal polynomials.

The first restriction on the elements of interest is that they have norm in $G_{q^{n-1}}$. Therefore $ord(\eta^{qK}) \mid q^{n-1}$. If $\eta \in \mathbb{F}_{P^q}$ has order x , then the order of its norm is

$$ord(\eta^{qK}) = \frac{x}{\gcd(x, qK)}.$$

This implies $x | q^n K$ and that these elements are contained in $G_{q^n K} \subset \mathbb{F}_{P^q}^*$.

The second restriction is that the element must generate a q^{th} root. If an element's order divides q^n it will be in the base field \mathbb{F}_P^* and incapable of generating a q^{th} root: since $K \equiv 1 \pmod{q^n}$, raising an element in \mathbb{F}_P to the K power does not change the element. An element in \mathbb{F}_P^* may be a q^{th} root, but it will not generate a new q^{th} root. This leads to a definition of root generating elements.

Definition 6.1.1 (Root generating element/subgroup): *Let q, P, n, K be defined as in table 2.1. A root generating element is an element in \mathbb{F}_{P^q} whose order divides $q^n K$ but does not divide q^n .*

The subgroups $G_{q^n} \subset G_{q^n K} \subset \mathbb{F}_{P^q}^*$ contain all elements with order dividing q^n and $q^n K$ respectively. Therefore the set of all root generating elements is $G_{q^n K} \setminus G_{q^n}$.

The following lemma enumerates the number of root generating elements in \mathbb{F}_{P^q} :

Lemma 6.1.2. *There are $q^n(K - 1)$ root generating elements.*

Proof. This follows directly from the fact that the set of all root generating elements is $G_{q^n K} \setminus G_{q^n}$:

$$\begin{aligned} \text{ord}(G_{q^n K} \setminus G_{q^n}) &= \text{ord}(G_{q^n K}) - \text{ord}(G_{q^n}) \\ &= q^n K - q^n \\ &= q^n(K - 1) \end{aligned} \tag{6.1.1}$$

□

Every root generating element has a minimal polynomial. Cipolla's algorithm finds root generating elements by finding a minimal polynomial whose norm is the appropriate q^{th} residue. Any root of this polynomial generates a q^{th} root of the given residue.

Definition 6.1.3 (Root mapping polynomial): *A root mapping polynomial is the minimal polynomial for a root generating element.*

Notice that if $f(x)$ is a root mapping polynomial then it is an irreducible degree q polynomial over \mathbb{F}_P . If $\eta \in \mathbb{F}_{P^q}$ is a root generating element, we know that $\eta \in G_{q^n K} \setminus G_{q^n}$. Therefore η is not in the base field and has minimal polynomial over \mathbb{F}_P of degree d with $d > 1$ and $d|q$. Since q is prime, this implies that η has minimal polynomial of degree q , and that all root mapping polynomials are irreducible degree q over \mathbb{F}_P .

The following lemma enumerates the root mapping polynomials:

Lemma 6.1.4. *There are $q^{n-1}(K - 1)$ root mapping polynomials.*

Proof. Root mapping polynomials are the polynomials whose roots are root generating elements. From equation (6.1.1) we know there are $q^n(K - 1)$ root generating elements. There are q conjugates for each root mapping polynomial, so there are:

$$\frac{q^n(K - 1)}{q} = q^{n-1}(K - 1) \tag{6.1.2}$$

root mapping polynomials. □

6.2 Enumeration of Polynomials based on the norm of its roots

Lemma 6.1.4 revealed that there were $q^{n-1}(K - 1)$ root mapping polynomials which might be used in Cipolla's algorithm to compute q^{th} roots in the subgroup of interest. Since the algorithm needs a polynomial whose norm is a particular q^{th} residue, the next enumeration question is: How many root mapping polynomials have norm $w \in G_{q^{n-1}}$? This section will prove that the root mapping polynomials are evenly distributed among the q^{n-1} possible q^{th} residues.

The following theorem enumerates polynomials with any fixed norm in \mathbb{F}_P^* , not just for those with norms in $G_{q^{n-1}}$.

Theorem 6.2.1. *The number of degree q monic irreducible polynomials over \mathbb{F}_P with norm (with respect to \mathbb{F}_{P^q}) $a \in \mathbb{F}_P^*$ is*

$$\tau(a) = \begin{cases} K & \text{if } a \text{ is a } q^{\text{th}} \text{ non-residue} \\ (K - 1) & \text{if } a \text{ is a } q^{\text{th}} \text{ residue} \end{cases} \quad (6.2.1)$$

Proof. Let $\gamma \in \mathbb{F}_{P^q}$ be a generator with $N(\gamma) = g$. The order of \mathbb{F}_{P^q} is $P^q - 1 = q^{n+1}sK$ (theorem 2.1.1), therefore every element in \mathbb{F}_{P^q} can be represented by γ^x for some $x \in \mathbb{Z}$ with $0 \leq x < q^{n+1}sK$. For every $0 \leq x < q^{n+1}sK$, let $x_1 = \left\lfloor \frac{x}{q^n s} \right\rfloor$ and $x_0 = x - x_1(q^n s)$: $x = x_0 + x_1 q^n s$ with $0 \leq x_1 < \left(\frac{q^{n+1}sK}{q^n s}\right)$ and $0 \leq x_0 < q^n s$. This mapping is bijection from the set $\{0, 1, \dots, q^{n+1}sK - 1\}$ to the set $\{\{0, 1, \dots, q^n s - 1\} \times \{0, 1, \dots, qK - 1\}\}$.

With this representation, we have $N(\gamma^x) = \gamma^{x_0 q K} = g^{x_0}$. Since $0 \leq x_0 < q^n s$, the norm of γ^x is uniquely determined by x_0 . There are qK values of x_1 for every fixed value of x_0 , therefore there are qK elements in \mathbb{F}_{P^q} with norm g^{x_0} .

An element $\gamma^x \in \mathbb{F}_P$ if and only if $x \equiv 0 \pmod{qK}$, therefore $\gamma^x \in \mathbb{F}_P$ if and only if $x_0 \equiv 0 \pmod{q}$ and $x_1 \equiv -x_0 (q^n s)^{-1} \pmod{K}$. Since $0 \leq x_1 < qK$, there are $\frac{qK}{K} = q$ base field elements with norm g^{x_0} . Therefore the number of elements in the base field with norm g^{x_0} is zero if $q \nmid x_0$ and q if $q \mid x_0$. Therefore the number of elements in $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$ with norm g^{x_0} is qK if $q \nmid x_0$ and $q(K - 1)$ if $q \mid x_0$. Since q is prime, all of these elements have minimal polynomials of degree q , therefore

$$\begin{aligned} \tau(a) &= \begin{cases} \frac{qK}{q} & \text{if } a \text{ is a } q^{\text{th}} \text{ non-residue} \\ \frac{q(K-1)}{q} & \text{if } a \text{ is a } q^{\text{th}} \text{ residue} \end{cases} \\ &= \begin{cases} K & \text{if } a \text{ is a } q^{\text{th}} \text{ non-residue} \\ (K - 1) & \text{if } a \text{ is a } q^{\text{th}} \text{ residue} \end{cases} \end{aligned}$$

□

Chapter 7

Breaking down Cipolla's Algorithm

Modifications to Cipolla's algorithm, reducing the required computation or making it well-behaved (definition 5.1.2), rely on how root generating elements are chosen. Chapter 6 enumerated the root generating elements. This chapter dissects their structure, both for theoretical and computational purposes.

The discrete logarithm isomorphism, θ^{-1} discussed in section 2.3, mapped \widehat{G} to $\mathbb{Z}_{\text{ord}(G)}$. Using this isomorphism on the subgroup $\widehat{G}_{q^n K} \subset \widehat{\mathbb{F}_{P^q}^*}$, root generating elements can be represented by an integer $x \in \mathbb{Z}_{q^n K}$. The first dissection of root generating elements in this chapter breaks x into a triplet (t, a, b) such that:

- t defines the q^{th} residue for which a root is desired;
- a defines which of the q roots the element will return;
- b defines all possible elements in $G_{q^n K}$ which return this particular root.

The nature of a set of root generating elements, whether they are well-behaved or not, will be easily observed in this triplet representation. Furthermore, computation of a particular root generating element or exhaustion over any given subset (for small P, q) is simplified by it.

The second dissection uses the CRT isomorphism $\widehat{\Psi}$, from section 2.3, to divide a root generating element into two parts. Root generating elements are a subset

of $G_{q^n K}$, therefore $\widehat{\Psi} : \widehat{G_{q^n K}} \rightarrow (\widehat{G_{q^n}} \times \widehat{G_K})$ is used. Given a root mapping polynomial f with root $\eta \in G_{q^n K}$ and norm w , let $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$. In this chapter we show that $\mathfrak{C}(f) = \eta_{q^n}$, the q^{th} root returned with Cipolla's algorithm and f . Exponentiation by K erases η_K , leaving η_{q^n} unchanged.

From this second dissection of root generating elements comes a partitioning of root mapping polynomials into equivalence classes. Each equivalence class contains exactly one polynomial for each q^{th} root of every element $w \in G_{q^n-1}$. Any reduced computation version of Cipolla's algorithm, as in chapter 8, requires polynomials to be found from a small collection of these equivalence classes: classes with small order (definition 7.3.4). Each equivalence class is determined by a representative polynomial (definition 7.3.1). In section 7.4 we show that if we have a root mapping polynomial f in an equivalence class S and can determine the representative polynomial for S , then the q^{th} root can be determined with a few operations in \mathbb{F}_P instead of a large exponentiation in \mathbb{F}_{P^q} . For a random root mapping polynomial the easiest way to compute the representative polynomial appears to require extracting η_{q^n} from its root η , or computing the q^{th} root using Cipolla's algorithm. However for the equivalence classes with the smallest possible order the representative polynomials are easy to determine. This will be shown in chapter 8.

7.1 Exponential structure of Cipolla's root finding algorithm

This section examines the structure of root generating elements using the isomorphism $dl_\mu(\eta)$, where μ is defined in table 7.1 and $\eta \in G_{q^n K}$ is a root generating element. Since every root generating element has order dividing $q^n K$, every root generating element $\eta \in \mathbb{F}_{P^q}$ can be represented as $\eta = \mu^x$, or $dl_\mu(\eta) = x$, for some

q, P, n, s, K	as in table 2.1
μ	$\mu \in \mathbb{F}_{P^q}$, an element of order $q^n K$;
u	$u \in \mathbb{F}_P$, an element of order q^n with $u \equiv \mu^K$;
h	$h \equiv g^{sq^{n-1}}$, a q^{th} root of unity in \mathbb{F}_P ;
w	$w \equiv u^{qt}$ a q^{th} residue in \mathbb{F}_P ;

Table 7.1: Variables used for analysis of Cipolla's algorithm

$x \in \mathbb{Z}_{q^n K}$. The purpose of the following theorem will be to uniquely represent every $x \in \mathbb{Z}_{q^n K}$, and therefore every $\eta \in G_{q^n K}$, as a triplet (t, a, b) where

1. t determines the q^{th} residue in $G_{q^{n-1}}$;
2. a determines the root;
3. b determines elements in $G_{q^n K}$ which generate the same root for a given residue,

and to define a function \mathcal{E} mapping these triplets back to $x \in \mathbb{Z}_{q^n K}$.

With \mathcal{E} and μ we can compute root generating elements for any root/residue pair desired: for a desired root/residue pair (t, a) pick a random b and compute $\mu^{\mathcal{E}(t,a,b)}$. The root mapping polynomial associated with the root generating element (i.e., its minimal polynomial) can be also be computed (see chapter 3.3 in [29]). Triplets which generate base field elements and conjugates can be computed (section 7.1.1) and eliminated from the computation. For smaller P, q values this function simplifies exhaustion over all root mapping polynomials. For larger cases, selective samples of various size and form can be generated. Besides using the function for computation, it also gives an alternative enumeration for root generating elements and root mapping polynomials.

Lemma 7.1.1. *Let q, P, n, K be defined as in table 2.1 and*

$$\mathcal{E}: (\{0, 1, \dots, q^{n-1} - 1\} \times \mathbb{Z}_q \times \mathbb{Z}_K) \rightarrow \mathbb{Z}_{q^n K}$$

be defined as:

$$\mathcal{E}(t, a, b) \equiv t + aq^{n-1}K + bq^n \pmod{q^n K}. \quad (7.1.1)$$

Then \mathcal{E} is a bijection.

Proof. Define $\mathcal{E}^{-1}: \mathbb{Z}_{q^n K} \rightarrow (\{0, 1, \dots, (q^{n-1} - 1)\} \times \mathbb{Z}_q \times \mathbb{Z}_K)$ by

$$\mathcal{E}^{-1}(x) = \left(t, \frac{x-t}{q^{n-1}} \pmod{q}, (x-t)q^{-n} \pmod{K} \right) \quad (7.1.2)$$

where $t \equiv x \pmod{q^{n-1}}$ and $0 \leq t < q^{n-1}$. Then \mathcal{E}^{-1} is the inverse of \mathcal{E} :

$$\begin{aligned} \mathcal{E}(\mathcal{E}^{-1}(x)) &= \mathcal{E}\left(\left(t, \frac{x-t}{q^{n-1}} \pmod{q}, (x-t)q^{-n} \pmod{K}\right)\right) \\ &= t + \left(\frac{x-t}{q^{n-1}} \pmod{q}\right)q^{n-1}K + ((x-t)q^{-n} \pmod{K})q^n \\ &= \begin{cases} t + \frac{x-t}{q^{n-1}}q^{n-1} \pmod{q^n} \\ t + ((x-t)q^{-n})q^n \pmod{K} \end{cases} \\ &= \begin{cases} x \pmod{q^n} \\ x \pmod{K} \end{cases} \end{aligned}$$

and since $\gcd(q, K) = 1$, this uniquely determines $x \in \mathbb{Z}_{q^n K}$. Since an inverse of \mathcal{E} exists, \mathcal{E} is a bijection. \square

Equation (7.1.1) tells us exactly how root generating elements map to elements in \mathbb{F}_P in terms of residue and root. If $\eta = \mu^{\mathcal{E}(t,a,b)}$ is a root generating element, then

$$\begin{aligned} dl_\mu(N(\eta)) &= (t + aq^{n-1}K + bq^n)qK \pmod{q^n K} \\ &= qKt \pmod{q^n K} \\ &\equiv qt \pmod{q^n} \\ dl_\mu(\mathfrak{C}(\eta)) &= tK + aq^{n-1}K^2 \pmod{q^n K} \\ &\equiv t + aq^{n-1} \pmod{q^n} \end{aligned}$$

Therefore, $N(\eta) = u^{qt}$ and $\mathfrak{C}(\eta) = u^{t+aq^{n-1}}$, where $u = \mu^K$ as defined in table 7.1. The (t, a) -values of the triplet represent the q^{th} residue and root for Cipolla's

algorithm, while the b -value dictates which root generating element is used for the given root/residue pair.

Besides containing all root generating elements, the subgroup $G_{q^n K}$ also contains a few non-root generating elements (the elements in G_{q^n}), as well as redundant elements (conjugates). The next section defines formulas for these non-root generating elements and the redundant conjugates.

7.1.1 Formulas for base field elements and conjugates

This section determines which triplets generate base field elements and conjugates. Base field elements can not be used in Cipolla's algorithm and if equation (7.1.1) is used to find root mapping polynomials, the conjugates should be avoided to reduce computational costs.

The mapping defined by lemma 7.1.1 generates all elements in $\mathbb{Z}_{q^n K}$, including those in its subring $K\mathbb{Z}_{q^n K}$ of order q^n . The subring $K\mathbb{Z}_{q^n K}$ is isomorphic to \mathbb{Z}_{q^n} with $K\mathbb{Z}_{q^n K} = \{Kx \mid x \in \mathbb{Z}_{q^n}\}$. Therefore $\mu^{Kx} \equiv u^x \in G_{q^n}$ is a base field element. The triplets, (t, a, b) , which generate the base field elements are:

$$\begin{aligned} \mathcal{E}(t, a, b) &\equiv 0 \pmod{K} \\ t + aq^{n-1}K + bq^n &\equiv 0 \pmod{K} \\ b &\equiv -tq^{-n} \end{aligned} \tag{7.1.3}$$

and

$$G_{q^n} = \{\mu^x \mid x = \mathcal{E}(t, a, -tq^{-n}), 0 \leq t < q^{n-1}, a \in \mathbb{Z}_q\}. \tag{7.1.4}$$

A root generating element has q conjugates, including itself, and each conjugate has the same root mapping polynomial. When equation (7.1.1) is used to generate root mapping polynomials, only one of its roots is needed. Recall that

the conjugates of an element $\alpha \in \mathbb{F}_{P^q}$ are α^{P^i} for $i = 0, 1, \dots, q - 1$. Conjugate elements have the same minimal polynomial, norm, and generate the same q^{th} root. Therefore if $\alpha = \mu^{\mathcal{E}(t,a,b)}$, then

$$\alpha^{P^i} = \mu^{\mathcal{E}(t,a,b^{(i)})}$$

for some $b^{(i)} \in \mathbb{Z}_K$. This implies that $\mathcal{E}(t, a, b) P^i \equiv \mathcal{E}(t, a, b^{(i)}) \pmod{q^n K}$. Since the $b^{(i)}$ portion of the equation vanishes modulo q^n , we only need to look at this equation modulo K . Solving for $b^{(i)} \pmod{K}$ gives:

$$b^{(i)} \equiv t \frac{(P^i - 1)}{q^n} + b P^i \pmod{K}. \quad (7.1.5)$$

For computational purposes, the following recursive version of this formula may be more suitable. Recall from section 2.1 that $q^n s = (P - 1)$ and that $\frac{P^i - 1}{P - 1} = \sum_{j=0}^{i-1} P^j$. Replacing $\frac{1}{q^n}$ with $\frac{s}{(P-1)}$ gives:

$$\begin{aligned} b^{(i)} &\equiv t s \sum_{j=0}^{i-1} P^j + b P^i \pmod{K} \\ &\equiv t s + P \left(t s \sum_{j=0}^{i-2} P^j + b P^{i-1} \right) \pmod{K} \\ &\equiv t s + P b^{(i-1)} \pmod{K}. \end{aligned} \quad (7.1.6)$$

7.1.2 An alternate enumeration

Lemma 7.1.1 also gives an enumeration of root mapping polynomials returning a specified q^{th} root. The (t, a) pair in the triplet representation (t, a, b) dictate the q^{th} residue (u^{qt}) and root resulting from applying Cipolla's algorithm ($\mathfrak{C}(\mu^{\mathcal{E}(t,a,b)}) = u^{t+aq^{n-1}}$). Two root mapping polynomials generate the same q^{th} root with if and only if they have the same (t, a) pair. Since there are $(K - 1)$ valid, non-base generating values of b which can be matched with (t, a) to form a root generating element, dividing out the q conjugates gives us that there are $\frac{(K-1)}{q}$ root mapping polynomials returning a specified q^{th} root with Cipolla's algorithm.

7.2 Similar root mapping polynomials

Section 7.1 analyzed the set of all root mapping polynomials based on an equation in $\mathbb{Z}_{q^n K}$. In this section the root mapping polynomials will be analyzed using the CRT isomorphism $\widehat{\Psi}$ from section 2.3. Root generating elements lie in the subgroup of order $q^n K$, with q^n and K relatively prime, therefore the root generating elements can be looked at as elements of $(\widehat{G}_{q^n} \times \widehat{G}_K)$.

Let $\eta \in G_{q^n K}$ be a root generating element. The mapping $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$ is defined as:

$$\begin{aligned}\eta_{q^n} &= \eta^K \\ \eta_K &= \eta^{q^n}.\end{aligned}$$

If the minimal polynomial for η is f then $\mathfrak{C}(f) = \eta_{q^n}$, the portion of η contained in G_{q^n} . Definition 6.1.1 states that the order of a root generating element divides $q^n K$ but does not divide q^n . This implies that $\eta_K \neq 1$. A non-trivial η_K implies that the minimal polynomial for η has degree q .

Exponentiation by K erases η_K without changing η_{q^n} because $\text{ord}(\eta_K) \mid K$ and $K \equiv 1 \pmod{q^n}$. Since the computational requirements of Cipolla's algorithm are determined by the size of K , if η can be found such that the order of η_K is very small, these requirements could be reduced. This is explained in more detail in section 8.1.

This section defines a partitioning of the root mapping polynomials such that polynomials in the same class have roots with equivalent η_K portions. Reducing the computational requirements of Cipolla's algorithm implies finding a polynomial (or one of its roots) in a root mapping polynomial equivalence class with a small ordered η_K portion.

Definition 7.2.1 (Similar root mapping polynomials): *Two root*

mapping polynomials, $f(x), h(x)$ are similar ($f(x) \sim h(x)$) if $f(x) = \sum_{i=0}^q c_i x^i$ and there exists $a \in \mathbb{F}_P$ such that

$$h(x) = a^q f(a^{-1}x) = \sum_{i=0}^q a^{q-i} c_i x^i. \quad (7.2.1)$$

Let $f(x)$ be a root mapping polynomial with root η . The following lemma shows that the set of all similar root mapping polynomials for f is exactly the set

$$\{f_a(x) = a^q f(a^{-1}x) \mid a \in G_{q^n}\}$$

with $f_a(\eta a) = 0$. In other words, similarity is the action of the group G_{q^n} on the set of all root mapping polynomials.

Lemma 7.2.2 (Set of similar root mapping polynomials). *Let f be a root mapping polynomial with root $\eta \in G_{q^n K}$. The set*

$$S = \{f_a(x) = a^q f(a^{-1}x) \mid a \in G_{q^n}\} \quad (7.2.2)$$

is the set of all root mapping polynomials similar to f , with $f_a(\eta a) = 0$.

Proof. Let the root mapping polynomial $f(x) = \sum_{i=0}^q c_i x^i$ define S , as in equation (7.2.2), with root η . Since $G_{q^n} \subset \mathbb{F}_P^*$, definition 7.2.1 gives us that the subset of root mapping polynomials in S are similar to f . From definitions 6.1.3 and 6.1.1, f_a is a root mapping polynomial if its root α has order $ord(\alpha) \mid q^n K$ and $ord(\alpha) \nmid q^n$. For all $a \in G_{q^n}$,

$$\begin{aligned} f_a(\eta a) &= a^q \sum_{i=0}^q c_i a^{-i} (\eta a)^i \\ &= a^q f(\eta) \\ &= 0. \end{aligned} \quad (7.2.3)$$

Therefore $f_a(\eta a) = 0 \forall f_a \in S$. Since $a \in G_{q^n}$ we know that $(\eta a)^{q^n} = \eta^{q^n} = \eta_K \neq 1$ and $(\eta a)^{Kq^n} = 1$ which implies $ord(\eta a) \mid q^n K$ and $ord(\eta a) \nmid q^n$. Therefore

ηa is a root generating element, f_a is a root mapping polynomial, and by definition $f(x) \sim f_a(x) \forall f_a \in S$.

Assume $h \sim f$. Then h is a root mapping polynomial and $\exists a \in \mathbb{F}_P$ such that $h(x) = a^q f(a^{-1}x)$. Furthermore, $h(\eta a) = a^q f(\eta) = 0$. Since h is a root mapping polynomial, its root ηa is a root generating element and $(\eta a)^{Kq^n} = 1$. This implies $a^{q^n} = 1$ and that $a \in G_{q^n}$. Therefore $h \in S \forall h \sim f$. Since $h \sim f \forall h \in S$ and $h \in S \forall h \sim f$, we know that S is the set of all similar root mapping polynomials to f . \square

Since similarity is the action of the group G_{q^n} on the set of all root mapping polynomials, it partitions the root mapping polynomials and is an equivalence relation (see theorem 4.2 in [20]). If the polynomials f, h are in an equivalence class S defined by root mapping polynomial similarity, with $f(\eta) = h(\eta a) = 0$ and $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$, then

$$\begin{aligned} \widehat{\Psi}(\eta a) &= \left((\eta a)^K, (\eta a)^{q^n} \right) \\ &= (\eta_{q^n} a, \eta_K). \end{aligned} \tag{7.2.4}$$

The multiplier a used to derive h from f can also be used to convert the q^{th} root generated by f to the q^{th} root generated by h . In other words $\mathfrak{C}(h) = a\mathfrak{C}(f)$.

Lemma 7.2.3. *Let f, h be two root mapping polynomials with $f \sim h$ and $h(x) = a^q f(a^{-1}x)$, $a \in G_{q^n}$. Then:*

$$\mathfrak{C}(h) = a\mathfrak{C}(f) \tag{7.2.5}$$

Proof. Let η be a root of $f(x)$. Then from lemma 7.2.2 we know that $a\eta$ is a root of $h(x)$. Since $a \in \mathbb{F}_P$ and $K \equiv 1 \pmod{P-1}$ we know that

$$\begin{aligned} \mathfrak{C}(h) &= (a\eta)^K \\ &= a\eta^K \\ &= a\mathfrak{C}(f). \end{aligned}$$

□

7.3 Representation and order of a root mapping polynomial equivalence class

Root mapping polynomial similarity partitions the set of root mapping polynomials into equivalence classes. One polynomial in each equivalence class will be used to represent the entire set. This representative polynomial is chosen to simplify the set description. Root generating elements with $\eta_{q^n} = 1$ generate the trivial q^{th} root of $(1)^{\frac{1}{q}} = 1$. This is the logical choice for the representative polynomial.

Definition 7.3.1 (Representative polynomial for a root mapping polynomial equivalence class): *Let S be an equivalence class of similar root mapping polynomials. A representative polynomial for S is a polynomial $r(x) \in S$ such that if $\alpha \in \mathbb{F}_{P^q}$ is a root of $r(x)$ then*

$$\alpha^K \equiv 1.$$

With this definition, Cipolla's algorithm performed on the representative polynomial is $\mathfrak{C}(r) = 1$. If $f_a = a^q r(a^{-1}x)$, as in equation (7.2.2), then from lemma 7.2.3:

$$\mathfrak{C}(f_a) = a.$$

Every equivalence class contains exactly one representative polynomial as well as exactly one polynomial, $f_a(x)$, for each $a \in G_{q^n}$. This is shown in the following lemma

Lemma 7.3.2. *Let S be an equivalence class of similar root mapping polynomials and $\mathfrak{C}(f)$ be the q^{th} root generated using Cipolla's algorithm with root mapping*

polynomial $f(x)$. Then

$$\mathfrak{C}(S) = \{\mathfrak{C}(f) \mid f \in S\} = G_{q^n}.$$

Proof. Let S be a root mapping polynomial equivalence class with $r(x) = \sum_{i=0}^q c_i x^i$, $r(x) \in S$, and $\mathfrak{C}(r) = a$. By definition

$$S = \{b^q r(b^{-1}x) \mid b \in G_{q^n}\},$$

and $\mathfrak{C}(f) \in G_{q^n} \forall f \in S$. For all $b \in G_{q^n}$ let $f_b(x) = (a^{-1}b)^q r((a^{-1}b)^{-1}x)$. Then $f_b(x) \in S$ and

$$\begin{aligned} \mathfrak{C}(f_b) &= (a^{-1}b) \mathfrak{C}(r) \\ &= (a^{-1}b) a \\ &= b. \end{aligned}$$

Therefore $\mathfrak{C}(S) = G_{q^n}$. □

From this lemma the size and number of equivalence classes is easily determined.

Lemma 7.3.3. *Let q, P, n, K be defined as in table 2.1 and S be an equivalence class of similar root mapping polynomials for computing q^{th} roots in \mathbb{F}_P . Then*

- $\text{ord}(S) = q^n$ and
- the number of equivalence classes is $\frac{(K-1)}{q}$.

Proof. By lemma 7.2.2, $\text{ord}(S) \leq q^n$ and from lemma 7.3.2, $\mathfrak{C}(S) = G_{q^n}$ so $\text{ord}(S) \geq q^n$. Therefore $\text{ord}(S) = q^n$.

There are $(K - 1)$ root mapping polynomials for each q^{th} residue in G_{q^n} (theorem 6.2.1) and the set of q^{th} residues in G_{q^n} is $G_{q^{n-1}}$. Therefore there are

$q^{n-1}(K - 1)$ root mapping polynomials. Dividing by the number of polynomials in each class gives us the number of similar root mapping polynomial equivalence classes:

$$\frac{q^{n-1}(K - 1)}{q^n} = \frac{(K - 1)}{q}.$$

□

As mentioned in the beginning of section 7.2, the computational requirements of Cipolla's algorithm can be reduced if a root generating element η can be found with low order η_K components. Root generating elements are found by generating root mapping polynomials and all polynomials in an equivalence class have roots with identical η_K component. Finding a root generating element with a low computational requirements implies finding a polynomial in an equivalence class with low order η_K .

Definition 7.3.4 (Order of a root mapping polynomial equivalence class): *The order of a root mapping polynomial equivalence class S is the order of the representative polynomial $r(x) \in S$.*

Equivalence class orders have several restrictions. The following lemma defines these restrictions and can be used to generate finite fields such that a root mapping polynomial equivalence class of a specific order R exists. Although of little value in solving the general q^{th} root problem, it is useful to generate these fields for analysis and they may have future value in designing new public key cryptosystems.

Lemma 7.3.5 (Properties of root mapping polynomial equivalence class orders). *Let q, P be defined as in table 2.1 and R be the order of a root mapping polynomial equivalence class for computing q^{th} roots in \mathbb{F}_P . Then:*

- $R = 2qt + 1$ for some $t \in \mathbb{N}$
- $P \not\equiv 1 \pmod{R}$
- $P^q \equiv 1 \pmod{R}$.

Proof. From definition 7.3.4, $R = \text{ord}(r)$ where r is the representative polynomial for the class. All representative polynomials have $\text{ord}(r) \mid K$ with $\text{ord}(r) > 1$, therefore $R \mid K$. Lemma 2.1.3 gives us that $R \equiv 1 \pmod{q}$. Since $q > 2$, $2 \not\equiv 1 \pmod{q}$ therefore $2 \nmid K$ and $2 \nmid R$. Therefore $R = 2qt + 1$ for some $t \in \mathbb{N}$. From 2.1.1 we know that $\gcd(K, (P - 1)) = 1$, therefore $\gcd(R, (P - 1)) = 1$ and $P \not\equiv 1 \pmod{R}$. Finally, $R \mid K \mid (P^q - 1)$ which implies $P^q \equiv 1 \pmod{R}$. \square

Any prime P with equivalence class of order R (where $R \equiv 1 \pmod{2q}$) must satisfy the following properties:

- $P \equiv 1 \pmod{q^2}$: this makes computation of q^{th} roots in \mathbb{F}_P difficult;
- $P \equiv h \pmod{R}$ where h is a primitive q^{th} root of unity: this insures that $P^q \equiv 1 \pmod{R}$ and $P \not\equiv 1 \pmod{R}$.

The Chinese remainder theorem can be used to generate possible P values, which can then be tested for primality.

The number of equivalence classes of order R is equal to the number of representative polynomials of order R . If $R = 2qt + 1$, there are $\phi(R)$ elements in \mathbb{F}_{P^q} with order R , all with representative polynomials as their minimal polynomials. Therefore there are $\frac{\phi(R)}{q}$ representative polynomials and equivalence classes with order R .

Reducing the computation in Cipolla's algorithm hinges on finding a root mapping polynomial equivalence class with small order. Therefore an important

question to ask is ‘how small, with respect to q , can the order of a root mapping polynomial equivalence class be?’ If R is represented as $R = 2qt + 1$ then the smallest value possible is $R = 2q + 1$ ($R > 1$ from the root generating element requirements). This smallest possible R value exists only for certain (q, P) pairs.

Definition 7.3.6 (Minimal order equivalence class): *Let q, P be defined as in table 2.1 and $R = 2qt + 1$ be the order of a root mapping polynomial equivalence class S for computing q^{th} roots in \mathbb{F}_P . The equivalence class S has minimal order if $t = 1$.*

The minimal order equivalence class, by this definition, is $R = 2q + 1$. This equivalence class exists if and only if:

- R is prime;
- $P \not\equiv 1 \pmod R$
- $P^q \equiv 1 \pmod R$.

Lemma 7.3.5 explains all of the requirements except primality. The order of a minimal order equivalence class must be prime otherwise other equivalence classes would exist with orders equivalent to its non-trivial factors. These classes would have smaller order, and the original would not be minimal. Examples of (q, P) pairs for which minimal order equivalence classes exist are in table A.2.

In general, polynomials in a given equivalence class are difficult to find. For this minimal order case however, the representative polynomial is easy to generate. The formula for the polynomials trace function is given in 8.3, with formulas for the remaining coefficients developed in section 8.4.

We show in section 7.4 that q^{th} roots can be computed with a few operations in \mathbb{F}_P if the root mapping polynomial and the representative polynomial for its

equivalence class are known. Finding a polynomial for any given norm in these minimal equivalence classes minimizes the work required in Cipolla's algorithm, however it also makes Cipolla's algorithm, even with the reduced computation, irrelevant. It is much more efficient to extract the q^{th} root from the coefficients of the two polynomials than to compute $\mathfrak{C}(f)$.

7.4 Knowledge of set representative implies knowledge of q^{th} roots

Reduced computation versions of Cipolla's algorithm require polynomials in low order equivalence classes to be found. If a reduced computation version of Cipolla's algorithm exists and the representative polynomial for that equivalence class can also be found, then the trace of the two polynomials will reveal the q^{th} root.

Assume we are trying to find the q^{th} root of $w \in G_{q^n} \subset \mathbb{F}_P$ and have found a root mapping polynomial f with $N(f) = w$. Let S be the equivalence class containing f and the representative polynomial for S be $r(x)$ with $f(x) = a^q r(a^{-1}x)$. Then we know that:

$$\begin{aligned}\mathfrak{C}(f) &= a\mathfrak{C}(r) \\ &= a\end{aligned}$$

and since q is prime, a can be extracted from any non-zero term of the polynomial, except the constant and q^{th} terms.

Let $r(x) = \sum_{i=0}^q c_i x^i$. From lemma 7.3.2 we know that r must have a non-zero coefficient c_d where $0 < d < q$. From the d^{th} -coefficients of the two polynomials, c_d and $a^{q-d}c_d$, we can compute a^{q-d} :

$$a^{q-d} = (c_d)^{-1} (a^{q-d}c_d).$$

Since q is prime, the order of a divides q^n , and $0 < d < q$, an inverse for $(q-d) \bmod q^n$ exists and $\mathfrak{C}(f)$ can be computed without the costly exponentiation of Cipolla's algorithm:

$$\begin{aligned}\mathfrak{C}(f) &\equiv (c_d^{-1} (c_d a^{(q-d)}))^{(q-d)^{-1} \bmod q^n} \\ &\equiv a.\end{aligned}\tag{7.4.1}$$

In other words the q^{th} root of w can be computed with only an inverse, a multiplication, and an exponentiation, all in \mathbb{F}_P .

If the trace of the polynomials is non-zero and known, then since $c_{q-1} = (-1) \text{Tr}(r)$ and $q - d = 1$, we have the formula

$$\mathfrak{C}(f) \equiv \text{Tr}(r)^{-1} \text{Tr}(f).\tag{7.4.2}$$

This is what happens in the case of a minimal order equivalence class. The formula for the trace of the representative polynomial is given in section 8.3, reducing q^{th} root computation to an inverse and multiplication over \mathbb{F}_P .

Chapter 8

Reducing computational requirements

Chapter 7 described the structure of root generating elements and their associated root mapping polynomials. Recall from chapter 2.3 that a root generating element $\eta \in \mathbb{F}_{P^q}$ can also be represented as $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$ with $\eta_{q^n} \in G_{q^n}$ and $\eta_K \in G_K$. The exponentiation by K used in Cipolla's algorithm, $\mathfrak{C}(\eta) = \eta^K$, erases η_K , leaving η_{q^n} unchanged:

$$\widehat{\Psi}(\mathfrak{C}(\eta)) = (\eta_{q^n}, 1).$$

If the order of η_K is small, that is if the root mapping polynomial is from a low order equivalence class, K (which is on the order of P^q for cryptographic sized P, q) can be replaced by a smaller exponent. The first section of this chapter contains the details of this reduced computation Cipolla's algorithm.

The reduced computation version of Cipolla's algorithm in section 8.1 has one requirement: polynomials with arbitrary norms from a low order equivalence class must be found. No efficient algorithm for finding these polynomials is currently known. However, generating the representative polynomials from minimal order classes (definition 7.3.6) is straight forward. The trace function for these polynomials (leading directly to the $(q-1)^{th}$ coefficient) requires a square root

and a few other operations in \mathbb{F}_P , while the remaining coefficients require a bit more work. Section 8.3 derives a simple formula for the trace of the representative polynomials. The formulas for the remaining coefficients are derived in section 8.4.

Although the representative polynomial generates the trivial q^{th} root, $(1)^{\frac{1}{q}} = 1$, knowledge of the representative polynomial and another root mapping polynomial, f , in the same equivalence class reveals $\mathfrak{C}(f)$ (equation (7.4.1)). In other words minimizing the computational requirements of Cipolla's algorithm in these fields implies the existence of another, more efficient q^{th} root algorithm. Knowledge of a non-zero trace simplifies this computation further.

8.1 Cipolla's algorithm for small order equivalence class

Let f be a root mapping polynomial from an equivalence class S . From section 7.2 we know that if f has root η then $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$ where η_K is the root of the representative polynomial for S and $\mathfrak{C}(f) = \eta_{q^n}$. Exponentiation by K in Cipolla's algorithm removes η_K from η because $\text{ord}(\eta_K) \mid K$. If the order of η_K is R , a small factor of K , then exponentiation by R is all that is required to remove η_K .

Exponentiation by the order of the equivalence class removes η_K from η , but unlike exponentiation by K there is no guarantee that it will leave η_{q^n} unchanged. We know from theorem 2.1.1 that $K \equiv 1 \pmod{q^n}$ for $q > 2$, or that the $\eta_{q^n}^K = \eta_{q^n}$. All we know about R is that $R \equiv 1 \pmod{q}$ (theorem 2.1.3). Therefore exponentiation by R may change η_{q^n} .

Fortunately this change can be compensated for by a small exponentiation in

\mathbb{F}_P . Since $K \equiv 1 \pmod{q^n}$ and $R|K$, we know that

$$R^{-1} \equiv \frac{K}{R} \pmod{q^n}.$$

We know that $\text{ord}(\eta^R) | q^n$ therefore

$$\mathfrak{C}(f) = \eta^{R(\frac{K}{R} \pmod{q^n})} \equiv \eta_{q^n}. \quad (8.1.1)$$

Since $(\frac{K}{R} \pmod{q^n}) < q^n$ and for most cases $q^n \ll K$, the size of the required exponentiation can be greatly reduced if an appropriate root mapping polynomial with sufficiently small order can be found.

8.2 Prime order equivalence classes

If we could generate polynomials in low order equivalence classes the reduced computation version of Cipolla's algorithm in section 8.1 could be used. The smallest orders are the prime divisors of K . There is no known efficient technique for finding a root mapping polynomial with a specific norm in these classes, but finding the representative polynomial, in at least one case, is easy. Minimal order equivalence classes, as in definition 7.3.6, have minimal order with respect to q . When these equivalence classes exist, the trace function for their representative polynomials are easy to compute. The formulas are derived in section 8.3 with the remaining coefficients computed in section 8.4.

The trace of these representative polynomials is the key element in computing the coefficients. Furthermore, only the trace of the representative polynomial is needed to compute the q^{th} root from any other polynomial in the class, without the cost of Cipolla's algorithm. Equation (7.4.1) gives an alternate equation for $\mathfrak{C}(f)$ if a non-zero coefficient of the representative polynomial, r , from the equivalence class containing f is known. Equation (7.4.2) gives a simplified version of this formula if the traces of the two polynomials is non-zero and known.

This section will prove some basic properties of the trace function for prime order representative polynomials. Section 8.3 uses these properties to develop trace function formulas for the representative polynomials of minimal order equivalence classes. Since prime order equivalence classes exist for the q^{th} root problem in every finite field, the theorems developed in this section may be used in future research to extend the trace formulas to larger order representative polynomials.

Lemma 7.3.5 states that when $q > 2$ the order of an equivalence class must have the form $R = 2tq + 1$ for some integer $t \geq 1$. For prime orders R , there are $\phi(R) = 2tq$ elements of order R in \mathbb{F}_{P^q} and $\frac{2tq}{q} = 2t$ minimal polynomials for these elements. These are the representative polynomials of order R . Let REP_R be the set of all representative polynomials of order R .

Every polynomial, $r_i(x) \in \text{REP}_R$, has a unique reciprocal polynomial $\bar{r}_i(x) \neq r_i(x)$ (see definition 2.6.9). Since roots of \bar{r}_i are the inverses of roots of r_i , the order of $\bar{r}_i(x)$ is also R . Therefore $\bar{r}_i(x) \in \text{REP}_R$. Furthermore, the reciprocal of the reciprocal is the original polynomial: $\overline{(\bar{r}_i)} = r_i$. Therefore REP_R can be ordered such that $\bar{r}_i = r_{t+i \bmod 2t}$.

The following theorem gives a formula for the sum of the product of reciprocals in REP_R . This theorem is used in the case of $R = 2q + 1$ to compute the two trace values, and may be valuable in extending the trace formulas to larger R .

Theorem 8.2.1. *Let P, q, K be defined as in table 2.1 and $R = 2qt + 1$ a prime integer with $R | K$. If $\text{REP}_R = \{r_i(x) \mid 0 \leq i < 2t\}$ is the set of order R representative polynomials for computing q^{th} roots in \mathbb{F}_P , ordered such that the reciprocal of r_i is $\bar{r}_i(x) = r_{t+i}(x)$, then*

$$\sum_{i=0}^{t-1} \text{Tr}(r_i) \text{Tr}(r_{t+i}) = \frac{R - q}{2} \quad (8.2.1)$$

Proof. Let $\{\eta_i \mid 0 \leq i < 2qt\}$ be the set of all elements of order $R = 2qt + 1$ in

\mathbb{F}_{Pq} . Then:

$$\begin{aligned}
2q \sum_{i=0}^{t-1} \text{Tr}(r_i) \text{Tr}(r_{t+i}) &= \sum_{i=0}^{2qt-1} \text{Tr}(\eta_i) \text{Tr}(\eta_i^{-1}) \\
&= \sum_{i=0}^{2qt-1} \left(\sum_{j=0}^{q-1} \eta_i^{P^j} \right) \left(\sum_{j=0}^{q-1} \eta_i^{-P^j} \right) \\
&= \sum_{i=0}^{2qt-1} \sum_{0 \leq j, k < q} \eta_i^{P^j - P^k} \\
&= \sum_{0 \leq j, k < q} \sum_{i=0}^{2qt-1} \eta_i^{P^j - P^k} \\
&= q \sum_{i=0}^{2qt-1} 1 + \sum_{\substack{0 \leq j, k < q \\ k \neq j}} \sum_{i=0}^{2qt-1} \eta_i^{P^j - P^k}
\end{aligned}$$

Since R is prime, for any constant $c \not\equiv 0 \pmod R$, $\sum_{i=0}^{2qt-1} \eta_i^c = \sum_{i=0}^{2qt-1} \eta_i$ and $\sum_{i=0}^{2qt-1} \eta_i = -1$. Therefore

$$\begin{aligned}
2q \sum_{i=0}^{t-1} \text{Tr}(r_i) \text{Tr}(r_{t+i}) &= 2q^2t + \sum_{\substack{0 \leq j, k < q \\ j \neq k}} (-1) \\
&= 2q^2t - q(q-1) \\
&= q(2qt + 1 - q) \\
&= q(R - q)
\end{aligned}$$

Therefore

$$\sum_{i=0}^{t-1} \text{Tr}(r_i) \text{Tr}(r_{t+i}) = \frac{q(R - q)}{2q} = \frac{R - q}{2}.$$

□

P	order of base field \mathbb{F}_P ;
q	a prime integer, $q > 2$, dividing $(P - 1)$;
R	$R = 2q + 1$ is prime with $P \not\equiv 1 \pmod R$ and $P^q \equiv 1 \pmod R$;
$r(x)$	$r(x) = x^q + \sum_{i=1}^{q-1} c_i x^i + (-1)$, a minimal polynomial over \mathbb{F}_P whose roots have order R .

Table 8.1: Minimal order root mapping polynomial equivalence class restrictions

Other identities for the traces, such as

$$\sum_{i=0}^{2t-1} \text{Tr}(r_i)^2 = -q$$

$$\sum_{i=0}^{t-1} (\text{Tr}(r_i) - \text{Tr}(\bar{r}_i))^2 = -R$$

$$\sum_{i=0}^{t-1} (\text{Tr}(r_i) + \text{Tr}(\bar{r}_i))^2 = R - 2q$$

are simple to prove with this theorem.

8.3 Trace formulas for minimal order representative polynomials

The representative polynomials with minimal order (definition 7.3.6 and table 8.1) will be determined in this section. Examples are worked out in section 8.5, giving formulas for the minimal order representative polynomials for $q = 5$ and $q = 11$. These formulas hold for any base field, \mathbb{F}_P , satisfying the conditions of a minimal order representative class given in section 7.3.

Let $\eta \in \mathbb{F}_{P^q}$ be an element of order $R = 2q + 1$ with minimal polynomial $r(x) = \sum_{i=0}^q c_i x^i$. Formulas for the traces of η and η^{-1} will be developed in this section by computing the sum and product of the two traces. These two values are then used to generate a quadratic equation with the trace as a variable. Solving this equation gives the two trace values.

Theorem 8.3.1. *Let $P, q, R, r(x)$ be defined as in table 8.1 and a root of $r(x)$ be η . Then*

$$\text{Tr}(\eta) + \text{Tr}(\eta^{-1}) \equiv -1 \quad (8.3.1)$$

$$\text{Tr}(\eta) \text{Tr}(\eta^{-1}) \equiv \frac{q+1}{2} \quad (8.3.2)$$

Proof. Let $P, q, R, r(x)$ be defined as in table 8.1 and $\eta \in \mathbb{F}_{P^q}$ be a root of $r(x)$. Since r and \bar{r} are degree q irreducible polynomials, the set of conjugates, $\{\eta^{P^j}, \eta^{-P^j} \mid 0 \leq j < q\}$, contain $2q$ unique elements of order R . There are $R-1 = 2q$ elements of order R in \mathbb{F}_{P^q} , therefore $\{\eta^{P^j}, \eta^{-P^j} \mid 0 \leq j < q\} = \{\eta^i \mid 0 < i < R\}$. Therefore

$$\begin{aligned} \text{Tr}(\eta) + \text{Tr}(\eta^{-1}) &= \sum_{i=0}^{q-1} \eta^{P^i} + \sum_{i=0}^{q-1} \eta^{-P^i} \\ &= \sum_{i=1}^{R-1} \eta^i \end{aligned}$$

Since $\text{ord}(\eta) = R$ we know that $\sum_{i=0}^{R-1} \eta^i = 0$ and $\text{Tr}(\eta) + \text{Tr}(\eta^{-1}) = \sum_{i=0}^{R-1} \eta^i - 1 = -1$.

From theorem 8.2.1 we know that $\text{Tr}(\eta) \text{Tr}(\eta^{-1}) = \frac{R-q}{2} = \frac{q+1}{2}$. □

A direct result of this theorem is a formula for the trace functions of $r(x)$.

Corollary 8.3.2. *Let $P, q, R, r(x)$ be defined as in table 8.1 and a root of $r(x)$ be η . Then the formula for the traces of η and η^{-1} are*

$$\text{Tr}(\eta), \text{Tr}(\eta^{-1}) \in \left\{ 2^{-1} \left(-1 \pm (-R)^{\frac{1}{2}} \right) \right\} \quad (8.3.3)$$

with the difference of the traces

$$\text{Tr}(\eta) - \text{Tr}(\eta^{-1}) \equiv \pm (-R)^{\frac{1}{2}}$$

Proof. Theorem 8.3.1 gave us that $Tr(\eta) + Tr(\eta^{-1}) = -1$ and $Tr(\eta) Tr(\eta^{-1}) = \frac{q+1}{2}$. Together these generate the quadratic equation:

$$Tr(\eta)^2 + Tr(\eta) + \frac{q+1}{2} \equiv 0.$$

Solving for the trace with the quadratic formula gives:

$$Tr(\eta), Tr(\eta^{-1}) \in \left\{ 2^{-1} \left(-1 \pm (-R)^{\frac{1}{2}} \right) \right\}.$$

With these formulas for the trace function, the difference of the traces can be computed:

$$Tr(\eta) - Tr(\eta^{-1}) \equiv 2^{-1} \left[\left(-1 \pm (-R)^{\frac{1}{2}} \right) - \left(-1 \mp (-R)^{\frac{1}{2}} \right) \right] \equiv \pm (-R)^{\frac{1}{2}}.$$

□

8.4 Minimal order representative polynomials

In this section the trace functions developed in the last section will be used to compute the individual coefficients of the two minimal order representative polynomials. The final formulas are independent of the base field and hold for any chosen field.

As before, let $r(x) = \sum_{i=0}^q c_i x^i$ be a minimal representative polynomial. If η is a root of $r(x)$, we know that $r(x) = \prod_{i=0}^{q-1} (x - \eta^{P^i})$. Let \mathcal{A}_i be the set of all subsets of $\{0, 1, \dots, (q-1)\}$ of size i . Expanding this formula for $r(x)$ gives us:

$$r(x) = \sum_{i=0}^q \left((-1)^{q-i} \sum_{A \in \mathcal{A}_{q-i}} \eta^{\sum_{j \in A} P^j} \right) x^i \quad (8.4.1)$$

and formulas for the individual coefficients are

$$c_i = (-1)^{q-i} \sum_{A \in \mathcal{A}_{q-i}} \eta^{\sum_{j \in A} P^j}. \quad (8.4.2)$$

Let $h \in \mathbb{Z}_R$ be a primitive q^{th} root of unity and define $G : \mathcal{A}_i \rightarrow \mathbb{Z}_R$ by $G(A) = \sum_{k \in A} h^k \pmod R$. Applying G to $\mathcal{B} \subseteq \mathcal{A}_i$ is defined by

$$G(\mathcal{B}) = \{G(A) \mid A \in \mathcal{B}\},$$

the collection of all $G(\mathcal{B})$ counting multiplicities. Recall that P is a primitive q^{th} root of unity. When G is applied to the full set, \mathcal{A}_i , $G(\mathcal{A}_i)$ is independent of which q^{th} root of unity is used.

Lemma 8.4.1. *Let q and R be defined as in table 8.1, $h \in \mathbb{Z}_R$ be a primitive q^{th} root of unity and \mathcal{A}_i be the set of all subsets of $\{0, 1, \dots, (q-1)\}$ of size i . The set*

$$G(\mathcal{A}_i) = \{G(A) \mid A \in \mathcal{A}_i\}$$

where $G(A) = \sum_{a_k \in A} h^{a_k} \pmod R$ is independent of the q^{th} root of unity h used.

Proof. Let $A \in \mathcal{A}_i$ with $A = \{a_k\}_{k=0}^{i-1}$, $m \in \mathbb{Z}_q^*$ and define $mA = \{ma_k \pmod q\}_{k=0}^{i-1}$ with $m\mathcal{A}_i = \{mA \mid A \in \mathcal{A}_i\}$. We know that A contains i distinct elements from $\{0, 1, \dots, (q-1)\}$ and m is a unit, therefore $mA \in \mathcal{A}_i$ and for $A, B \in \mathcal{A}_i$, $mA = mB$ if and only if $A = B$. Every possible subset of i distinct elements is contained in \mathcal{A}_i therefore $m\mathcal{A}_i = \mathcal{A}_i$. If $h \in \mathbb{Z}_R$ is a primitive q^{th} root of unity then for any other primitive q^{th} root of unity $b \in \mathbb{Z}_R \exists m \in \mathbb{Z}_q^*$ such that $h \equiv b^m$.

Therefore

$$\begin{aligned} G(\mathcal{A}_i) &= \left\{ \sum_{j \in A} b^{mj} \pmod R \mid A \in \mathcal{A}_i \right\} \\ &= \left\{ \sum_{j \in A} b^j \pmod R \mid A \in m\mathcal{A}_i \right\} \\ &= \left\{ \sum_{j \in A} b^j \pmod R \mid A \in \mathcal{A}_i \right\}. \end{aligned} \tag{8.4.3}$$

□

Using this function the formula for the coefficients can be rewritten as

$$c_i = (-1)^{q-i} \sum_{k \in G(\mathcal{A}_{q-i})} \eta^k.$$

Simplified formulas for the coefficients are obtained by partitioning \mathcal{A}_i into subsets $B \in \mathcal{A}_i$ such that $\sum_{j \in G(B)} \eta^j$ is a known value. Counting the subsets which generate the different fixed values gives us the coefficient formulas.

The formula for the $(q-i)^{th}$ coefficient is obtained by analyzing \mathcal{A}_i . This analysis only needs to be done for $1 < i \leq \frac{q-1}{2}$. From lemma 2.6.10 we know that $\bar{c}_i = c_0^{-1} c_{q-i}$. Therefore the coefficients for $i \in \{0, 1, q-1, q\}$ are already known: $c_0 = -1$, $c_{q-1} = -Tr(\eta)$, $c_1 = Tr(\eta^{-1})$. Furthermore the sets \mathcal{A}_i and \mathcal{A}_{q-i} are closely related, with the partitioning of \mathcal{A}_i giving us the partitioning for \mathcal{A}_{q-i} . Define $\bar{A} = \{0, 1, \dots, (q-1)\} \setminus A$ where $A \in \mathcal{A}_i$. Then the set \mathcal{A}_{q-i} for $0 \leq i \leq q$ can be redefined as

$$\mathcal{A}_{q-i} = \{\bar{A} \mid A \in \mathcal{A}_i\}.$$

Since $\sum_{k \in A} P^k + \sum_{k \in \bar{A}} P^k = \sum_{k=0}^{q-1} P^k = 0$, we know that $\sum_{k \in \bar{A}} P^k = -\sum_{k \in A} P^k$ and

$$c_i = (-1)^{q-i} \sum_{k \in G(\mathcal{A}_i)} \eta^{-k} \quad (8.4.4)$$

$$c_{q-i} = (-1)^i \sum_{k \in G(\mathcal{A}_i)} \eta^k. \quad (8.4.5)$$

The partition of \mathcal{A}_i used to create formulas for the coefficients divides \mathcal{A}_i into subsets of size q . Therefore each coefficient requires

$$\frac{ord(\mathcal{A}_i)}{q} = \frac{\binom{q}{i}}{q}$$

computations in \mathbb{Z}_R . Using this technique there are

$$\sum_{i=2}^{\frac{q-1}{2}} \frac{\binom{q}{i}}{q} = \frac{2^{q-1} - 1}{q} - 1 \quad (8.4.6)$$

computations in \mathbb{Z}_R to perform to obtain the coefficients of r, \bar{r} . For cryptographically sized q this is computationally infeasible. However for smaller q it allows us to generate formulas for the minimal representative polynomials for a given $(q, R = 2q + 1)$ pair.

8.4.1 Distributions in \mathbb{Z}_R

The following lemma defines the basic sets used in formulas for the remaining coefficients. An element $\eta \in \mathbb{F}_{P^q}$ of order R generates all elements of order R in \mathbb{F}_{P^q} . Since $R = 2q + 1$ is prime, there are $2q$ elements of order R . The q conjugates of η are η^{P^j} for $0 \leq j < q$. The remaining q elements of order R are the conjugates of η^{-1} .

Lemma 8.4.2. *Let P, q, R be defined as in table 8.1 and*

$$S_r = \{P^j \bmod R \mid 0 \leq j < q\} \quad (8.4.7)$$

$$S_n = \{-P^j \bmod R \mid 0 \leq j < q\} \quad (8.4.8)$$

$$S_z = \{0\} \quad (8.4.9)$$

be sets of elements in \mathbb{Z}_R . Then S_r contains all quadratic residues, S_n contains all quadratic non-residues, and

$$\mathbb{Z}_R = S_n \cup S_r \cup S_z \quad (8.4.10)$$

Proof. Let P, q, R be defined as in table 8.1 and S_r, S_n, S_z be defined as above. By definition, $P^{\frac{R-1}{2}} \equiv P^q \equiv 1 \pmod R$ so P is a quadratic residue. Since q is odd, $(-1)^q \equiv -1 \pmod R$ so (-1) is a quadratic non-residue. Therefore from corollary 2.4.4, P^i are quadratic residues and $-P^i$ are quadratic non-residues in \mathbb{Z}_R for all $i \in \mathbb{Z}$. This implies

- $S_r = \{P^i \bmod R \mid 0 \leq i < q\}$ contains only quadratic residues and

- $S_n = \{-P^i \bmod R \mid 0 \leq i < q\}$ contains only quadratic non-residues.

Since 0 is neither a quadratic residue or non-residue, the three collections, S_r, S_n, S_z , are pairwise disjoint. P is a primitive q^{th} root of unity in \mathbb{Z}_R since q is prime, $P \not\equiv 1 \bmod R$, and $P^q \equiv 1 \bmod R$, therefore $P^i \equiv P^j \bmod R$ for $0 \leq i, j < q$ if and only if $i \equiv j \bmod q$. This implies that S_r, S_n each contains q distinct elements. Therefore $\text{ord}(S_r \cup S_n \cup S_z) = 2q + 1 = R$ and $\mathbb{Z}_R = S_r \cup S_n \cup S_z$. \square

The first thing to notice about this partitioning of \mathbb{Z}_R is that S_r, S_n generate two sets of roots, one for each minimal order root mapping polynomial. If $\eta \in \mathbb{F}_{P^q}$ has order R then all $(R - 1) = 2q$ elements of order R are in

$$\{\eta^i \mid i \in S_r\} \cup \{\eta^i \mid i \in S_n\}$$

with $\{\eta^i \mid i \in S_z\} = \{1\}$. Let $r(x)$ be the minimal polynomial for η . The conjugates of η , and therefore all the roots of $r(x)$, are $\{\eta^{P^i} \mid 0 \leq i < q\} = \{\eta^i \mid i \in S_r\}$. Let $\bar{r}(x)$ be the reciprocal polynomial for $r(x)$, as in definition 2.6.9. All of the conjugates of η^{-1} , and therefore all roots of $\bar{r}(x)$, are in $\{\eta^{-P^i} \mid 0 \leq i < q\} = \{\eta^i \mid i \in S_n\}$.

8.4.2 A partitioning of \mathcal{A}_i

This section will show that every set \mathcal{A}_i can be partitioned into subsets $B \subset \mathcal{A}_i$ with $\text{ord}(B) = q$ such that

$$G(B) \in \{S_r, S_n, q \cdot S_z\}$$

where $q \cdot S_z$ represents q copies of the set S_z . If r_i, n_i, z_i are the number of subsets $B \subset \mathcal{A}_i$ generating $S_r, S_n, q \cdot S_z$ respectively, then the formulas for the coefficients

are

$$\begin{aligned} c_{q-i} &\equiv (-1)^i \left(r_i \sum_{j \in S_r} \eta^j + n_i \sum_{j \in S_n} \eta^j + qz_i \right) \\ &\equiv (-1)^i (r_i \text{Tr}(\eta) + n_i \text{Tr}(\eta^{-1}) + qz_i). \end{aligned} \quad (8.4.11)$$

The partitioning is accomplished by characterizing elements $A \in \mathcal{A}_i$ by a set of minimal distances between each $a_j \in A$. The minimal distances force an ordering on A up to cyclic shifts. However the elements in A , and therefore their distances, are in \mathbb{Z}_q which does not have a well defined ordering. For this reason the following definition and lemma are needed for a complete ordering.

Definition 8.4.3 (Minimal distance): Let \mathcal{A}_i be the set of all subsets of $\{0, 1, \dots, (q-1)\}$ of size i , with $1 < i < (q-1)$ and $A \in \mathcal{A}_i$. If $j, k \in A$ with $j \neq k$, the (ordered) **distance** from k to j is:

$$D(k, j) = (j - k) \bmod q \in \{0, 1, \dots, (q-1)\} \quad (8.4.12)$$

Using the integer ordering $0 < 1 < \dots < (q-1) < q$, the **minimal distance element** of $k \in A$, is $j \in A$, $j \neq k$ such that $D(k, j) < D(k, t)$ for all $t \in A$ and $t \notin \{k, j\}$.

Every element in A is distinct, therefore the distances $D(j, k)$ from $a_j \in A$ to every other $a_k \in A$ are distinct. Because these distances are distinct, minimal distances for each $a_j \in A$ exist and are also distinct. These distances create an ordering on elements in A .

Definition 8.4.4 (Minimal distance ordering): Let \mathcal{A}_i be the set of all distinct subsets of $\{0, 1, \dots, (q-1)\}$ of size i , with $1 < i < (q-1)$. A **minimal distance ordering** of $A \in \mathcal{A}_i$ is $A = \{a_0, a_1, \dots, a_{i-1}\}$ such that for $0 \leq j < i$ the element $a_{(j+1) \bmod i}$ has the minimal distance from a_j .

A minimal distance ordering only determines adjacent elements. If $A = \{a_j\}_{j=0}^{i-1}$ is a minimal distance ordering then so is any cyclic shift of A :

$$\{a_k, a_{(k+1 \bmod i)}, \dots, a_{(k-1 \bmod i)}\}, \quad 0 \leq k < i.$$

Lemma 8.4.5 (Minimal distance ordering exists and is distinct up to cyclic shift). *Let \mathcal{A}_i be the set of all distinct subsets of $\{0, 1, \dots, q-1\}$ of size i where q is prime, $1 < i < (q-1)$, and $A \in \mathcal{A}_i$. Then there exists a minimal distance ordering of elements in A :*

$$A = \{a_0, a_1, a_2, \dots, a_{i-1}\}$$

which is unique up to cyclic shift.

Proof. A minimal distance ordering is determined by distinct minimal distance elements, so if an ordering exists it uniquely determines adjacent elements. Therefore if a minimal distance ordering exists, it is distinct up to cyclic shift.

Since $a_j \in \{0, 1, \dots, q-1\}$ and $a_j \neq a_k \forall (k \neq j)$, the set can be ordered over the integers such that $a_j < a_{j+1}$ for $0 \leq j < (i-1)$. This ordering over the integers is also a minimal distance ordering. For $0 \leq j < (i-1)$ the distance function described over the integers is

$$D(a_j, a_k) = \begin{cases} a_k - a_j & j < k < i \\ q + a_k - a_j & 0 \leq k < j \end{cases},$$

therefore

$$D(a_j, a_{j+1}) = a_{j+1} - a_j < a_k - a_j = D(a_j, a_k) \quad j+1 < k < i$$

$$D(a_j, a_{j+1}) = a_{j+1} - a_j < q + a_k - a_j = D(a_j, a_k) \quad 0 \leq k < j$$

For $j = i-1$, $D(a_{i-1}, a_j) = q + a_j - a_{i-1}$, therefore $D(a_{i-1}, a_0) < D(a_{i-1}, a_j)$ for $0 < j < (i-1)$. Therefore the ordering described is a minimal ordering, and minimal orderings exist. \square

Example 8.4.6 – Minimal distance and minimal distance ordering:

For example, let $q = 11$, $h = 2$ be a primitive 11-th root of unity in \mathbb{Z}_{23} , and $A \in \mathcal{A}_4$ be $A = \{3, 1, 7, 9\}$. Notice that $G(A) = 2^3 + 2^1 + 2^7 + 2^9 = 8 + 2 + 13 + 6 = 6$. The distances $D(k, t)$ for the elements are:

$k \setminus t$	3	1	7	9
3	–	9	4	6
1	2	–	6	8
7	7	5	–	2
9	5	3	9	–

The minimal distance orderings of $A = \{3, 1, 7, 9\}$ are $\{1, 3, 7, 9\}$, $\{3, 7, 9, 1\}$, $\{7, 9, 1, 3\}$ and $\{9, 1, 3, 7\}$.

If $A = \{a_0, a_1, \dots, a_{i-1}\}$ is a minimal distance ordering for A and the ordered collection of distances is

$$\mathcal{D} = \{d_j \mid 0 \leq j < i\}$$

where $d_j = D(a_j, a_{j+1 \bmod i})$, then an alternate representation of A is:

$$(a_0; \mathcal{D}) = \left\{ a_0 + \sum_{l=0}^{j-1} d_l \bmod q \mid 0 \leq j < i \right\}. \quad (8.4.13)$$

Distance representations such as this are used to partition \mathcal{A}_i .

Definition 8.4.7 (Distance representation and distance representation sets): Let \mathcal{A}_i be the set of all distinct subsets of $\{0, 1, \dots, (q-1)\}$ of size i with $1 < i < (q-1)$, $A \in \mathcal{A}_i$ and

$$A = \{a_0, a_1, \dots, a_{i-1}\}$$

a minimal distance ordering of A . The distance representation set of A is the set with multiplicities $\mathcal{D} = \{d_0, d_1, \dots, d_{i-1}\}$ ordered up

to cyclic shifts with $d_j = D(a_j, a_{j+1 \bmod i})$ as defined in equation (8.4.12) and

$$A = (a_0; \mathcal{D}) = \left\{ a_j \mid 0 \leq j < i; a_j \equiv a_0 + \sum_{l=0}^{j-1} d_l \pmod{q} \right\}$$

is a distance representation of A .

With this definition the distance representation set is:

$$\begin{aligned} \mathcal{D} &= \{d_0, d_1, \dots, d_{i-1}\} \\ &\equiv \{d_t, d_{t+1 \bmod i}, \dots, d_{t+(i-1) \bmod i}\} \end{aligned}$$

for any $0 \leq t < i$. However when used in a distance representation, the particular cyclic shift must be known:

$$A = (a_t; \{d_t, d_{t+1 \bmod i}, \dots, d_{t+(i-1) \bmod i}\}) \quad (8.4.14)$$

Example 8.4.8 – Distance representation: From example 8.4.6 let $A = \{1, 3, 7, 9\}$ be a minimal distance ordering with distance representation set $\mathcal{D} = \{2, 4, 2, 3\}$. The set A can be represented by any of the following distance representations:

$$\begin{array}{ll} (1; \{2, 4, 2, 3\}) & (3; \{4, 2, 3, 2\}) \\ (7; \{2, 3, 2, 4\}) & (9; \{3, 2, 4, 2\}) \end{array}$$

Working out the first distance representation gives:

$$A = \{1, (1 + 2 = 3), (3 + 4 = 7), (7 + 2 = 9)\}$$

with $9 + 3 = a_0 \equiv 1 \pmod{11}$.

Distance representation sets for $A \in \mathcal{A}_i$ are distinct (up to cyclic shifts) as they are directly related to the minimal distance ordering of A . Every distance representation with the properties in lemma 8.4.9 represents some $A \in \mathcal{A}_i$.

Lemma 8.4.9 (Distance representation set properties). *The ordered set with multiplicities $\mathcal{D} = \{d_j \mid 0 \leq j < i\}$ is a distance representation set for elements in \mathcal{A}_i ($1 < i < q - 1$) if and only if:*

1. $0 < d_j < q$ for $0 \leq j < i$;

2. $\sum_{j=0}^{i-1} d_j = q$.

Proof. Let $A \in \mathcal{A}_i$ be $A = \{a_0, a_1, \dots, a_{i-1} \mid 0 \leq a_j < q\}$ be ordered such that $a_j < a_{j+1}$ for $0 \leq j < (i - 1)$. This ordering is a minimal ordering (proved in lemma 8.4.5) and has distance representation set $\mathcal{D} = \{d_0, d_1, \dots, d_{i-1}\}$ with $d_j = a_{j+1} - a_j$ for $0 \leq j < (i - 1)$ and $d_{i-1} = q + a_0 - a_{i-1}$. Therefore $\sum_{j=0}^{i-1} d_j = q$ and by definition $0 \leq d_j < q$. Minimal distance orderings are unique up to cyclic shifts, therefore the distance representation set is unique up to cyclic shifts. Therefore if \mathcal{D} is a distance representation set then it satisfies the constraints of the lemma.

Let $\mathcal{D} = \{d_j \mid 0 < d_j < q; 0 \leq j < i\}$ with $\sum_{j=0}^{i-1} d_j = q$. Then $(a_0; \mathcal{D}) \in \mathcal{A}_i$ for $0 \leq a_0 < q$: The set $(a_0; \mathcal{D})$ contains i elements by definition, so $(a_0; \mathcal{D}) \notin \mathcal{A}_i$ if and only if there exists $0 \leq j < k < i$ such that $a_j = a_k$. This implies

$$a_0 + \sum_{l=0}^{j-1} d_l \equiv a_0 + \sum_{l=0}^{k-1} d_l \pmod{q}$$

$$0 \equiv \sum_{l=j}^{k-1} d_l \pmod{q}.$$

Since $0 \leq j < k < i$ we know that $0 < \sum_{l=j}^{k-1} d_l < q$ which contradicts this assumption, therefore $(a_0; \mathcal{D})$ contains i distinct elements and $(a_0; \mathcal{D}) \in \mathcal{A}_i$. \square

Example 8.4.10 – Distance representation: From example 8.4.8, $A = \{1, 3, 7, 9\}$ has the distance representation $A = (1; \mathcal{D})$ with

$\mathcal{D} = \{2, 4, 2, 3\}$. Other sets in \mathcal{A}_4 generated by this distance set are $(0; \mathcal{D}) = \{0, 2, 6, 8\}$ and $(3; \mathcal{D}) = \{3, 5, 9, 0\}$. Recall from example 8.4.6 that $G(A) = 6$ with $h \equiv 2$. Notice that $G((0; \mathcal{D})) = 3$, $G((3; \mathcal{D})) = 1$ and $6, 3, 1 \in S_r$, all quadratic residues in \mathbb{Z}_{23} .

Given the properties of distance representation from lemma 8.4.9, we can show that the set of all possible distance representations partitions \mathcal{A}_i and that each distinct (up to cyclic shift) distance representation set \mathcal{D} generates exactly q of sets $A \in \mathcal{A}_i$. Furthermore, if $G(\mathcal{D})$ is G applied to the subset of \mathcal{A}_i generated by the distance representation set \mathcal{D} , then $G(\mathcal{D}) \in \{S_r, S_n, q \cdot S_z\}$.

Lemma 8.4.11 (Distance representations generate q elements from \mathcal{A}_i). *If \mathcal{D} is a distance representation set from \mathcal{A}_i ($1 < i < (q - 1)$), then:*

1. \mathcal{D} is distinct for $A \in \mathcal{A}_i$ up to cyclic shifts;
2. \mathcal{D} defines exactly q distinct sets $(t; \mathcal{D}) \in \mathcal{A}_i$, for $t \in \mathbb{Z}_q$ as defined in equation (8.4.13);

3. The set

$$G(\mathcal{D}) = \left\{ \sum_{k \in (t; \mathcal{D})} h^k \bmod R \mid 0 \leq t < q \right\} \quad (8.4.15)$$

where $h \in \mathbb{Z}_R$ is a primitive q^{th} root of unity, is either S_r , S_n or $q \cdot S_z$.

Proof. Let $\mathcal{D} = \{d_k \mid 0 \leq k < i\}$ be a distance representation set of \mathcal{A}_i with $1 < i < (q - 1)$.

1. The minimal distance ordering for $A \in \mathcal{A}_i$ is distinct up to cyclic shift (lemma 8.4.5) and \mathcal{D} is defined by this ordering, therefore \mathcal{D} is distinct up to cyclic shift.

2. Let $t, z \in \mathbb{Z}_q$ with $(t; \mathcal{D}) = (z; \mathcal{D})$. Since minimal distance orderings are unique up to cyclic shifts, $\exists 0 < r \leq i$ such that $t + \sum_{j=0}^{r-1} d_j \equiv z \pmod{q}$. But this implies that $d_{k+r \bmod i} = d_k$ for all $0 \leq k < i$. Therefore \mathcal{D} has a repeated cycle of length r and

$$\begin{aligned} \sum_{j=0}^{i-1} d_j &= \frac{i}{r} \sum_{j=0}^{r-1} d_j \\ q &= \frac{i}{r} \sum_{j=0}^{r-1} d_j. \end{aligned}$$

Since $0 < r \leq i < (q-1)$ we know that $0 < \sum_{j=0}^{r-1} d_j \leq q$. We know that $\frac{i}{r}$ is a divisor of q , $i < q$ and q prime therefore $r = i$, $\frac{i}{r} = 1$ and $\sum_{j=0}^{r-1} d_j = q$. Therefore $t = z$ and every $t \in \mathbb{Z}_q$ generates a unique set $(t; \mathcal{D}) \in \mathcal{A}_i$.

3. We know that:

$$\begin{aligned} G(\mathcal{D}) &= \left\{ \sum_{k \in (u; \mathcal{D})} h^k \pmod{R} \mid 0 \leq u < q \right\} \\ &= \left\{ \sum_{j=0}^{i-1} h^{u + \sum_{k=0}^{j-1} d_k} \pmod{R} \mid 0 \leq u < q \right\} \\ &= \{h^u G((0; \mathcal{D})) \pmod{R} \mid 0 \leq u < q\} \end{aligned} \quad (8.4.16)$$

and since h is a quadratic residue, every element in $G(\mathcal{D})$ has the same quadratic residuosity (from corollary 2.4.4) or all are 0. If $G((0; \mathcal{D})) \not\equiv 0 \pmod{R}$ then every element in $G(\mathcal{D})$ is distinct: Assume that $\exists 0 \leq u, v < q$ such that $G((u; \mathcal{D})) = G((v; \mathcal{D}))$. Then $h^{u-v} G((0; \mathcal{D})) \equiv G((0; \mathcal{D})) \pmod{R}$ and since $G((0; \mathcal{D})) \not\equiv 0 \pmod{R}$ we know that $u = v$. Therefore $G(\mathcal{D})$ contains q distinct elements and either $G(\mathcal{D}) = q \cdot S_z$ or $G(\mathcal{D}) = S_r$ or $G(\mathcal{D}) = S_n$.

□

Lemma 8.4.11 tells us that for all distance representation set \mathcal{D} ,

$$\sum_{j \in G(\mathcal{D})} \eta^j \in \{Tr(\eta), Tr(\eta^{-1}), q\}.$$

The last step in generating the formulas for the coefficients is to show that \mathcal{A}_i is partitioned by the distance representation sets. The set of all distance representation sets is well defined for a given \mathcal{A}_i (lemma 8.4.9), every $A \in \mathcal{A}_i$ is represented by exactly one distance representation set and every distance representation set generates q distinct elements in \mathcal{A}_i (lemma 8.4.11). This leads immediately to partitioning of \mathcal{A}_i .

Theorem 8.4.12 (Distance representations partition \mathcal{A}_i). *The set \mathbb{D}_i of all distinct distance representation sets \mathcal{D} with properties described in lemma 8.4.9 partitions \mathcal{A}_i into:*

$$\mathcal{A}_i = \bigcup_{\mathcal{D} \in \mathbb{D}_i} \bigcup_{t=0}^{q-1} (t; \mathcal{D})$$

$$\text{with } ord(\mathbb{D}_i) = \frac{\binom{q}{i}}{q}.$$

Proof. From lemma 8.4.11 each distance representation set \mathcal{D} represents exactly q elements in \mathcal{A}_i and each $A \in \mathcal{A}_i$ is represented by exactly one $\mathcal{D} \in \mathbb{D}_i$. Therefore

$$\mathcal{A}_i = \bigcup_{\mathcal{D} \in \mathbb{D}_i} \bigcup_{t=0}^{q-1} (t; \mathcal{D})$$

$$\text{and } ord(\mathbb{D}_i) = \frac{ord(\mathcal{A}_i)}{q} = \frac{\binom{q}{i}}{q}. \quad \square$$

Letting r_i, n_i, z_i be the number of distance representation sets from \mathcal{A}_i which generate $S_r, S_n, q \cdot S_z$ respectively gives us equation (8.4.11). Since the computation of r_i, n_i, z_i and the trace formulas do not rely on P , formulas for the coefficients can be generated which are independent of the actual base field used.

8.5 Minimal representative polynomials for $q = 5$ and $q = 11$

The first example uses $q = 5$, $R = 11$. The value $h = 4$ is a quadratic residue/ q^{th} root of unity in \mathbb{Z}_{11} . Only the difference representation sets for \mathcal{A}_2 need to be analyzed.

Difference representation
sets for $\mathcal{A}_2/\mathcal{A}_3$

\mathcal{D}	$G((0; \mathcal{D}))$	in
1 4	$4^0 + 4^1 = 5$	S_r
2 3	$4^0 + 4^2 = 6$	S_n

$$r_2 = n_2 = 1 = r_3 = n_3$$

$$z_2 = 0 = z_3$$

Therefore

$$c_2 = (-1)^3 (Tr(r) + Tr(\bar{r})) = 1 \quad c_3 = (-1)^2 (Tr(r) + Tr(\bar{r})) = -1$$

$$\bar{c}_2 = (-1)^3 (Tr(\bar{r}) + Tr(r)) = 1 \quad \bar{c}_3 = (-1)^2 (Tr(\bar{r}) + Tr(r)) = -1$$

For a concrete example, the prime $P = 751$ has extension field \mathbb{F}_{751^5} with minimal order equivalence classes. Let $Tr(r) = 2^{-1} (-1 + (-11)^{1/2}) = 199$, $Tr(\bar{r}) = 2^{-1} (-1 - (-11)^{1/2}) = 551$, then the minimal representative polynomials are:

$$r(x) = x^5 + 552x^4 + 750x^3 + x^2 + 551x + 750$$

$$\bar{r}(x) = x^5 + 200x^4 + 750x^3 + x^2 + 199x + 750$$

For a slightly more complex example, let $q = 11$ and $R = 23$. The value $h = 2$ is a quadratic residue/ q^{th} root of unity in \mathbb{Z}_{23} . For $q = 11$ the difference representation sets for $\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5$ must be analyzed. The (r_i, n_i, z_i) triplets for these difference representations are:

i	r_i	n_i	z_i
2	2	3	0
3	7	7	1
4	15	14	1
5	19	21	2

The order 23 degree 11 representative polynomials for appropriate fields are $r(x) = \sum_{i=0}^q c_i x^i$:

$$\begin{aligned}
c_{10} &= -Tr(r) & c_1 &= Tr(\bar{r}) \\
c_9 &= (2Tr(r) + 3Tr(\bar{r})) = Tr(\bar{r}) - 2 & c_2 &= -(Tr(r) - 2) \\
c_8 &= -(7Tr(r) + 7Tr(\bar{r}) + 11) = -4 & c_3 &= 4 \\
c_7 &= (15Tr(r) + 14Tr(\bar{r}) + 11) = Tr(r) - 3 & c_4 &= -(Tr(\bar{r}) - 3) \\
c_6 &= -(19Tr(r) + 21Tr(\bar{r}) + 22) = -(2Tr(\bar{r}) + 3) & c_5 &= 2Tr(r) + 3
\end{aligned}$$

For a concrete example let $P = 63647$ with $(-23)^{\frac{1}{2}} = \pm 5481$. Setting $Tr(r) = 2740$ and $Tr(\bar{r}) = 60906$ gives order 23 representative polynomials:

$$\begin{aligned}
r(x) &= x^{11} + 60907x^{10} + 60904x^9 + 63643x^8 + 2737x^7 + 5749x^6 \\
&\quad + 5483x^5 + 2744x^4 + 4x^3 + 60909x^2 + 60906x + (-1) \\
\bar{r}(x) &= x^{11} + 2741x^{10} + 2738x^9 + 63643x^8 + 60903x^7 + 58164x^6 \\
&\quad + 57898x^5 + 60910x^4 + 4x^3 + 2743x^2 + 2740x + (-1)
\end{aligned}$$

Chapter 9

Fourier transforms on root mapping polynomial and elements in \mathbb{F}_{Pq}

The ill-defined nature of the q^{th} root function is the main difficulty in the equivalence proof in chapter 5. For any q^{th} residue $w \in \mathbb{F}_P$ there exist q valid solutions to $(w)^{\frac{1}{q}}$ in \mathbb{F}_P . However, once one solution is known, the remaining are trivial to compute (lemma 2.5.3): if $(w)^{\frac{1}{q}} = a$, then

$$\left\{ (w)^{\frac{1}{q}} \right\} = \{ ah^i \mid 0 \leq i < q \}$$

where $h \in \mathbb{F}_P$ is a primitive q^{th} root of unity.

Cipolla's algorithm generates q^{th} roots using the root mapping polynomials defined in chapter 6. The algorithm returns random q^{th} roots because generation of root mapping polynomials is random. However, just as with q^{th} roots, once a root mapping polynomial is known, polynomials returning the remaining roots are easily found (lemma 7.2.3): if $f(x)$ is a root mapping polynomial for which $\mathfrak{C}(f) = (w)^{\frac{1}{q}} = a$, then

$$\left\{ (w)^{\frac{1}{q}} \right\} = \{ \mathfrak{C}(f_i) \mid f_i(x) = h^q f(h^{-i}x); 0 \leq i < q \} \quad (9.0.1)$$

where $h \in \mathbb{F}_P$ is once again a primitive q^{th} root of unity and $\mathfrak{C}(f_i) = ah^i$.

A primitive q^{th} root of unity also gives us an alternate way to compute the q conjugates for an element. If $u \in \mathbb{F}_P$ is a q^{th} non-residue, then $f(x) = (x^q - u)$

is irreducible over \mathbb{F}_P (see lemma 2.7.1). We will show in lemma 9.1.1 that the conjugates of $\eta \in \mathbb{F}_P[x]/(f(x))$, where $\eta = \sum_{j=0}^{q-1} r_j x^j$, are

$$\left\{ \eta^{P^i} \mid 0 \leq i < q \right\} = \left\{ \eta_i \mid 0 \leq i < q; \eta_i = \sum_{j=0}^{q-1} (h^{ji} r_j) x^j \right\}. \quad (9.0.2)$$

Notice that the formulas for f_i and η_i are very similar:

$$f_i(x) = x^q + \sum_{j=0}^{q-1} (h^{-ji} c_i) x^j \qquad \eta_i = \sum_{j=0}^{q-1} (h^{ji} r_j) x^j$$

and resemble discrete Fourier transforms. This chapter explores the relationship of discrete Fourier transforms to these alternate root generating polynomials (f_i) and the almost identical formulas for conjugates (η_i). A simple cyclic shift of a transformed root mapping polynomial or element in \mathbb{F}_{P^q} generates the q alternative forms. Therefore the q alternative forms can be represented by a single cyclicly ordered set of q elements in \mathbb{F}_P . Examining polynomials, or their roots, in their transformed state gives us a representation independent of the root generated or which conjugate it is.

Besides giving us a simplified form in which to examine root mapping polynomial and their roots, transformed polynomials and elements reduce computational costs associated with Cipolla's algorithm. In their transformed state, degree q polynomial components can be chosen to increase the probability of it being irreducible. Furthermore, a dimension $2q$ discrete Fourier transform enables us to perform multiplication in \mathbb{F}_{P^q} in parallel. Although polynomial multiplication using discrete Fourier transforms is fairly standard (see [1]), the reduction modulo $(x^q - u)$ is not. Generally the polynomial can not be reduced without transforming the polynomial back to its original state. The algorithm in section 9.4 describes a technique for reduction which does not require the inverse transformation.

9.1 Simplified conjugate formula

Before describing the Fourier transforms on root mapping polynomials and elements in \mathbb{F}_{P^q} , the simplified representations of conjugates must be proved. The following lemma proves that η_i , as in equation (9.0.2) is the i -th conjugate of η : $\eta_i = \eta^{P^i}$.

Lemma 9.1.1. *Let P, q, u be defined as in table 7.1 with $h = u^{\frac{P-1}{q}}$. If $\eta \in \mathbb{F}_P[x]/(x^q - u)$ with $\eta = \sum_{j=0}^{q-1} r_j x^j$, then*

$$\eta_k = \eta^{P^k} \equiv \sum_{j=0}^{q-1} r_j h^{kj} x^j$$

is the k -th conjugate of η .

Proof. Since $\text{ord}(u) | q^n$ and $q^n | \text{ord}(u)$ we know that $\text{ord}(h) = q$, and h is a primitive q^{th} root of unity. The conjugates of η are $\eta_k = \eta^{P^k}$ for $0 \leq k < q$, or

$$\eta^{P^k} \equiv \sum_{j=0}^{q-1} r_j x^{jP^k}$$

Since $x^{P-1} \equiv (x^q)^{P-1/q} \equiv u^{P-1/q} = h \pmod{(x^q - u)}$, the equation reduces to:

$$\begin{aligned} \eta_k &\equiv \sum_{j=0}^{q-1} r_j x^{j(P^k-1)+j} \\ &\equiv \sum_{j=0}^{q-1} r_j h^{j\left(\frac{P^k-1}{P-1}\right)} x^j \end{aligned}$$

From lemma 2.1.4 we know that $\frac{P^k-1}{P-1} \equiv k \pmod{q}$, therefore

$$\eta_k \equiv \sum_{j=0}^{q-1} r_j h^{jk} x^j.$$

□

9.2 Fourier transforms over \mathbb{F}_P on root mapping polynomials and elements in \mathbb{F}_{P^q}

The discrete Fourier transform, generally used in signal processing and for convolutions, is defined over the complex numbers as the function $\mathcal{F}: \mathbb{C}^n \rightarrow \mathbb{C}^n$

$$\mathcal{F}([a_k]_{k=0}^{n-1}) = [\mathcal{F}_j]_{j=0}^{n-1}, \quad (9.2.1)$$

where $h = e^{-\frac{2\pi i}{n}}$ is a primitive n -th root of unity in \mathbb{C} and $\mathcal{F}_j = \sum_{k=0}^{n-1} a_k h^{jk}$.

The inverse discrete Fourier transform is defined as:

$$\mathcal{F}^{-1}([\mathcal{F}_j]_{j=0}^{n-1}) = \left[\frac{1}{n} \sum_{j=0}^{n-1} \mathcal{F}_j h^{-jk} \right]_{k=0}^{n-1}. \quad (9.2.2)$$

These formulas and their variants are described in [1], [23], [12], and many others. The formula over any field F is identical, except a primitive n -th root of unity in F , instead of in \mathbb{C} , is used.

The discrete Fourier transform of dimension n over the field F using the primitive n^{th} root of unity $h \in F$ is equivalent to the Chinese remainder theorem mapping

$$\Psi: F[x]/(x^n - 1) \rightarrow \bigotimes_{j=0}^{n-1} F[x]/(x - h^j). \quad (9.2.3)$$

This mapping was discussed in section 2.3 and generalized over ideals in a commutative ring in [25] and [19]. An n dimensional vector $A \in F^n$ can also represent polynomials in $F[x]/(x^n - 1)$:

$$A = [a_j]_{j=0}^{n-1}$$

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Reducing $A(x)$ modulo $(x - h^j)$ is equivalent to $A(x)$ evaluated at h^j or the j -th

term of $\mathcal{F}(A)$:

$$\begin{aligned}\mathcal{F}_j(A) &= \sum_{k=0}^{n-1} a_k h^{jk} \\ &\equiv A(x) \pmod{x - h^j} \\ &= A(h^j)\end{aligned}\tag{9.2.4}$$

Since the degree of $A(x) < n$ and $\prod_{j=0}^{n-1} (x - h^j) = (x^n - 1)$

$$\Psi(A(x)) = [\mathcal{F}_j]_{j=0}^{n-1}.$$

The notation $\mathcal{F}(A(x))$ will be used to denote this function:

$$\mathcal{F}(A(x)) = \Psi(A(x)) = [\mathcal{F}_j]_{j=0}^{n-1}.$$

Root mapping polynomials and elements in $\mathbb{F}_P[x]/(x^q - u)$ are expressed as polynomials over \mathbb{F}_P , therefore they can be reduced modulo $(x^q - 1)$ and represented using the vector obtained with the mapping in equation (9.2.3). Elements in $\mathbb{F}_P[x]/(x^q - u)$ in reduced form have degree less than q , therefore no information is lost in this transformation, however root mapping polynomials have degree q and will be reduced modulo $(x^q - 1)$. Let $f(x) = x^q + \sum_{i=0}^{q-1} c_i x^i$ be a root mapping polynomial in $\mathbb{F}_P[x]$. The discrete Fourier transform of $f(x)$, with initial reduction, is:

$$\begin{aligned}f(x) &\equiv \sum_{i=0}^{q-1} c_i x^i + 1 \pmod{x^q - 1} \\ \mathcal{F}(f(x)) &= \left[\sum_{i=0}^{q-1} c_i h^{ik} + 1 \pmod{x - h^k} \right]_{k=0}^{q-1}.\end{aligned}$$

The inverse transform in equation (9.2.2) will return $f(x) \pmod{x^q - 1}$, so $f(x)$ can be completely recovered by

$$f(x) = \mathcal{F}^{-1}(\mathcal{F}(f(x))) - 1 + x^q.\tag{9.2.5}$$

The purpose of applying discrete Fourier transforms to root mapping polynomials and elements in \mathbb{F}_{P^q} is to give an alternate form of polynomial and element independent of either the q^{th} root it returned by Cipolla's algorithm or which conjugate it was. If $A(x) = \sum_{i=0}^{q-1} a_i x^i$ is in $\mathbb{F}_P[x]$ (either an element in $\mathbb{F}_P[x]/((x^q - u))$ or a root mapping polynomial reduced modulo $(x^q - 1)$) and $A_k(x) = \sum_{i=0}^{q-1} a_i h^{ik} x^i$ then $\mathcal{F}_j(A_k(x))$ is

$$\begin{aligned}\mathcal{F}_j(A_k) &= \sum_{i=0}^{q-1} (a_i h^{ik}) h^{ij} \\ &= \sum_{i=0}^{q-1} a_i h^{i(j+k)}.\end{aligned}$$

This equation implies that the discrete Fourier transform of A_k is just the transform of A cyclicly shifted k places.

Theorem 9.2.1. *Let P, q, u be defined as in table 7.1 with $h = u^{P-1/q}$. If $A \in \mathbb{F}_P[x]$ with $A(x) = \sum_{j=0}^{q-1} a_j x^j$ and $A_k = \sum_{i=0}^{q-1} a_i h^{ik}$ then*

$$\mathcal{F}(A_k) = [\mathcal{F}_{j+k \bmod q}(A)]_{j=0}^{q-1}$$

where $\mathcal{F}_j(A) = \sum_{i=0}^{q-1} a_i h^{ij}$, the j -th component of $\mathcal{F}(A)$.

Proof. From the proof of lemma 9.1.1 we know that $h = u^{P-1/q}$ is a primitive q^{th} root of unity.

$$\begin{aligned}\mathcal{F}(A_k) &= [\mathcal{F}_j(A_k)]_{j=0}^{q-1} \\ &= \left[\sum_{i=0}^{q-1} (a_i h^{ik}) h^{ij} \right]_{j=0}^{q-1} \\ &= \left[\sum_{i=0}^{q-1} a_i h^{i(k+j \bmod q)} \right]_{j=0}^{q-1} \\ &= [\mathcal{F}_{j+k \bmod q}(A)]_{j=0}^{q-1}\end{aligned}$$

□

Theorem 9.2.1 tells us that a k -cyclic shift of the transformed vector, reduced root mapping polynomial f or element η , produces the transform of f_{q-k} (with the adjustments in equation (9.2.5)) or η_k . These equations can be used to generate the alternate polynomials or conjugates and introduces a new form in which to study the polynomials and elements.

9.3 Properties of Fourier transformed root mapping polynomials and their roots

Given a q^{th} residue $w \in \mathbb{F}_P$, and a root mapping polynomial f with norm $N(f) = w$, Cipolla's algorithm generates a specific q^{th} root: $\mathfrak{C}(f) = a$. The uncertainty of which root would be returned causes problems in the proof of equality between the discrete logarithm and q^{th} root problems. The discrete Fourier transform gives us a way to examine a set of elements from which all q^{th} roots can be generated. This section examines some of the traits of root mapping polynomials in their discrete Fourier transformed state.

There are two traits we can use to search for transformed root mapping polynomials. Let $f(x) = x^q + \sum_{i=0}^{q-1} c_i x^i$. If the element whose q^{th} root we are searching for is w , then $c_0 = (-1)^q w$. Since $c_0 = \frac{1}{q} (\sum_{i=0}^{q-1} \mathcal{F}_i(f)) - 1$, choosing any $(q-1)$ of the terms $\{\mathcal{F}_i(f)\}$ uniquely determines the remaining term. Furthermore, a necessary condition for $f(x)$ to be irreducible is that none of the elements in $\{\mathcal{F}_i(f)\}$ is equivalent to 0.

Lemma 9.3.1. *No coefficient of a discrete Fourier transformed root mapping polynomial is equivalent to 0.*

Proof. Let $f(x)$ be a root mapping polynomial for computing q^{th} roots in \mathbb{F}_P . The discrete Fourier transform of $f(x)$ is $\mathcal{F}(f) = [\mathcal{F}_j(f) = f(h^j)]_{j=0}^{q-1}$. Therefore

if $\mathcal{F}_j(f) = 0$, this implies that $(x - h^j) \mid f$, which implies $f(x)$ is not irreducible and $f(x)$ is not a root mapping polynomial. Therefore no term of the discrete Fourier transformed $f(x)$ can be 0. \square

These two restrictions may be useful for generating root mapping polynomials as they reduce the search space for irreducible polynomials. Most current irreducibility tests for polynomials $f(x)$ of degree q over \mathbb{F}_P check that f has order dividing $P^q - 1$ but not $P - 1$; the first stage checks that $f(x)$ has no roots in \mathbb{F}_P (its order does not divide $P - 1$). Fourier transformed polynomials with no zero terms automatically gives us that h^j , for $0 \leq j < q$, is not a root of $f(x)$. However there are currently no known techniques for irreducibility testing polynomials in their transformed state beyond checking that no q^{th} root of unity is a root. Any practical use of transformed polynomials in generating root mapping polynomials must transform the polynomials back to polynomial state to finish the testing.

The discrete Fourier transform of the roots of these associated root mapping polynomials are also closely related. Recall from section 9.1 that in their transformed state the roots of a particular root mapping polynomial in $\mathbb{F}_P[x]/((x^q - u))$ are simple cyclic shifts of one another. If $f(x)$ is a root mapping polynomial with root η , then the alternate polynomials are $f_i(x) = f(h^{-i}x)$ with root ηh^i (lemma 7.2.3). Therefore the set

$$\left\{ [\mathcal{F}_{i+k}(h^j \eta)]_{i=0}^{q-1} \mid 0 \leq j, k < q \right\}$$

is a transformed set of q^2 elements in \mathbb{F}_{P^q} which compute the q^{th} root of w .

9.4 Multiplication in \mathbb{F}_{P^q} using transformed elements

This section describes a technique for performing multiplication in \mathbb{F}_{P^q} using a dimension $2q$ discrete Fourier transform. Recall from section 9.2 that discrete

Fourier transforms can be viewed as an application of the Chinese remainder theorem, mapping $\mathbb{F}_P[x]/(x^n - 1)$ to $\bigotimes_{i=0}^{n-1} \mathbb{F}_P[x]/(x - h^i)$ where $h \in \mathbb{F}_P$ is a primitive n^{th} root of unity. This was used in the Schönhage-Strassen multiplication technique described in 7.5 of [1]. As long as the resulting degree is less than the dimension of the transform, operations in $\mathbb{F}_P[x]$ can be performed. Elements in $\mathbb{F}_P[x]/(x^q - u)$ are polynomials in $\mathbb{F}_P[x]$ with degree less than q and the product of any two elements, before reduction modulo $(x^q - u)$, have degree less than $2q$. Therefore a discrete Fourier transform of dimension $2q$ allows multiplication of any two reduced elements in $\mathbb{F}_P[x]/(x^q - u)$ without loss of information. If \mathbb{F}_P has odd characteristic and q is an odd prime, then a $2q$ -th primitive root of unity is guaranteed to exist. The only difficulty is the reduction.

Let $\alpha, \beta \in \mathbb{F}_P[x]/(x^q - u)$, $\psi \in \mathbb{F}_P$ be a $2q$ -th primitive root of unity. Define

$$\Gamma \equiv \alpha\beta(x^q + u) \equiv \sum_{i=0}^{2q-1} c_i x^i \pmod{x^{2q} - 1} \quad (9.4.1)$$

which is explained further in equations (9.4.2) and (9.4.3). This section will show that $\alpha\beta \equiv \sum_{i=0}^{q-1} c_{i+q} x^i \pmod{x^q - u}$ and describe a technique for converting $\Gamma \in \prod_{i=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^i)$ into $\bar{\Gamma} \in \prod_{i=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^i)$ where $\bar{\Gamma} = \sum_{i=0}^{q-1} c_{i+q} x^i \equiv \alpha\beta \pmod{x^q - u}$. Notice that

$$\Gamma \equiv [\mathcal{F}_j(\alpha) \mathcal{F}_j(\beta) \mathcal{F}_j(x^q + u)]_{j=0}^{2q-1}$$

which implies that two dimension $2q$ vector multiplications over \mathbb{F}_P are required. This is much less than the standard q^2 multiplications needed to perform a standard polynomial multiplication. But what is gained in the multiply is lost in the reduction. The reduction technique described here requires q multiplications in $\bigotimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$, or q dimension $2q$ vector multiplies.

The value of Γ is created using the Chinese remainder theorem as in section 2.3 and the fact that $(x^{2q} - 1) \equiv (u^2 - 1) \pmod{x^q - u}$. With these two facts we

can create a polynomial congruent to 0 modulo $(x^{2q} - 1)$ and $\alpha\beta(x^{2q} - 1)$ modulo $(x^q - u)$:

$$C(\alpha\beta) = \alpha\beta(u^2 - 1) + (x^q - u) \left(-(u^2 - 1)(x^q - u)^{-1} \alpha\beta \bmod (x^{2q} - 1) \right) \quad (9.4.2)$$

with degree less than $3q$. Dividing $C(\alpha\beta)$ by $(x^{2q} - 1)$ therefore reduces the degree to a value less than q and returns a polynomial equivalent to $\alpha\beta \bmod x^q - u$:

$$\begin{aligned} \frac{C(\alpha\beta)}{(x^{2q} - 1)} &= \frac{\alpha\beta(u^2 - 1) + (x^q - u) \left((x^q + u) \alpha\beta \bmod x^{2q} - 1 \right)}{(x^{2q} - 1)} \\ &= \frac{\alpha\beta(u^2 - 1) + (x^q - u)\Gamma}{(x^{2q} - 1)}. \end{aligned} \quad (9.4.3)$$

The degree of $\Gamma \in \mathbb{F}_P[x]/(x^{2q} - 1)$ is bounded by $2q$ as is $\Theta = \alpha\beta(u^2 - 1) \in \mathbb{F}_P[x]$. Letting $\Theta = \alpha\beta(u^2 - 1) = \sum_{i=0}^{2q-1} d_i x^i$ gives us that

$$\begin{aligned} C(\alpha\beta) &= \Theta + (x^q - u)\Gamma \\ &= \sum_{i=2q}^{3q-1} c_{i-q} x^i + \sum_{i=q}^{2q-1} (c_{i-q} - u c_i + d_i) x^i + \sum_{i=0}^{q-1} (d_i - u c_i) x^i \end{aligned}$$

and since $C(\alpha\beta) \equiv 0 \bmod x^{2q} - 1$, this implies that

$$\frac{C(\alpha\beta)}{x^{2q} - 1} = \sum_{i=0}^{q-1} c_{i+q} x^i. \quad (9.4.4)$$

One way to compute $\alpha\beta \bmod x^q - u$ would be to compute the individual components of $\Gamma \in \bigotimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$, convert it back to a polynomial in $\mathbb{F}_P[x]/(x^{2q} - 1)$, subtract off the lower q terms, divide by x^q , then convert it back to an element in $\bigotimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$. However the goal is to remain in $\bigotimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$ and spare the cost of conversion. The algorithm which follows does essentially the same procedure but remains in $\bigotimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$. If done in serial, the number of multiplies in \mathbb{F}_P is approximately the same number needed to perform one transform. Minimal communication is required when implemented in parallel.

The conversion of $\Gamma \equiv \sum_{i=0}^{2q-1} c_i x^i$ into $\bar{\Gamma} \equiv \sum_{i=0}^{q-1} c_{i+q} x^i$ is done in q steps. Starting at step $i = 0$, the current constant term, c_i is computed, subtracted off, then the remaining polynomial multiplied by x^{-1} . This makes c_{i+1} the new constant term for the next step of the algorithm. After q steps, the low order q terms will have been subtracted and the polynomial multiplied by x^{-q} .

In the following algorithm, vector notation is used to indicate an element in $\otimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$ with operations on these elements being standard component wise vector multiply, add, and subtract. For example, if $[b_j], [a_j] \in \otimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$ then $[b_j] \otimes [a_j] = [b_j a_j]$.

Algorithm 9.4.1: Polynomial DFT reduction

Input: $\Gamma = [\gamma_j]_{j=0}^{2q-1} \equiv \sum_{i=0}^{2q-1} c_i x^i \pmod{x^{2q} - 1}$

Output: $\bar{\Gamma} = [\bar{\gamma}_j]_{j=0}^{2q-1} \equiv \sum_{i=0}^{q-1} c_{i+q} x^i \pmod{x^{2q} - 1}$

- 1: $\Gamma^{(0)} = \left[\gamma_j^{(0)} \right]_{j=0}^{2q-1} = \Gamma$
- 2: For $i = 0$ to $q - 1$:
 - i: $c_i = (2q)^{-1} \sum_{j=0}^{2q-1} \gamma_j^{(i)}$
 - ii: $\Gamma^{(i+1)} = [\psi^{-j}]_{j=0}^{2q-1} \otimes \left(\Gamma^{(i)} \ominus [c_i]_{j=0}^{2q-1} \right)$
- 3: return $\bar{\Gamma} = \Gamma^{(q)}$.

Step i of the algorithm is computes $\mathcal{F}_0^{-1}(\Gamma^{(i)})$, producing the current constant term. Step ii subtracts off the constant term and multiplies the result by $x^{-1} \in \otimes_{j=0}^{2q-1} \mathbb{F}_P[x]/(x - \psi^j)$. Therefore $\Gamma^{(i)} \equiv \sum_{j=i}^{2q-1} c_j x^{j-i}$, and $\Gamma^{(q)} \equiv \sum_{j=0}^{q-1} c_{j+q} x^j$.

Example 9.4.2 – DFT polynomial multiplication: Let $P = 101$,

$q = 5$, $u = 16$ and $\psi = 14$ with the powers of ψ and ψ^{-1} being:

j	9	8	7	6	5	4	3	2	1	0
ψ^j	65	84	6	87	100	36	17	95	14	1
$\mathcal{F}(x^{-1})$	14	95	17	36	100	87	6	84	65	1

The example goes through multiplication of α and β :

$$\alpha = x^4 + 3x^3 + 0x^2 + 2x + 91 \text{ mod } x^q - u$$

$$\beta = 3x^4 + 82x^3 + 0x^2 + 79x + 29 \text{ mod } x^q - u$$

The elements in their transformed states and the resulting product, Γ are given below:

$$\mathcal{F}(\alpha) = [23 \ 58 \ 27 \ 48 \ 87 \ 30 \ 11 \ 20 \ 4 \ 97]$$

$$\mathcal{F}(\beta) = [59 \ 4 \ 85 \ 61 \ 73 \ 16 \ 18 \ 73 \ 11 \ 92]$$

$$\mathcal{F}(x^q + u) = [15 \ 17 \ 15 \ 17 \ 15 \ 17 \ 15 \ 17 \ 15 \ 17]$$

$$\mathcal{F}(\Gamma) = [54 \ 5 \ 85 \ 84 \ 22 \ 80 \ 41 \ 75 \ 54 \ 6]$$

Each of the next q steps removes c_i and divides by x . Note that

$$(2q)^{-1} \equiv 91 \pmod{P}:$$

$$\begin{array}{l}
\Gamma^{(0)} \equiv [54 \ 5 \ 85 \ 84 \ 22 \ 80 \ 41 \ 75 \ 54 \ 6] \\
\hline
i = 0 \qquad c_0 = 91(54 + 5 + 85 + 84 + 22 + 80 + 41 + 75 + 54 + 6) \\
\qquad \qquad \qquad \equiv 91 \pmod{101} \\
\bar{\Gamma}^{(0)} = \Gamma^{(0)} - 91 \equiv [64 \ 15 \ 95 \ 94 \ 32 \ 90 \ 51 \ 85 \ 64 \ 16] \\
\Gamma^{(1)} = \bar{\Gamma}^{(0)} x^{-1} \equiv [88 \ 11 \ 100 \ 51 \ 69 \ 53 \ 3 \ 70 \ 19 \ 16] \\
\hline
i = 1 \qquad c_1 = 91(76) \equiv 48 \pmod{101} \\
\bar{\Gamma}^{(1)} = \Gamma^{(1)} - 48 \equiv [40 \ 64 \ 52 \ 3 \ 21 \ 5 \ 56 \ 22 \ 72 \ 69] \\
\Gamma^{(2)} = \bar{\Gamma}^{(1)} x^{-1} \equiv [55 \ 20 \ 76 \ 7 \ 80 \ 31 \ 33 \ 30 \ 34 \ 69] \\
\hline
i = 2 \qquad c_2 = 91(31) = 94 \\
\bar{\Gamma}^{(2)} = \Gamma^{(2)} - 94 \equiv [62 \ 27 \ 83 \ 14 \ 87 \ 38 \ 40 \ 37 \ 41 \ 76] \\
\Gamma^{(3)} = \bar{\Gamma}^{(2)} x^{-1} \equiv [60 \ 40 \ 98 \ 100 \ 14 \ 74 \ 38 \ 78 \ 39 \ 76] \\
\hline
i = 3 \qquad c_3 = 91(11) = 92 \\
\bar{\Gamma}^{(3)} = \Gamma^{(3)} - 92 \equiv [69 \ 49 \ 6 \ 8 \ 23 \ 83 \ 47 \ 87 \ 48 \ 85] \\
\Gamma^{(4)} = \bar{\Gamma}^{(3)} x^{-1} \equiv [57 \ 9 \ 1 \ 86 \ 78 \ 50 \ 80 \ 36 \ 90 \ 85] \\
\hline
i = 4 \qquad c_4 = 91(67) = 37 \\
\bar{\Gamma}^{(4)} = \Gamma^{(4)} - 37 \equiv [20 \ 73 \ 65 \ 49 \ 41 \ 13 \ 43 \ 100 \ 53 \ 48] \\
\Gamma^{(5)} = \bar{\Gamma}^{(4)} x^{-1} \equiv [78 \ 67 \ 95 \ 47 \ 60 \ 20 \ 56 \ 17 \ 11 \ 48] \\
\hline
\end{array}$$

The inverse DFT on $\Gamma^{(5)}$ is:

$$\begin{aligned}
\Gamma^{(5)} &\equiv 97x^4 + 22x^3 + 99x^2 + 73x + 29 \pmod{x^{2q} - 1} \\
&\equiv \alpha\beta \pmod{x^q - u}
\end{aligned}$$

Chapter 10

Conclusions

The q^{th} root problem was suggested as a basis for public key cryptosystems in [4]. Several designs were described in [4] and a single pass authenticated key exchange provably secure against man-in-the-middle attack was described in [22]. The general nature of the problem enables these systems to be designed over \mathbb{F}_P , elliptic curves (see [6]), or any group containing a suitably sized subgroup of order q^2 for which the discrete logarithm problem is difficult.

q^{th} root cryptosystems are based on the assumption that the difficulty of the q^{th} root problem is equivalent to that of the discrete logarithm problem. This equivalence is suggested by the fact that the most efficient q^{th} root techniques over a general finite cyclic group of order q^n , where q is prime and $n > 1$, require a discrete logarithm computation. These q^{th} root techniques are equivalent to, or are special cases of, algorithm 4.2.1. Further evidence linking the two problems was given in [4] and chapter 5 in the form of an algorithm. The algorithm computes discrete logarithms using a well-behaved q^{th} root oracle. The ability to compute discrete logarithms enables q^{th} roots to be computed, and if a well-behaved q^{th} root oracle existed, then discrete logarithms could be computed using algorithm 5.2.1, reinforcing the hypothesis that the two problems are equivalent.

A proof of equivalence between two problems generally assumes that one problem is solvable, then uses that solution to solve the second problem. Unfortunately the q^{th} root mapping is ill defined (definition 5.1.1) and there is no known, well defined q^{th} root function over general finite cyclic groups which does not require discrete logarithm computations. Equivalence was shown for the two problems (chapter 5 and in [4]) only after the well-behaved assumptions were made about the behaviour of the q^{th} root mapping.

Although no general purpose q^{th} root algorithm independent of discrete logarithm computations exists, one special purpose algorithm does: Cipolla's algorithm. This algorithm works only over a finite field and requires the use of a degree q extension field. To find the q^{th} root of w , an irreducible degree q polynomial with norm w is generated. The type of root returned by the algorithm depends entirely on this polynomial. Current techniques for finding these polynomials are random (see appendix B.1), therefore the algorithm returns a random q^{th} root.

If Cipolla's algorithm was more efficient than currently known discrete logarithm techniques it would show that, with current technologies, the q^{th} root problem was computationally less expensive than the discrete logarithm problem. If the algorithm was not only more efficient than a discrete logarithm computation but was well-behaved, it would give us another way to compute discrete logarithms. However the algorithm is not less expensive: its current form is far more costly than computing discrete logarithms for any cryptographically secure value of q . The goal of this research was to analyze Cipolla's algorithm to determine if its run time could be reduced or if it could be modified to return well-behaved q^{th} roots.

After finding a degree q irreducible polynomial with norm w , Cipolla's algorithm raises one of the polynomials roots, $\eta \in \mathbb{F}_{P^q}$, to the K power: $\mathfrak{C}(\eta) = \eta^K$.

Since the q^{th} residues we are interested in are in $G_{q^n} \subset \mathbb{F}_P$, the element $\eta \in G_{q^n K}$. Using the Chinese remainder theorem, η can be represented as an element in $G_{q^n} \times G_K$: $\widehat{\Psi}(\eta) = (\eta_{q^n}, \eta_K)$ where $\eta_{q^n} \in G_{q^n} \subset \mathbb{F}_P$ and $\eta_K \in G_K \subset \mathbb{F}_{P^q} \setminus \mathbb{F}_P$. Section 7.2 shows that η_{q^n} is exactly the q^{th} root generated by the algorithm: $\mathfrak{C}(\eta) = \eta_{q^n}$.

The very large exponentiation required by the algorithm removes the portion of the root exclusively in the extension field (η_K), leaving the q^{th} root untouched (η_{q^n}). Ignoring the cost of finding this polynomial, the main expense of the algorithm comes from the size of the exponentiation. Reducing the cost of Cipolla's algorithm implies reducing the size of this exponentiation, which implies generating a polynomial with low order η_K .

Generating root mapping polynomials (the polynomials needed for Cipolla's algorithm) with low order η_K portions equates to finding a polynomial in a low order equivalence class, with the equivalence class order defined in 7.3.4. If a root mapping polynomial and the representative polynomial for its class could be found, then section 7.4 showed that the q^{th} root would be revealed from the coefficients of the two polynomials, making Cipolla's algorithm extraneous. Furthermore, if a polynomial from the lowest possible order equivalence class were known the q^{th} root would be revealed from the $(q-1)^{th}$ coefficient of the polynomial and the trace formulas derived in section 8.3. Since $-Tr(\eta)$ is the $(q-1)^{th}$ coefficient of the polynomial and there are only two equivalence classes with this order, the existence of minimal order Cipolla's algorithm implies that the q^{th} root is already known.

Cipolla's algorithm reveals little about the connection between the q^{th} root and discrete logarithm problems, however it points to new directions for research into these problems. We discovered a simple formula for the trace of minimal order

elements in $G_K \subset \mathbb{F}_{P^q}$. If these equations could be extended to other low order elements in $\mathbb{F}_{P^q} \setminus \mathbb{F}_P$ we may be able to use this to compute q^{th} roots.

Another area which may deserve further research is the analysis of Fourier transformed elements and polynomials. The transformed polynomial returned a set which, when viewed as a cyclicly ordered set of q elements in \mathbb{F}_P , represented a root mapping polynomial independent of the root it generates. It may be possible to exploit adjacency relationships for irreducible polynomial generation.

Appendix A

Tables

A.1 Examples of prime (q, P) pairs for which $q^2 \mid (P - 1)$

Table A.1 gives primes for which q^2 divides $(P - 1)$. The table is divided into sections for $q = 3, 5, 7, 11, 13, 19$.

A.2 Examples of fields with minimal order root mapping polynomial equivalence class

Table A.2 contains primes P such that $q^2 \mid (P - 1)$ and $2q + 1 \mid P^q - 1$. Minimal order equivalence class (order $R = 2q + 1$, as in definition 7.3.6 on page 95) exist for computing q^{th} roots in \mathbb{F}_P . Formulas the representative polynomials for $(q, R) = (11, 23)$ are given in section 8.5.

Prime	P-1	g	K
19	$3^2 2^1$	2	127
127	$3^2 2^1 7^1$	3	5419
5419	$3^2 2^1 7^1 43^1$	3	$31 \cdot 313 \cdot 1009$
1009	$3^2 7^1 2^4$	11	$37 \cdot 9181$
37	$3^3 2^2$	2	$7 \cdot 67$
73	$3^2 2^3$	5	1801
109	$3^3 2^2$	6	$7 \cdot 571$
163	$3^4 2^1$	2	$7 \cdot 19 \cdot 67$
181	$3^2 2^2 5^1$	2	$7 \cdot 1093$
101	$5^2 2^2$	2	$1381 \cdot 31 \cdot 491$
151	$5^2 3^1 2^1$	6	(prime)
1801	$5^2 3^2 2^3$	11	$101 \cdot (\text{prime})$
197	$7^2 2^2$	2	$29 \cdot 97847 \cdot 2957767$
491	$7^2 2^1 5^1$	2	$617 \cdot 1051 \cdot 3093060713$
883	$7^2 2^1 3^2$	2	$379 \cdot 3389 \cdot (\text{prime})$
727	$11^2 2^1 3^1$	5	$6972321 \cdot (\text{prime})$
1453	$11^2 2^2 3^1$	2	$23 \cdot (\text{composite})$
677	$13^2 2^4$	2	(prime)
2029	$13^2 3^1 2^2$	2	$53^2 \cdot 65677 \cdot (\text{prime})$
3719	$13^2 11^1 2^1$	2	$157 \cdot 188137 \cdot (\text{composite})$
10831	$19^2 5^1 3^1 2^1$	7	$1787 \cdot 2053 \cdot 29917819 \cdot (\text{prime})$

Table A.1: Small test primes with $q^2 \mid P - 1$

$q \setminus R$	$11 \setminus 23$	$23 \setminus 47$	$41 \setminus 83$	$83 \setminus 167$
	11617	205253	20173	7316119
	1453	14813	1260751	21562571
	3389	31741	319391	1736029
	9439	165049	181549	21907021
	2663	33857	188273	20832337

Table A.2: Primes for which minimal order equivalence class exist

Appendix B

Other Algorithms

B.1 Degree q polynomial irreducibility testing in \mathbb{F}_P

The first step in computing the q^{th} root of $w \in \mathbb{F}_P$ with Cipolla's algorithm is to find an irreducible polynomial in \mathbb{F}_P which has constant term $(-1)^q w$. In general this implies testing random (except for the constant term) degree q polynomials for irreducibility. The irreducibility tests given here are specialized variants of those discussed in [16]. Other techniques can be found in [42]. They have been adapted for polynomials with prime degree and incorporated for the final use of the polynomial: to compute the q^{th} root of an element in \mathbb{F}_P using Cipolla's algorithm.

Let P, q, n, K be defined as in table 2.1, with q^{th} residue $w \in \mathbb{F}_P$. If a randomly generated monic polynomial, $f(x) = \sum_{i=0}^q c_i x^i$, with $c_0 = (-1)^q w$ is irreducible, it has a root $\eta \in \mathbb{F}_{P^q} \setminus \mathbb{F}_P$ with $\eta^{qK} = w$ and $\eta^K \in \mathbb{F}_P$. If f is not irreducible, then $f(x) = \prod_{j=0}^d h_j^{e_j}(x)$ with $h_j(x)$ irreducible polynomials of degree $0 < \deg(h_j) < q$, $e_j \in \mathbb{N}$ and $\sum_{j=0}^d e_j \deg(h_j) = q$. The order of a root a degree $i > 0$ irreducible polynomial in \mathbb{F}_P must divide $(P^i - 1)$. Since q is prime we know from lemma 2.1.4 that $\gcd(K, P^i - 1) = 1$ for $0 \leq i < q$. Therefore if η is a root of a composite f , its order must be relatively prime to K . If $\eta \notin \mathbb{F}_P$, then

$\eta^K \notin \mathbb{F}_P$. However if f is an irreducible polynomial with root η , then $\eta^K \in \mathbb{F}_P$.

If the polynomial is irreducible, this test also produces the q^{th} root.

Algorithm B.1.1: Attempt to compute $\mathfrak{C}(f) = (w)^{\frac{1}{q}}$ with appropriately generated random f

Input: $f(x) = \sum_{i=0}^q c_i x^i$ with $c_q = 1$, $c_0 = (-1)^q w$ and $w^{q^{n-1}s} = 1$

Output: q^{th} root or FAIL

- 1: Compute $b = x^{P-1} \bmod f$;
- 2: If $\gcd(f, b-1) \neq 1$ return FAIL;
- 3: Compute $a = x^K \bmod f \in \mathbb{F}_P$;
- 4: If $a \in \mathbb{F}_P$: return a .
- 5: else return FAIL.

The first two steps of the algorithm checks to see if any degree 1 polynomials divide f . These first two steps can be skipped. This is not cost effective for large q as more of the very expensive exponentiations by K would be required. Since the polynomials are being sought to solve the q^{th} root problem and the end result is the q^{th} root, we can assume no polynomials of degree 1 divide f , compute what might be the q^{th} root, and if the result is in the base field check to see that it is a q^{th} root: $a^q \equiv w$. If it does we have computed a q^{th} root and if not $f(x)$ was not irreducible.

Algorithm B.1.2: Simplified attempt to compute $\mathfrak{C}(f) = (w)^{\frac{1}{q}}$ with appropriately generated random f

Input: $f(x) = \sum_{i=0}^q c_i x^i$ with $c_q = 1$, $c_0 = (-1)^q w$ and $w^{q^{n-1}s} = 1$

Output: q^{th} root or FAIL

- 1: Compute $a = x^K \bmod f$;
- 2: If $a \notin \mathbb{F}_P$: return FAIL;
- 3: else if $a^q = w$ return a .
- 4: else FAIL;

The probability of choosing an irreducible polynomial out of the set

$$\left\{ \sum_{i=0}^q c_i x^i \mid c_0 = (-1)^q w, c_q = 1, c_i \in \mathbb{F}_P \forall 0 < i < q \right\}$$

is approximately $\frac{1}{q}$ when P, q are large. There are P^{q-1} polynomials in this set, and only $(K - 1)$ of them are irreducible (theorem 6.2.1). The probability of hitting an irreducible polynomial at random is:

$$\begin{aligned} \frac{K - 1}{P^{q-1}} &= \frac{K - 1}{(q^{n+1} s K + 1) / (q^n s + 1)} \\ &\approx \frac{K - 1}{qK} \\ &\approx \frac{1}{q} \end{aligned}$$

Therefore adding the computational work required by irreducibility testing increases Cipolla's algorithm computation by a factor of q on average.

Bibliography

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1974.
- [2] Eric Bach and Jeffrey Shallit, *Algorithmic number theory - volume 1: Efficient algorithms*, ch. 7, MIT, 1997.
- [3] ———, *Algorithmic number theory - volume 1: Efficient algorithms*, second ed., MIT, 1997.
- [4] Cheryl Beaver, Peter Gemmell, Anna Johnston, and William Newmann, *On the cryptographic value of the q^{th} root problem*, Proceedings of the International Conference on Information and Computer Security, Lecture Notes in Computer Science, Springer, 1999, Sydney, Australia, pp. 135–142.
- [5] E. Brickell, D. Gordon, K. McCurley, and D. Wilson, *Fast exponentiation with precomputation: Algorithms and lower bounds*, Tech. report, Sandia National Laboratories, 1992.
- [6] R. Broker and P. Stevenhagen, *Elliptic curves with a given number of points*, Algorithmic Number Theory, VI International Symposium, Springer-Verlag New York, Inc., 2004, pp. 117–131.
- [7] M. Cipolla, *Un metodo per la risoluzione della congruenza di secondo grado*, Rendiconto dell'Accademia delle Scienze Fisiche e Matematiche **9** (1903), 154–163.

- [8] D. Coppersmith, A.M. Odlyzko, and R. Schroepfel, *Discrete logarithms in $gf(p)$* , *Algorithmica* (1986), no. 1, 1–15.
- [9] Whit Diffie and Martin Hellman, *New directions in cryptography*, *Transactions on Information Theory*, no. 22, IEEE, November 1976, pp. 644–654.
- [10] ———, *Exhaustive cryptanalysis of the nbs data encryption standard*, *Computer* (1977), no. 10, 74–84.
- [11] *Federal Information Processing Standards Publication 186-2: Announcing the Digital Signature Standard (DSS)*, January 2000.
- [12] Richard C. Dorf (ed.), *The electrical engineering handbook*, second ed., CRC Press, 2000 Corporate Blvd., N.W; Boca Raton, Florida 33431, 1997.
- [13] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, *Advances in Cryptology – CRYPTO, Lecture Notes in Computer Science*, no. 196, 1984.
- [14] U. Feige, A. Fiat, and A. Shamir, *Zero-knowledge proofs of identity*, *Journal of Computing*, vol. 1, 1988.
- [15] John B. Fraleigh, *A first course in abstract algebra*, 3 ed., Addison-Wesley Publishing Co., Reading, Massachusetts, 1982.
- [16] Shuhong Gao and Daniel Panario, *Tests and constructions of irreducible polynomials over finite fields*, *FoCM '97: Selected papers of a conference on Foundations of computational mathematics (New York, NY, USA)*, Springer-Verlag New York, Inc., 1997, pp. 346–361.
- [17] D.M. Gordon, *Discrete logarithms in $gf(p)$ using the number field sieve*, *SIAM Journal on Discrete Mathematics* (1993), no. 6, 124–138.

- [18] G.H. Hardy and W.M. Wright, *An introduction to the theory of numbers*, fifth ed., Oxford Science Publications, New York, 1979.
- [19] Thomas W. Hungerford, *Algebra*, ch. III, pp. 131–133, in theorem 2.24 [20], 1974.
- [20] ———, *Algebra*, Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, U.S.A., 1974.
- [21] Anna Johnston, *A generalized q^{th} root algorithm*, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, January 1999, Baltimore, Maryland.
- [22] Anna Johnston and Peter Gemmell, *Authenticated key exchange provably secure against the man-in-the-middle attack*, Journal of Cryptology **15** (2002), no. 2, 139–148.
- [23] D.E. Knuth, *The art of computer programming: Seminumerical algorithms*, 2 ed., vol. 2, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.
- [24] Neal Koblitz, *A course in number theory and cryptography*, second ed., Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, U.S.A., 1994.
- [25] Serge Lang, *Algebra*, third ed., ch. [II,§2], pp. 63–64, in [26], 1971.
- [26] ———, *Algebra*, third ed., Addison-Wesley Publishing Co., Reading, Massachusetts, 1971.

- [27] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone, *An efficient protocol for authenticated key agreement*, Tech. Report CORR 98-05, Department of C&O, University of Waterloo, Canada, March 1998.
- [28] Franz Lemmermeyer, *Reciprocity laws: From euler to eisenstein*, Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, U.S.A., 2000.
- [29] Rudolf Lidl and Harald Niederreiter, *Finite fields*, second ed., Cambridge University Press, 1997.
- [30] F.J. MacWilliams and N.J.A. Sloane, *The theory of error-correcting codes*, seventh ed., Elsevier Science Publishers B.V., 1992.
- [31] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1996.
- [32] A. M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Advances in Cryptology: Proceedings of EuroCrypt '84 (Berlin) (Thomas Beth, Norbert Cot, , and Ingemar Ingemarsson, eds.), Springer-Verlag, 1984, Lecture Notes in Computer Science Volume 209, pp. 224–316.
- [33] S.C. Pohlig and M.E. Hellman, *An improved algorithm for computing logarithms over $gf(p)$ and its cryptographic significance*, Transactions on Information Theory, no. 24, IEEE, 1978, pp. 106–110.
- [34] J.M. Pollard, *Monte carlo methods for index computation (mod p)*, Mathematics of Computation, no. 32, 1978, pp. 918–924.
- [35] ———, *Kangaroos, monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447.

- [36] M.O. Rabin, *Digitalized signatures and public key functions as intractable as factorization*, Tech. Report MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.
- [37] R.L. Rivest, A. Shamir, and L.M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM (1978), no. 21, 120–126.
- [38] Kenneth H. Rosen (ed.), *Handbook of discrete and combinatorial mathematics*, CRC Press, 2000 Corporate Blvd., N.W; Boca Raton, Florida 33431, 1999.
- [39] C.P. Schnorr, *Efficient identification and signatures for smart cards*, Advances in Cryptology – CRYPTO, vol. 435, Lecture Notes in Computer Science, no. 435, 1989, pp. 239–252.
- [40] R. Schoof, *Elliptic curves over finite fields and the computation of square roots*, Mathematics of Computation (1985), 483–494.
- [41] Daniel Shanks, *Five number-theoretic algorithms*, Proceedings of the Second Manitoba Conference on Numerical Mathematics, no. VII, 1972, University of Manitoba, Winnipeg, Manitoba, pp. 51–70.
- [42] V. Shoup, *Fast construction of irreducible polynomials over finite fields*, Journal of Symbolic Computation **17** (1993), 371–391.
- [43] Douglas Stinson, *Cryptography: Theory and practice*, 1 ed., CRC Press, 2000 Corporate Blvd., N.W; Boca Raton, Florida 33431, 1995.
- [44] P. C. van Oorschot and M. J. Wiener, *Improving implementable meet-in-the-middle attacks by orders of magnitude*, Advances in Cryptology - Crypto

'96 (Berlin) (Neal Koblitz, ed.), Springer-Verlag, 1996, Lecture Notes in Computer Science Volume 1109, pp. 229–236.

- [45] P.C. van Oorschot and M.J. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28.