Root Finding over Finite Fields for Secure Multiparty Computation

van der Meer, Noah B.

Award date:
2021

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Root Finding over Finite Fields for Secure Multiparty Computation

Bachelor Thesis

Noah van der Meer

Supervisor:
Berry Schoenmakers

Eindhoven, August 2021

Abstract

In the past few decades, significant progress has been made in several areas of computational number theory and algebra. These developments have made it possible for computers to solve a variety of difficult problems, which was previously impractical. One problem which has received a lot of attention is the problem of factoring polynomials with coefficients in finite fields. This thesis focuses on root finding, a specific instance of this more general polynomial factorization problem. We discuss several algorithms for this task, propose implementations and optimizations and demonstrate their effectiveness in practice. We show an application in secure multiparty computation with the recently proposed Secure Shuffle protocol and demonstrate its use in practice using the MPyC Python library.

Contents

Notation

F_{p^k}	Finite field of cardinality p^k
$M(n)$	Field operations used to multiply two polynomials of degree at most n
$S(n)$	Largest prime factor of an integer n
$\omega(n)$	Number of distinct prime factors of an integer n
$\nu_r(n)$	Exponent of r in the prime factorization of n
$D(f)$	Formal derivative of a polynomial f
$\chi(a)$	Quadratic character in F_p

Chapter 1

Introduction

In this thesis we present a variety of methods for dealing with the computational problem of factoring polynomials over finite fields. Polynomial factorization has many use cases in mathematics and computer science, for instance in cryptography or in digital error correction techniques. This thesis will be concerned with the former category, focusing specifically on applications in secure multiparty computation. The polynomial factorization problem is related to various other computational problems, such as finding (non)residues or finding primitive roots in finite fields, both of which are also treated in this thesis.

1.1 Polynomial Factorization

In the late 1970s, the incredibly influential RSA cryptosystem was developed by Ron Rivest, Adi Shamir and Leonard Adleman. RSA provides methods for encrypting information and producing (and also verifying) digital signatures. In the present day, RSA is used in a variety of applications, such as securing connections on the world-wide-web, for digital signatures on documents and supporting secure email. The security of RSA is essentially based on the fact that in general, given an integer $n \in \mathbb{N}$, it is computationally very expensive to compute the complete prime factorization of n, i.e.

$$n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

where $p_1, ..., p_k$ are distinct primes. Currently, no efficient algorithms for solving this computational problem are known, which is why RSA is considered secure.

In computational algebra, many similarities exist between integers and polynomials, so much so that many influential works in literature treat both at the same time. However, one aspect in which the two can be seen to be different is the computational problem of factoring. While no efficient algorithms are known for computing the prime factorization of a number, numerous effective methods have been discovered and applied to the problem of polynomial factorization. Moreover, polynomial factorization has been used in the construction of various other algorithms, for instance in cryptography and digital communication. One application is secure multiparty computation, which is the application we will focus on.

In this thesis, we will be concerned with factoring polynomials over finite fields of the form:

$$f = (X - r_1)(X - r_2) \dots (X - r_l) \tag{1.1}$$

The focus on polynomial factorization techniques of this form is motivated by the secure multiparty computation (MPC) protocols recently presented in [Vre20], specifically the protocols for generating random permutations and for securely shuffling a list of secret values. In order to generate a random permutation of l elements, the protocol involves computing the polynomial factorization of a degree l polynomial as one of its core steps. This parameter l can be arbitrarily

large, further motivating the need for efficient factorization algorithms. These MPC protocols are discussed in Chapter 6.

The focus of the current work will be on finite fields F_p where $p > 2$ is prime. The motivating applications in secure multiparty computation also use such prime fields, and this helps in presenting the main algorithmic ideas that can used for polynomial factorization. With a little effort, many of these results can be generalized to deal with general fields F_{p^k}. Alternatively, [Ber70] presents a deterministic polynomial-time reduction from the problem of factoring general polynomials over finite fields F_{p^k}, to that of factoring polynomials of the form (1.1) over a prime field F_p. The methods presented in the current work could therefore potentially also be relevant for polynomial factorization in general.

Finally, observe that the case $p = 2$ is also excluded from consideration. This case often requires special attention, and the motivating applications in secure multiparty computation use significantly larger primes. Methods that deal exclusively with finite fields of the form F_{2^k} have been presented in the literature, see for instance [ZG02] for the specific case $k = 1$ and [KS97] for general k.

1.2 Secure Multiparty Computation

Secure multiparty computation is a branch of cryptography that provides techniques for multiple parties to evaluate a mathematical function over its secret inputs. Often, these inputs are distributed among participants through some form of secret sharing, in which each participant possesses a share of the input yet no individual participant is able to reconstruct the complete input without collaborating with the other parties. A scheme which is commonly used for this purpose is Shamir secret sharing [Sha79].

An illustrative example in secure multiparty computation is the so-called Yao's Millionaire's Problem, which will be briefly described here. Suppose that Alice and Bob are two millionaires that would like to determine which of them is richer, without disclosing their holdings to anyone (i.e. not to each other, and not to any external party). If A denotes the wealth of Alice and B denotes the wealth of Bob, they essentially wish to evaluate the logical statement

$$A \geq B$$

In his original paper [Yao82], Yao provides a solution to this problem in the form of a protocol that should be followed by Alice and Bob. While this example is perhaps mostly of academic interest, secure multiparty computation techniques can be used for real-world applications of high importance, such as electronic voting and digital auctioning [Bog+09]. In the current work, the focus is on the secure shuffle MPC protocols recently presented in [Vre20] which internally require polynomial factorization over finite fields. The secure shuffle protocol allows a group of participants to shuffle a list of secret-shared values, which could find real-world use in applications such as online card games.

In recent years, multiparty computation techniques have become more accessible for developers through the availability of dedicated libraries. An example of this is the MPyC library [Sch18], which will be used to implement the secure shuffle protocols from [Vre20].

1.3 Outline

Chapter 2 provides some of the preliminaries for finite fields that will be used throughout the report. A brief summary is provided for some of the computational primitives for polynomials over finite fields, such as polynomial multiplication, polynomial gcds, multi-point evaluation and interpolation and finally a *polynomial translation* operation. This chapter also includes some important results regarding residues and non-residues in finite fields, such as (a generalization of) Ankeny's Theorem [Ank52].

In Chapter 3, we focus on the problem of computing modular roots in F_p in general. This is a particular instance of the polynomial factorization problem, where the polynomial in question is of the form

$$X^r - c \tag{1.2}$$

where $r \mid p-1$ and where c is some rth power residue. The cases $r = 2$ and $r = 3$ are of particular interest since algorithms that address these cases can be used to directly factor polynomials of a low degree. For polynomials of degree 2, the familiar quadratic formula (also commonly known as the 'ABC formula' in Dutch) can be used, assuming that one is able to compute square roots. In a similar manner, the cubic and quartic formulas can be used to factor polynomials of degree 3 and 4 respectively. All of these formulas can be expressed in terms of square roots and cube roots. It is well-known that no formulas of this form exist for polynomials of higher degrees [Ros95].

Chapter 4 provides techniques for finding primitive roots in finite fields. For general prime fields F_p, finding such primitive roots can be rather difficult (the prime factorization of $p - 1$ is not necessarily known), but using the approach described in Section 4.2, we can construct in deterministic polynomial time a set of elements which is guaranteed to contain a primitive root. The size of this set is also bounded by a polynomial.

In Chapter 5, several algorithms are presented for factoring polynomials of the form (1.1) over finite fields. Most of these algorithms are based on existing methods from the literature, although one of these has only received limited attention in the past. One of these factorization algorithms is probabilistic in nature, in the sense that it requires random elements from F_p during the factorization process. The remaining algorithms are deterministic, which may be desirable in certain situations.

Chapter 6 is dedicated to the secure shuffle protocol and its use of polynomial factorization techniques. The current implementation of the protocol relies on the external NTL C++ library for a variety of operations. An implementation written entirely using Python and MPyC [Sch18] is provided, along with experimental results for running the protocol in practice.

Chapter 2

Finite Fields

The algorithms introduced in later sections rely on various computational primitives for finite fields, such as polynomial multiplication and techniques for finding (non-)residues. This chapter aims to provide a brief summary on known results and techniques on these topics. In general, proofs have been omitted although references to literature in which these results are discussed are provided.

First, Section 2.1 is concerned with various aspects and techniques for computing with polynomials in $F_p[X]$, such as for polynomial multiplication, division, gcds and finally also a type of translation that can be performed on polynomials. Section 2.2 presents results on and techniques for finding (non-)residues in finite fields. Some of the results presented in this chapter rely on unproven but well-known conjectures such as the extended riemann hypothesis. These results are marked as such throughout the thesis.

2.1 Computational aspects

Throughout this thesis, algorithms will be analyzed according to the computational model and the notation used in [Sho09]. The asymptotic behaviour of algorithms is expressed in terms of field operations over F_p (i.e. additions, multiplications and divisions) using standard "Big-Oh" notation. This is consistent with [Sho09] (see Chapter 3) and [VG13] (see Chapter 25), two comprehensive books on the topic of computational algebra. In literature, one often also encounters the "Soft-Oh" notation $\tilde{O}$ in which logarithmic factors are swallowed as well. However, this would not allow us to distinguish the asymptotic behaviour of some algorithms later on and is therefore not used.

An algorithm is said to run in *polynomial-time* if the asymptotic running time is bounded by some polynomial in the input size. For instance, throughout this report, polynomials are represented by coefficient lists $[a_0, ..., a_l]$ and thus such an algorithm is polynomial-time if it is polynomial in l and $\log(p)$. In the literature this sometimes also includes $S(p-1)$, the largest prime factor of $p-1$, of which an example is [Gat87].

2.1.1 Polynomial operations

Throughout this report, let $M(l)$ denote the number of operations necessary to compute the product (that is, the coefficient representation) of two polynomials $f, g \in F_p[X]$ with $\deg(f), \deg(g) \leq l$. Using a fairly basic approach, this can be achieved using $O(l^2)$ operations. In [CK87] and shortly after in [CK91], general algorithms for multiplying such polynomials using $O(l \log(l) \log\log(l))$ operations are presented. These advanced algorithms make use of fast fourier transforms in various ways, although discussing this further is beyond the scope of the current work. We will make the common assumption that $M(a)/a \leq M(b)/b$ for $0 < a \leq b$, which will allow us to express the asymptotic behaviour of algorithms in terms of general $M(l)$ later on.

Similar to the problem of multiplication, division with remainder of a polynomial $f \in F_p[X]$ by another polynomial $g \in F_p[X]$ can be performed in $O(\deg(g)\deg(q))$ operations using the standard approach, where $f = qg + r$ with $\deg(r) < \deg(g)$ [Sho09] (see Chapter 17). Alternatively, using the methods presented in [AH74] (see Chapter 8), polynomial division with two polynomials with $\deg(f), \deg(g) \leq l$ can be performed in $O(M(l))$ operations. The polynomial gcd of two such polynomials can be computed using $O(l^2)$ operations using the euclidean algorithm, or $O(M(l)\log(l))$ operations using a recursive *Half-GCD* approach as presented in [AH74] (see Chapter 8).

2.1.2 Polynomial evaluation and interpolation

Let $f \in F_p[X]$ be an arbitrary polynomial of degree $l > 0$. Using the well-known Horner's Method, f can be evaluated at a point $u \in F_p[X]$ using $O(l)$ operations, and in a straightforward manner, f can thus also be evaluated at the points $u_1, u_2, ..., u_k \in F_p$ using $O(kl)$ operations. If $k \leq l$, using fast multi-point evaluation this can be further improved to $O(M(l)\log(l))$ operations [VG13] (see Chapter 10). Similarly the reverse operation of polynomial interpolation using l distinct points can be performed using $O(M(l)\log(l))$ operations as well [VG13] (see Chapter 10). The following result is a simple consequence:

Proposition 2.1.1. *Consider the finite field F_p, let $f \in F_p[X]$ be a polynomial of degree $l > 0$ with $p \geq l$ and let $\alpha \in F_p$ be arbitrary. The polynomial translation $\hat{f} = f(X + \alpha)$ can be computed using $O(M(l)\log(l))$ field operations in F_p.*

Proof. Let $u_1, u_2, ..., u_l \in F_p$ be arbitrary. Using fast multipoint evaluation, f can be evaluated in $u_1 + \alpha, u_2 + \alpha, ..., u_l + \alpha$ using $O(M(l)\log(l))$ field operations. Thus $\hat{f}(u_1), \hat{f}(u_2), ..., \hat{f}(u_l)$ are known, and using fast polynomial interpolation, the coefficients of $\hat{f}$ can be computed using $O(M(l)\log(l))$ field operations. ■

This result will be useful in one of the factorization algorithm later on, although the construction used in the proof of Proposition 2.1.1 is mainly of theoretical significance. More efficient algorithms for this task could be constructed, although this is beyond the scope of the current work. In the literature, the polynomial $f(X+\alpha)$ has occasionally also been referred to as a *Taylor shift*.

2.1.3 Polynomial rings

Finally, this section contains some general results regarding $F_p[X]$, which will be used later on in the factorization algorithms. More details can be found in [Ber70].

Lemma 2.1.1. *Consider the polynomial ring $F_p[X]$. It is the case that*

$$X^p - X = \prod_{r \in F_p} (X - r)$$

Lemma 2.1.2. *Let $f \in F_p[X]$ be a monic polynomial, and let (g_i) be a family of pairwise relatively prime polynomials in $F_p[X]$. If $f \mid \prod_i g_i$, then*

$$f = \prod_i \gcd(f, g_i)$$

2.2 Residues

Consider once more the finite field F_p, and let $r \mid p-1$ be some integer. An element $a \in F_p$ is called an *rth power residue* if there exists a $b \in F_p$ such that $b^r \equiv a$. If an element is not an rth power residue, it is called an rth power non-residue. The following relatively well-known proposition provides a simple yet effective method of checking the residuosity of an element [Leh59].

Proposition 2.2.1 (Euler's criterion)**.** *Consider the finite field F_p, and let $r \mid p-1$. An element $a \in F_p^*$ is an rth power residue if and only if $a^{\frac{p-1}{r}} \equiv 1$.*

This result allows one to check the residuosity of an element using one exponentiation, which can be done in $O(\log(p))$ multiplications in F_p by using fast modular exponentiation techniques. Algorithms that perform faster in practice also exist, such as those based on the quadratic reciprocity law for the case $r = 2$, but for the current work Euler's criterion will suffice.

2.2.1 Finding non-residues

Many algorithms in algebra and number theory rely on the availability of non-residues. Significant examples include computing modular roots and finding primitive roots. In Chapters 3 and 4, rth power non-residues with $r \mid p-1$ prime will be necessary for a variety of algorithms, and thus this section is dedicated to finding such non-residues.

If r is large enough, then one expects non-residues to be abundant within F_p^*. Indeed, one can show that $\frac{p-1}{r}$ elements are rth power residues, while the remainder are non-residues. The probability that a uniformly distributed $a \in F_p^*$ is an rth power non-residue is approximately $\frac{1}{r}$, and since residuosity can be tested rather quickly using $O(\log(p))$ operations, this provides an effective probabilistic method of finding non-residues. Trying random elements $a \in F_p^*$ is expected to succeed within $O(1)$ tries, leading to an algorithm that uses an expected $O(\log(p))$ operations.

While finding non-residues in a probabilistic manner is fairly easy, at this time no unconditional deterministic polynomial-time methods for finding non-residues in F_p^* are known. However, assuming the extended riemann hypothesis, the following results can be used to construct a conditional algorithm [Bac90b]:

Theorem 2.2.1 (Assuming ERH)**.** *Consider the finite field F_p and let G be a proper multiplicative subgroup of F_p^* such that $a \in G$ for all $a < b$. Then $b < 2(\log(p))^2$*

Corollary 2.2.1 (Assuming ERH)**.** *Consider the finite field F_p, and let $r \mid p-1$ be some divisor. The least rth power non-residue b satisfies $b \leq 2(\log p)^2$*

This can be seen as a generalization of Ankeny's Theorem, which applies to quadratic residues [Ank52]. Using these results, one can consider the elements $1, 2, 3, ..., 2(\log p)^2$ and check for residuosity using Proposition 2.2.1. This leads to a deterministic algorithm that uses $O((\log p)^3)$ field operations at most.

Chapter 3

Modular roots

3.1 Overview

This chapter is concerned with determining solutions to equations of the form

$$X^r \equiv c \tag{3.1}$$

where $r \geq 2$ is some prime and $c \in F_p$. We can make a distinction between the following cases:

1. $\gcd(r, p-1) = 1$
2. $r \mid p-1$

In case (1), there exists a $d \in \mathbb{N}$ such that $rd \equiv 1 \pmod{p-1}$ and c^d is the unique rth root of c as a result of Euler's totient theorem. In case (2), solutions to equation (3.1) only exist if c is an rth power residue. This case is significantly more difficult to solve in general, and the only algorithms that are known to solve this efficiently without relying on unproven conjectures are probabilistic in nature. If certain elements (e.g. rth non-residues) are given beforehand, the problem can be solved efficiently in a deterministic manner, but presently, it is unclear whether the problem can be solved efficiently through an unconditional deterministic algorithm that works in general. For the rest of this section, we will assume that $p - 1 = r^s t$, where $s \geq 1$ and $\gcd(r, t) = 1 = \alpha r + \beta t$ are all known.

The problem of solving equation (3.1) is a very specific instance of the broader polynomial factorization problem, which might occur in certain applications. In addition, the quadratic, cubic and quartic formulas for computing the roots of polynomials of degree $2, 3$ and 4 can be expressed in terms of square roots and cube roots, which will become relevant in Chapter 5.

The remainder of this chapter is divided into 4 sections. Section 3.2 is dedicated to the Adleman-Manders-Miller algorithm, along with a full derivation and complexity analysis. Section 3.3 is concerned with problem of computing square roots specifically. First, several special cases are handled (e.g. $p \equiv 5 \pmod 8$) and later in Section 3.3.2 an algorithm that works for all $p \equiv 1 \pmod 4$ is presented. This algorithm was proposed in [Per86] and was briefly analyzed in [Bac90a] although appears to have received only limited attention in the literature. Still, the experimental results in Section 3.4 demonstrate that the algorithm is competitive with more common algorithms such as Cipolla.

3.2 Adleman-Manders-Miller

A commonly used algorithm for computing modular square roots is the Tonelli-Shanks algorithm, originally discovered by A. Tonelli in 1891 and later rediscovered by D.Shanks in 1973. This algorithm is once more discussed in [AMM77], in which authors also provide notes on how this method could potentially be generalized to compute rth roots in general. In this section, the

resulting algorithm is presented, along with a full derivation and complexity analysis. Although the presentation is slightly different, the underlying ideas are the same as those used in [AMM77]. The algorithm is essentially based on the following two propositions:

Proposition 3.2.1. *Let $a \in F_p$ be such that $r \nmid \mathrm{ord}_p(a)$. Then a^α is an rth root of a in F_p.*

Proof. Since $r \nmid \mathrm{ord}_p(a)$, it must be the case that $\mathrm{ord}_p(a) \mid t$. Therefore

$$(a^\alpha)^r \equiv a \cdot (a^{\alpha r - 1}) \equiv a \cdot a^{-\beta t} \equiv a$$

■

Proposition 3.2.2. *Let $g \in F_p$ be an rth power non-residue and let $a \in F_p$ be arbitrary. If $0 < j < s$ such that $a^{r^j t} \equiv 1$ and $a^{r^{j-1} t} \neq 1$, then there exists a $\lambda \in \{1, ..., r-1\}$ such that*

$$(a g^{\lambda r^{s-j}})^{r^{j-1} t} \equiv 1$$

Proof. Consider the equation

$$X^r \equiv 1 \tag{3.2}$$

For convenience, define $u = g^{r^{s-1} t}$. It is clear that u is a solution to equation (3.2), and by Proposition 2.2.1:

$$u = g^{r^{s-1} t} = g^{(p-1)/r} \neq 1$$

Therefore $\mathrm{ord}_p(u) = r$ and u is in fact a primitive rth root of unity. The multiplicative subgroup $S = \langle u \rangle = \{u^i \mid i = 0, ..., r-1\}$ generated by u contains r distinct elements, which are also solutions to equation (3.2). By the fundamental theorem of algebra, these are all solutions of the equation. Finally, observe that since it is given that $a^{r^j t} \equiv 1$, the element $a^{r^{j-1} t}$ is also a solution to (3.2), which implies that there exists a $\xi \in \{0, ..., r-1\}$ such that

$$a^{r^{j-1} t} = u^\xi \tag{3.3}$$

Note that $\xi \neq 0$ since $a^{r^{j-1} t} \neq 1$. Then $\lambda = r - \xi$ is as required ■

As a result of this proposition, as long as $\mathrm{ord}_p(a)$ contains a factor r, there exist λ and j such that $\mathrm{ord}_p(a g^{\lambda r^{s-j}})$ has less factors r in its prime factorization. This is the essential idea behind the modular root algorithm.

Let g be an rth power non-residue and consider once more the rth power residue c from equation (3.1). If $r \nmid \mathrm{ord}_p(c)$, we immediately find an r root of c through Proposition 3.2.1. Therefore, assume that $r \mid \mathrm{ord}_p(c)$ and let $j = \nu_r(\mathrm{ord}_p(c)) > 0$ such that $c^{r^j t} \equiv 1$ and $c^{r^{j-1} t} \neq 1$. As a consequence of Proposition 3.2.2, there exists a λ_1 such that

$$1 \equiv (c g^{\lambda_1 r^{s-j}})^{r^{j-1} t} \equiv c^{r^{j-1} t} g^{\lambda_1 r^{s-1}}$$

If $j > 1$, we may continue the process with $c_2 = c g^{\lambda_1 r^{s-j}}$. For the sake of describing the main algorithmic ideas, assume that $\nu_r(\mathrm{ord}_p(c_2)) = j - 1$, i.e. the order decreases by one in each application of Proposition 3.2.2. The Algorithm given in Section 3.2.1 will make use of larger jumps whenever possible to improve efficiency in practice. Apply Proposition 3.2.2 on c_2, to obtain a λ_2 such that

$$1 \equiv (c g^{\lambda_1 r^{s-j}} g^{\lambda_2 r^{s-(j-1)}})^{r^{j-2} t} \equiv c^{r^{j-2} t} g^{\lambda_1 r^{s-2} t} g^{\lambda_2 r^{s-1} t}$$

By repeatedly applying Proposition 3.2.2 in this manner, we obtain a sequence $\lambda_1, ..., \lambda_k$ such that

$$1 \equiv c^t g^{t(\lambda_1 r^{s-j} + \lambda_2 r^{s-j+1} + ... + \lambda_j r^{s-1})}$$

It follows that

$$\begin{aligned}(c^\alpha g^{-\beta t(\lambda_1 r^{s-j-1} + \lambda_2 r^{s-j} + ... + \lambda_j r^{s-2})})^r &\equiv c \cdot c^{\alpha r - 1} g^{-\beta t(\lambda_1 r^{s-j} + \lambda_2 r^{s-j+1} + ... + \lambda_j r^{s-1})} \\ &\equiv c \cdot (c^t g^{t(\lambda_1 r^{s-j} + \lambda_2 r^{s-j+1} + ... + \lambda_j r^{s-1})})^{-\beta} \\ &\equiv c\end{aligned}$$

and thus $c^\alpha g^{-\beta t \sum_{i=1}^{j} \lambda_i r^{s-j-2+i}}$ is an rth root of c as desired.

3.2.1 Algorithm

Proposition 3.2.2 itself does not necessarily provide any means of determining the λ that are necessary in the algorithm. However, the corresponding proof does present some insight. In particular, if we define $u = g^{r^{s-1}t}$ as in the proof, we are essentially looking for a $\xi \in \{1, ..., r-1\}$ such that

$$a^{r^{j-1}t} = u^\xi \tag{3.4}$$

Equivalently, $\xi = \log_u(a^{r^{j-1}t})$, and we are computing a discrete logarithm in the subgroup $\langle u \rangle = \{1, u, u^2, ..., u^{r-1}\}$. Computing discrete logarithms in F_p is typically seen as a difficult problem and currently no polynomial-time algorithms are known to exist that work in general on classical computers (although using a quantum computer it *is* possible, see [Sho99]). However, in this case we only need to compute a discrete logarithm in the smaller subgroup $\langle u \rangle$. In applications where $r << p$, a brute-force search might become feasible, which is the solution suggested in [AMM77].

Alternatively, one could pre-compute the powers $u^2, u^3, ..., u^{r-1}$ and store these in a table sorted based on the canonical representatives in $\{0, ..., p-1\}$. In this way, ξ can be found rapidly through a binary search algorithm using $O(\log(r))$ comparisons. This approach can be seen in [Van05]. Similarly, the powers $g^t, g^{rt}, g^{r^2 t}, ..., g^{r^{s-1}t}$ can be computed and stored in a table at the beginning of the algorithm, and this idea of using tables to improve the algorithm can already be seen in [Ber01]. This avoids performing unnecessary operations, effectively trading storage for efficiency. The complete algorithm is now as follows:

Algorithm 1: Adleman-Manders-Miller

Input : prime field F_p, prime r with $r \mid p-1$, rth power residue c, rth power non-residue b

Output: element $x \in F_p$ such that $x^r = c$

1 Let s, t be such that $p - 1 = r^s t$, with $\gcd(r, t) = 1$. Also, let $\alpha, \beta \in \mathbb{N}$ such that $\alpha r + \beta t = 1$.
2 Set $g \leftarrow b^t$, compute the powers $g^r, g^{r^2}, g^{r^3}, ..., g^{r^{s-1}}$ and store these in table G.
3 Set $u \leftarrow g^{r^{s-1}}$, compute the powers $u^2, u^3, ..., u^{r-1}$ and store these in a sorted table U.
4 Set $L \leftarrow 1$ and $C \leftarrow c^t$
5 **while** $j > 0$ **do**
6 Find $j \in \{1, ..., s-1\}$ such that $C^{r^j} \equiv 1$.
7 Find $\xi \in \{1, ..., r-1\}$ such that $C^{r^{j-1}} \equiv u^\xi$ using table U. Set $\lambda = r - \xi$
8 Set $C \leftarrow C g^{\lambda r^{s-j}}$ using table G
9 Set $L \leftarrow g^{\lambda r^{s-j-1}}$ using table G
10 **end**
11 Compute and return $c^\alpha L^{-\beta}$

Proposition 3.2.3. *Algorithm 1 runs using at most $O(r + \log p + s^2 \log r)$ operations in F_p and storage for $O(r + s)$ elements of F_p.*

Proof. It is assumed that s, t and α, β are provided beforehand. Next, $g \cdot b^t$ can be computed using $O(\log t)$ operations through the usual square-and-multiply exponentiation techniques. The powers $g^r, g^{r^2}, ..., g^{r^{s-1}}$ can be computed in $O(s \log r) \subseteq O(\log p)$ operations through repeated exponentiation by r. The powers $u^2, u^3, ..., u^{r-1}$ are computed using $O(r)$ multiplications, and finally C can be computed using $O(\log t)$ operations.

In each iteration of the for-loop, j can be found using a brute-force approach, computing $C^r, C^{r^2}, ..., C^{r^{j-1}}$ until eventually $C^{r^j} = 1$. This can be done using $O(s \log r)$ operations through repeated exponentiation by r. Next, ξ can be found using binary search in $O(\log r)$ comparisons. Using table G, the elements $Cg^{\lambda r^{s-j}}$ and $g^{\lambda r^{s-j-1}}$ can be computed using $O(\log r)$ operations (observe that $\lambda \leq r$). Thus overall, one iteration of the for-loop uses $O(s \log r)$ operations, for a total cost of $O(s^2 \log r)$ operations. Finally, c^α and L^β can be computed using $O(\log(p))$ operations (observe that we may take $\alpha, \beta \pmod{p-1}$). ■

This result shows that if one has access to an rth power non-residue, modular roots can be computed efficiently in deterministic polynomial-time. However, as can be seen in Proposition 3.2.3, the complexity of Algorithm 1 is dependent on the size of s. If s is small, this provides a highly effective method of computing modular roots. However, if s is large, more efficient algorithms exist, especially for specific r. This is the subject of the next few sections.

Several parts of Algorithm 1 could potentially be pre-computed when using the algorithm in practice. In particular, g, u and the tables G and U are data-independent and only depend on the rth power non-residue b. At the cost of storing $O(r + s)$ elements, the resulting algorithm would become more efficient in practice, although asymptotically the result from Proposition 3.2.3 remains the same.

3.3 Square roots

The algorithm presented in the previous section is applicable to general r and can be effective in practice. However as indicated before, several algorithms exist that perform better when r is known beforehand. In this section, we focus on the case $r = 2$, computing square roots.

3.3.1 Special cases

Let $c \in F_p$ be a quadratic residue. If $p \equiv 3 \pmod 4$, it is relatively well-known that $c^{(p+1)/4}$ is a square root of c, which essentially follows from Proposition 2.2.1. On the other hand, if $p \equiv 1 \pmod 4$, the situation becomes slightly more involved. In the case that $p \equiv 5 \pmod 8$, the method presented in [Atk92] can be used to obtain square roots with little effort. In [Mül04], this procedure is further extended to the case $p \equiv 9 \pmod{16}$. The method for $p \equiv 5 \pmod 8$ is briefly described below.

If $p \equiv 5 \pmod 8$, the $2 \in F_p$ is a quadratic non-residue in F_p and thus also $2c$ is a quadratic non-residue. If we define $\xi = (2c)^{(p-5)/8}$ and $\zeta = 2c\xi^2$, it follows that

$$\zeta^2 \equiv (2c)^{(p-5)/4} \cdot 2c \equiv (2c)^{(p-1)/4} \equiv -1$$

It follows directly that $(\zeta - 1)^2 \equiv -2\zeta \equiv -4c \cdot \xi^2$, which implies

$$\begin{aligned}(c(\zeta - 1)\xi)^2 &\equiv c^2(\zeta - 1)^2\xi^2 \\ &\equiv -4c^3 \cdot (2c)^{(p-5)/2} \\ &\equiv -c \cdot (2c)^{(p-1)/2} \\ &\equiv c\end{aligned}$$

Note that ξ can be computed using $O(\log(p))$ operations through the usual square-and-multiply exponentiation techniques. Thus if $p \equiv 5 \pmod 8$, we can compute square roots using a $O(\log(p))$ algorithm and with modest constants within the Big-Oh notation. This approach is unconditionally deterministic, as opposed to Algorithm 1 which relies on the availability of a quadratic non-residue in this case.

3.3.2 Peralta

In [Per86], two probabilistic algorithms are presented for computing square roots. Similar to the commonly used Cipolla algorithm, these algorithms are based on extension field arithmetic. Although the presentation is quite different, the first algorithm turns out to be equivalent to the Berlekamp root finding method described in Section 5.2 as noted in [Bac90a]. The second algorithm provides a new method for computing square roots, although once more it turns out that this method had been discovered earlier in 1917 by Pocklington. This algorithm has received only limited attention in the literature, but in Section 3.4 we demonstrate that this algorithm is in fact competitive with the more common Cipolla-Lehmer and Adleman-Manders-Miller algorithms.

If $p = 3 \pmod 4$, square roots can readily be computed as noted in Section 3.3.1 and thus assume that $p = 1 \pmod 4$. The algorithm is essentially based on the following proposition:

Proposition 3.3.1. *Consider the finite field F_p and let $b \in F_p$ and $(Y + b)^t = \gamma Y + \rho$ with $\gamma, \rho \neq 0$. There exists an $0 < i < s$ such that*

$$(Y + b)^{t2^i} = \alpha Y \pmod{Y^2 + c}$$

for some $\alpha \in F_p$.

Proof. Let l be the smallest value such that

$$(Y + b)^{t2^l} \equiv 0Y + a$$

for some for some $a \in F_p$. Since $(Y + b)^{t2^s} = (Y + b)^{p-1} \equiv 1$, it is the case that $l \leq s$. Also, $l > 0$ since $\gamma \neq 0$. Suppose $\alpha, \beta \in F_p$ are such that

$$(Y + b)^{t2^{l-1}} \equiv \alpha Y + \beta \pmod{Y^2 + c}$$

Then $0Y + a \equiv (\alpha Y + \beta)^2 \equiv 2\alpha\beta Y + \beta^2 - \alpha^2 c$, and thus $\alpha\beta = 0$. This implies that $\beta = 0$, since $\alpha = 0$ would contradict minimality of l. Also, $l - 1 > 0$ since it was assumed that $\rho \neq 0$.■

Now suppose that we have found $0 < i < s$ such that $(Y + b)^{t2^i} = wY \pmod{Y^2 + c}$ for some $w \in F_p$. If we let $u, v \in F_p$ such that $(Y + b)^{t2^{i-1}} = uY + v \pmod{Y^2 + c}$, then

$$v^2 - u^2 c + 2uvY \equiv v^2 + u^2 Y^2 + 2uvY \equiv (uY + v)^2 \equiv wY$$

which implies $v^2 - u^2 c \equiv 0 \pmod p$. Hence vu^{-1} is a square root of c, as desired. This method requires an element $b \in F_p$ such that $v + uY = (Y + b)^t$ with $u, v \neq 0$. The following result, which is Theorem 4 from [Per86], shows that such elements are in fact abundant:

Proposition 3.3.2. *The probability that a random $b \in F_p^*$ yields $(Y + b)^t = v + uY$ with both u, v distinct from 0 is $1 - \frac{1}{2^{s-1}}$*

This suggests that trying random elements $b \in F_p^*$ is a feasible strategy, with the probability of success increasing as s increases. This is also demonstrated by experimental results in Section

3.4. The algorithm is now given as following.

Algorithm 2: Peralta

Input : prime field F_p with $p = 1 \pmod 4$, quadratic residue c
Output: A square root of c

Set $A = F_p[Y]/(Y^2 + c)$;
Try random $b \in F_p^*$ until $(Y + b)^t = u_0 Y + v_0$ in A, with $u_0, v_0 \neq 0$;
$i \leftarrow 0$;
while $v_i \neq 0$ **do**
    Compute u_{i+1}, v_{i+1} such that $u_{i+1}Y + v_{i+1} = (u_i X + v_i)^2$ in A;
    $i \leftarrow i + 1$;
end
return $v_i \cdot u_i^{-1}$;

Proposition 3.3.3 (Expected complexity)**.** *Algorithm 2 runs in* $O(\log p)$ *expected operations in* F_p

Proof. As a consequence of Proposition 3.3.2, we expect the initial search for b to succeed within $O(1)$ attempts. Each attempt uses $O(\log(t))$ field operations. Each iteration of the main while-loop uses $O(1)$ field operations, and by Proposition 3.3.1 the number of iterations is bounded from above by s iterations. ■

The asymptotic complexity of Algorithm 2 can be seen to match that of the more commonly used Cipolla algorithm. Notably, s does not influence the efficiency of Algorithm 2 asymptotically, although in practice the algorithm performs better when s is large as a consequence of Proposition 3.3.2. This is the opposite of the Adleman-Manders-Miller algorithm, which performs significantly worse as s increases. As a final note, observe that the element b required by the algorithm is dependent on the input c, as opposed to the arbitrary non-residue used in the Adleman-Manders-Miller algorithm.

3.4 Experimental Results

In this section, experimental results are presented for some of the algorithms discussed in this chapter. These have been implemented in Python using the MPyC library for secure multiparty computation [Sch18]. In order to measure the computational speed of these algorithms, a fixed number of random residues is first sampled on which the algorithm in question will be applied. The average time necessary for the algorithm to complete is then reported as the average measurement over these random residues.

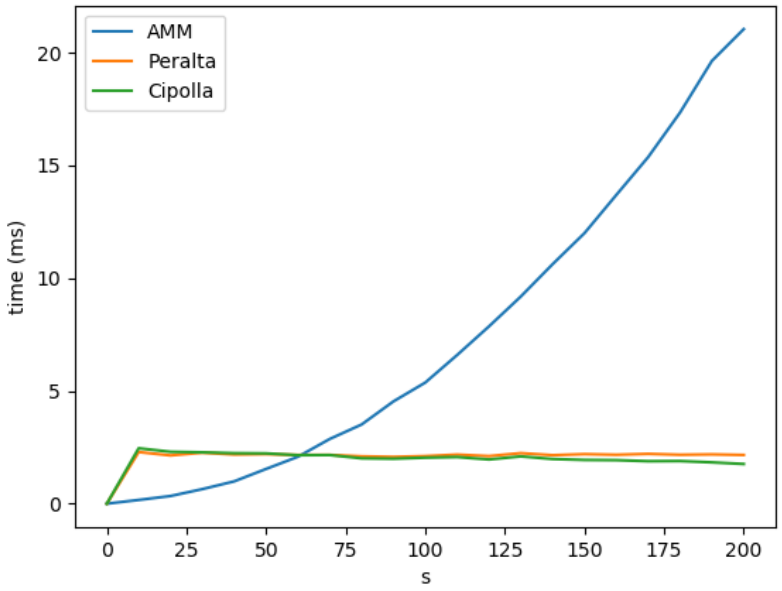

Figure 3.1: Comparison between Adleman-Manders-Miller (with pre-computations), Peralta and Cipolla algorithms for 256-bit primes with varying s

$s \rightarrow$	4	20	40	60	80	100	120	140	160	180	200
AMM	0.11	0.34	0.99	2.08	3.51	5.37	7.86	10.62	13.69	17.34	21.04
Peralta	2.55	2.14	2.18	2.16	2.10	2.11	2.11	2.15	2.17	2.17	2.16
Cipolla	2.38	2.30	2.24	2.15	2.01	2.04	1.96	1.98	1.93	1.89	1.76

Table 3.1: Computing square roots for 256-bit primes with varying s. Entries indicate time in milliseconds needed to compute one square root, averaged over $d = 2500$ random quadratic residues

In Figure 3.1 and Table 3.1, the computation time used by the Python implementations of the Adleman-Manders-Miller, Peralta and Cipolla algorithms is shown for varying s. The implementation of the Cipolla algorithm is based on [BS96] (Chapter 7). All results were obtained with a desktop system equipped with an Intel i5-4670K processor and 32GB of memory. No optimizations were made to the algorithms beyond the ones mentioned in this chapter.

Figure 3.1 demonstrates that for small s, the Adleman-Manders-Miller algorithm is highly effective and outperforms the competing algorithms. On the other hand, as s increases, the performance of the Adleman-Manders-Miller algorithm rapidly drops and instead the Peralta and Cipolla algorithms perform significantly better. This matches our expectation from the earlier complexity analysis of the Adleman-Manders-Miller algorithm in Proposition 3.2.3. Figure 3.1 suggests that a scheme using a combination of Adleman-Manders-Miller and one of Peralta, Cipolla, could be very effective for computing square roots. In such a scheme, based on the given finite field F_p the optimal *regime* could be determined (i.e. the underlying square root algorithm). Note however that this breaking point at which the regime should be changed depends on the particular implementation. Figure 3.1 also shows that the Peralta algorithm can be effective in practice as an algorithm for computing square roots. The implementation of the Cipolla algorithm is slightly more efficient, although this could potentially change if further implementation-level optimizations are made to the Peralta algorithm.

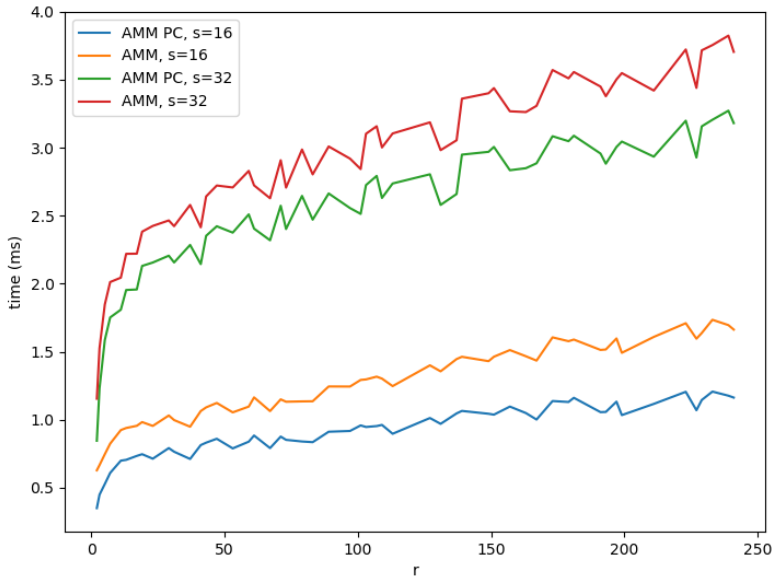

Figure 3.2: Adleman-Manders-Miller (with and without pre-computations) for 386-bit primes p, with $s = 16$ and $s = 32$ and with the horizontal axis spanning the prime numbers between 2 and 250

Next, we turn our attention to the Adleman-Manders-Miller algorithm specifically, which works for general r. Figure 3.2 shows the time needed by the Adleman-Manders-Miller algorithm for increasing r and for $s \in \{16, 32\}$. The pre-computations suggested in Section 3.2 can be seen to be effective in reducing the practical computation times used by the algorithm. Moreover, the results match the complexity analysis in Proposition 3.2.3 in that the computation times of Adleman-Manders-Miller increase roughly linearly as r increases.

Chapter 4

Primitive Roots

For a finite field F_p, a primitive root is an element that generates the entire multiplicative group F_p^*. Some the polynomial factorization algorithms discussed in the next chapter rely on the availability of such a primitive root, but finding primitive roots is often rather non-trivial. In fact, testing whether a given element $g \in F_p^*$ is a primitive root is in many cases already very difficult.

If the complete prime factorization of $p-1 = q_1^{e_1} \dots q_k^{e_k}$ is known, one can easily check whether an element g is a primitive root. The generate-and-test approach for finding a primitive root would therefore become feasible. Thus, if the prime factorization of $p-1$ is known, primitive roots are readily available. However, this is not necessarily the case in all applications, and the problem of computing a prime factorization is considered to be very hard. Still, using the methods described in the next few sections, one can often still use primitive roots in practice.

4.1 Least primitive root

While the problem of pinpointing a primitive root remains difficult, there are methods of constructing small collections which are guaranteed to contain a primitive root. In [Sho90b], it is shown that assuming the extended riemann hypothesis, the least primitive root in F_p is bounded as following:

Theorem 4.1.1 (Assuming ERH)**.** *Consider the prime field F_p. The least primitive root in F_p is in $O((r\log(r))^4(\log p)^2)$, where $r = \omega(p-1)$.*

This can be combined with the following result from [HW79] (see Chapter 22):

Theorem 4.1.2. *Let $\omega(n)$ denote the number of distinct prime factors of an integer n. Then $\omega(n) \in O(\frac{\log n}{\log\log n})$*

Combining these two results, it follows that the smallest primitive root is in $O((\log p)^6)$ and one may thus consider the elements $1, 2, ...,$ up to some polynomial. Since testing whether an element is a primitive root is hard, it is in general not possible to pinpoint which of these elements is a primitive root efficiently. Nevertheless, even if it turns out that a candidate primitive root g is not actually a primitive root, the various factorization algorithms will still terminate. Thus, one can run the algorithm in question using increasing candidate primitive roots, which is guaranteed to succeed at some point as a consequence of Theorem 4.1.1.

4.2 Bounded collections

While the previous section provides a feasible deterministic method for using primitive roots, the idea can be significantly improved. In [Bac97], a method is described that allows one to construct a set of $O(\log(p)^4(\log\log(p))^{-3})$ elements, which under the ERH is guaranteed to contain a primitive

root. Moreover, it can be shown that this set can be constructed in a number of operations polynomial in $\log(p)$. The method is essentially based on the following result:

Proposition 4.2.1 (Assuming ERH)**.** *Consider the prime field F_p, and let $B \geq 1$ such that $B\log(B) = 30\log(p)$. Let T denote the set of prime divisors of $p-1$ exceeding B. If T is non-empty, there exists a prime b with*

$$b \leq 5\frac{(\log(p))^4}{(\log\log(p))^2}$$

such that for all $q \in T$, b is a qth power non-residue in F_p.

Proof. This is a special case of Lemma 2.4 in [Bac97]. ■

Now let $B \in \mathbb{R}$ as in proposition, and suppose that the all prime factors of $p-1$ less than B are known, i.e.

$$p - 1 = q_1^{e_1} q_2^{e_2} \dots q_r^{e_r} \cdot t$$

where $q_1, q_2, .., q_r < B$ are distinct primes and t does not have any prime factors smaller than B. Additionally, suppose that for each q_i, we have access to a q_ith power non-residue b_i. If we define a as

$$a = \prod_{i=1}^{r} b_i^{(p-1)/q_i^{e_i}}$$

then a has order $q_1^{e_1} q_2^{e_2} \dots q_r^{e_r} = (p-1)/t$. If $t = 1$, we have found a primitive root and we are done. If $t > 1$, then by Proposition 4.2.1 there exists a prime b with

$$b \leq 5\frac{(\log(p))^4}{(\log\log(p))^2}$$

which is a qth non-residue for all prime divisors $q \mid t$. This impplies that that $b^{(p-1)/t}$ has order t, and thus $ab^{(p-1)/t}$ has order $p-1$ as desired.

In this construction, a trade-off is made in picking the parameter B. If B satisfies $B\log(B) = 30\log(p)$, then any $\hat{B} > B$ will in fact also work for Proposition 4.2.1. However, this comes at the cost of having to compute a larger portion of the prime factorization of $p-1$. In general, an appropriate B can be found by applying a numerical root finding algorithm (e.g. Brent's Algorithm [Bre13]) to the function $x \mapsto (x\log(x) - 30\log(p))$. In algorithmic form, the procedure is as following:

Algorithm 3: Generate-Primitive-Root-Set

Input : prime field F_p, parameter $B \in \mathbb{R}$ with $B\log(B) = 30\log(p)$
Output: Set S containing a primitive root

Compute the partial prime factorization $p - 1 = q_1^{e_1} q_2^{e_2} \dots q_r^{e_r} \cdot t$ of primes $q_i < B$.;
for $i \leftarrow 1$ *to* r **do**
  Try $b_i = 1, 2, ...$ until a q_ith power non-residue is found;
end
$a \leftarrow \prod_{i=1}^{r} b_i^{(p-1)/q_i^{e_i}}$;
$S \leftarrow \{ab^{(p-1)/t} \mid b \wedge b \leq 5\frac{(\log p)^4}{(\log\log p)^2}\}$;
return S;

The correctness of the algorithm follows from the preceding discussion. Since this method relies on Proposition 4.2.1, correctness is actually conditional on the ERH. Note that in the construction

of S, also non-prime b are used, which is essentially for computational purposes. If we were to restrict S to use only prime numbers b, the prime number theorem would imply that

$$|S| \leq \pi\left(5\frac{(\log p)^4}{(\log\log p)^2}\right) \approx \frac{5\frac{(\log p)^4}{(\log\log p)^2}}{\log(5\frac{(\log p)^4}{(\log\log p)^2})} \in O\left(\frac{(\log p)^4}{(\log\log p)^3}\right)$$

Thus, the size of S can potentially be reduced by a factor $\log\log(p)$, although this would come at the cost of $5\frac{(\log p)^4}{(\log\log p)^2}$ primality tests.

Proposition 4.2.2 (Assuming ERH, Worst-case complexity)**.** *Algorithm 3 runs using at most* $O(\frac{(\log p))^5}{(\log\log p)^2})$ *operations in* F_p

Proof. The prime factors $q_1, q_2, ..., q_r$ of $p-1$ can be found by trial division using $O(B) \subseteq O(\log(p))$ divisions. In the for-loop, residuosity of an element can be checked using Euler's criterion in $O(\log(p))$ field operations, and combined with Theorem (2.2.1), one iteration of the for-loop uses $O((\log p)^3)$ operations. As for the number of iterations, observe that by construction $B\log(B) = 30\log(p)$. Fairly quickly $B < 30\log(p)$, and by the prime number theorem:

$$r \leq \pi(B) \leq \pi(30\log(p)) \in O\left(\frac{\log p}{\log\log(p)}\right)$$

For $1 \leq i \leq r$, the element $b_i^{(p-1)/q_i^{e_i}}$ can be computed in $O(\log(p))$ operations, and thus a can be computed in $O(r\log(p)) \subseteq O(\frac{(\log p)^2}{\log\log p})$ operations. Finally, each element $ab^{(p-1)/t}$ of S can be computed using $O(\log(p))$ operations and thus overall S can be constructed in $O\left(\frac{(\log p))^5}{(\log\log p)^2}\right)$ operations. ■

Chapter 5

Polynomial Factorization

5.1 Overview

In this chapter, the problem of computing the factorization of a class of polynomials over prime fields is discussed. Specifically, it is assumed that the polynomial $f \in F_p[X]$ to be factored is square-free and completely splits over F_p. That is, if $\deg(f) = l > 0$, f can be written as

$$f = (X - r_1)(X - r_2)\dots(X - r_l) \tag{5.1}$$

where $r_1, r_2, ..., r_l \in F_p$ are distinct non-zero roots. The assumption that f is square-free does not pose a problem, since repeated factors can be extracted easily by considering $\gcd(D(f), f)$, where $D(f)$ denotes the formal derivative of f. Moreover, in [Ber70], it is shown that the problem of factoring arbitrary polynomials over general finite fields F_{p^k} can be reduced in deterministic polynomial time to factoring polynomials of the form (5.1).

Significant effort has been spent in the past few decades on developing methods for the problem of polynomial factorization. Two particularly influential contributions are [Ber70] and [CZ81], which both describe probabilistic techniques for factoring general polynomials in expected polynomial-time. In recent years, most of the algorithms that are used in practice are based on the structure of the Cantor-Zassenhaus algorithm introduced in the second work. Typically, the factorization process is divided into three distinct steps:

- Square-free Factorization: the polynomial to be factored is split into square-free factors as described before.
- Distinct Degree Factorization: the square-free input polynomial is split into factors h_i for $i = 1, ..., l$, such that each h_i is the product of irreducible factors of degree i.
- Equal Degree Factorization: the irreducible factors of degree i of each of the h_i are computed.

For polynomials of the form (5.1), the third line of algorithms is clearly applicable with $i = 1$. Significant past contributions to the Cantor-Zassenhaus line of algorithms were made in [VS92], [KS98] and recently in [KU11].

The remainder of this chapter is divided into 3 sections. Section 5.2 is dedicated to factoring methods based on the techniques pioneered and popularized in [Ber67] [Ber70] by Berlekamp, which essentially rely on splitting polynomials on the quadratic residuosity of the roots. Using these methods, we can construct two simple yet effective algorithms for factoring polynomials over general prime fields F_p. The first algorithm is probabilistic of the Las-Vegas variety and runs in expected polynomial-time, while the second algorithm is deterministic and has a polynomial average-case running time. Presently, it appears out of reach to obtain a sub-exponential bound for the worst-case running time, although an analysis of the average-case running time and empirical results in Section 5.4 suggest that in practice the two algorithms have similar performance.

In Section 5.3, the methods introduced in Section 5.2 are generalized to split polynomials based on 2^ith residuosity of roots. These methods have been studied in [Moe77] and subsequently generalized even further in [Rón89] to arbitrary residues. Through these methods, a deterministic algorithm can be constructed that runs in polynomial time when $p-1$ is highly divisible by 2. This algorithm relies on the availability of a primitive root, for which the techniques from Chapter 4 can be applied. In [Gat87], a generalization of this basic construction is used to prove theoretical results regarding polynomial factorization and primitive roots.

All of the algorithms discussed in this chapter have been implemented in Python using the MPyC library [Sch18]. Section 5.4 contains various experimental results demonstrating the practical applicability of the algorithms.

5.2 Berlekamp

Let $f \in F_p[X]$ be a polynomial of degree l of the form (5.1) and let $\alpha \in F_p$. It is easy to see that the *polynomial translation* $f(X-\alpha)$ can be written as

$$f(X-\alpha) = \prod_{i=1}^{l}(X-(\alpha+r_i))$$

Moreover, as a consequence of Lemma 2.1.1:

$$f(X-\alpha) \mid X^p - X = X(X^{\frac{p-1}{2}}+1)(X^{\frac{p-1}{2}}-1)$$

The polynomial $X^{\frac{p-1}{2}}+1$ is the product of those linear factors $(X-r)$ where r is a quadratic non-residue, and similarly $X^{\frac{p-1}{2}}-1$ is the product of those linear factors $(X-r)$ such that r is a quadratic residue. Now since $X \nmid f$, Lemma 2.1.2 provides a factorization of $f(X-\alpha)$ as

$$f(X-\alpha) = \gcd\left(f(X-\alpha), X^{\frac{p-1}{2}}+1\right) \cdot \gcd\left(f(X-\alpha), X^{\frac{p-1}{2}}-1\right) \tag{5.2}$$

Hence, $f(X-\alpha)$ is split based on the quadratic character of $r_1+\alpha, r_2+\alpha, ..., r_l+\alpha$. This idea will be the common basis of the algorithms discussed in the next few sections.

5.2.1 Probabilistic Algorithm

In the previous section, an arbitrary $\alpha \in F_p$ was used to decompose $f(X-\alpha)$ into two factors: one containing the $\alpha+r_i$ such that $\chi(\alpha+r_i)=1$, and one containing those $\alpha+r_i$ with $\chi(\alpha+r_i)=-1$. If α is drawn from a uniform distribution over F_p, one would expect these splits to be approximately balanced. In the worst case, $\alpha+r_1, \alpha+r_2, ..., \alpha+r_l$ all have the same quadratic residuosity and no non-trivial splits are produced, but the chance of this happening is quite slim, as can be seen in the following proposition

Proposition 5.2.1. *Let $x, y \in F_p$ with $x \neq y$ be given, and let α be uniformly distributed over F_p. Then*

$$\mathbb{P}(\chi(x+\alpha) \neq \chi(y+\alpha)) \geq \frac{1}{2}$$

Proof. Let x, y be given, and consider the set $S \subseteq F_p$ defined as

$$S = \{\alpha \in F_p : \chi(x+\alpha) = \chi(y+\alpha)\}$$

Observe that $\chi(x+\alpha)=0$ if and only if $\alpha=-x$, and since by assumption $x \neq y$, this implies $\chi(y+\alpha) \neq 0$. Similarly $\chi(y+\alpha)=0$ implies $\alpha=-y$ and $\chi(x+\alpha)\neq 0$. Combined with Euler's criterion, $\alpha \in S$ if and only if

$$1 = \chi((x+\alpha)(y+\alpha)) = ((x+\alpha)(y+\alpha))^{\frac{p-1}{2}} \tag{5.3}$$

This equation has at most $\frac{p-1}{2}$ solutions, and thus $|S| \leq \frac{p-1}{2} < \frac{|F_p|}{2}$. ■

Using these methods, the main idea behind this algorithm is to split f into two factors h_1, h_2 of potentially lower degree, and apply the same method recursively. As a consequence of Proposition 5.2.1, all roots of f are expected to be recovered eventually. Up until now, the polynomial translation $f(X - \alpha)$ has been used to compute

$$\gcd\left(f(X-\alpha), X^{(p-1)/2} - 1\right)$$

and subsequently decompose f. Using Proposition 2.1.1, this translation could be computed in $O(M(l)\log(l))$ field operations, which is asymptotically the same as the cost of computing the gcd. However, this translation step can be eliminated by instead considering

$$\gcd\left(f, (X+\alpha)^{(p-1)/2} - 1\right)$$

which will result in an algorithm which is slightly simpler and faster in practice. Considering all of the above, the following Las-Vegas type algorithm can be constructed:

Algorithm 4: Factor-Berlekamp-Probabilistic

Input : prime field F_p, monic completely splitting square-free polynomial $f \in F_p[X]$
Output: Set of distinct linear factors f_i

```
1  H ← {f};
2  F ← ∅;
3  while |H| > 0 do
4  |  Pick α ∈ F_p at random;
5  |  for h ∈ H do
6  |  |  d ← gcd(h, (X + α)^((p−1)/2) − 1);
7  |  |  if d ≠ 1 ∧ d ≠ h then
8  |  |  |  Remove h from H;
9  |  |  |  if deg(d) = 1 then F ← F ∪ {d};
10 |  |  |  else H ← H ∪ {d};
11 |  |  |  if deg(d/h) = 1 then F ← F ∪ {d/h};
12 |  |  |  else H ← H ∪ {d/h};
13 |  |  end
14 |  end
15 end
16 return F;
```

Proposition 5.2.2 (Expected cost)**.** *Algorithm 4 runs using an expected $O(M(l)\log(l)\log(p))$ operations and $O(\log(l))$ random elements in F_p.*

Proof. First consider the inner for-loop. Let $h \in H$, and for convenience denote $k = \deg(h)$. Using fast modular exponentiation, a representative of $(X+\alpha)^{(p-1)/2} - 1$ modulo h can be computed using $O(M(k)\log(p))$ operations, after which the gcd can be computed using $O(M(k)\log(k)) \subseteq O(M(k)\log(p))$ operations. Thus, overall one iteration of the inner for-loop uses $O(M(k)\log(p))$ operations.

In Section 2.1, we introduced the assumption on $M(.)$ that $M(a)/a \leq M(b)/b$ when $0 < a \leq b$.

Define $n = \sum_{h \in H} \deg(h)$, and observe that

$$\begin{aligned}\sum_{h \in H} M(\deg(h)) \log(p) &= \log(p) \sum_{h \in H} \deg(h) \frac{M(\deg(h))}{\deg(h)} \\ &\leq \log(p) \frac{M(n)}{n} \sum_{h \in H} \deg(h) \\ &= \log(p) M(n) \\ &\leq \log(p) M(l)\end{aligned}$$

Therefore one iteration of the main while-loop uses $O(M(l) \log(p))$ operations in F_p.

Let I be the random variable denoting the total number of iterations of the outer while-loop, and for $1 \leq i \leq I$, let H_i denote the set H at the beginning of the ith iteration. Let the random variables $I_{a,b}$ denote the last iteration in which the factors $(X - r_a)$ and $(X - r_b)$ were not split apart yet, i.e.

$$I_{a,b} = \max\{i \geq 1 : (X - r_a) \mid h \wedge (X - r_b) \mid h \text{ for some } h \in H_i\}$$

for $1 \leq a, b \leq l$. In any iteration, if two factors $(X - r_a)$ and $(X - r_b)$ have not been split yet, as a consequence of Proposition 5.2.1, they will be split in the next iteration with a probability $\geq \frac{1}{2}$, which implies that

$$\mathbb{P}(I_{a,b} \geq i) \leq \frac{1}{2^{i-1}}$$

for $1 \leq i \leq I$. This implies that

$$\begin{aligned}\mathbb{P}(I \geq i) &\leq \sum_{1 \leq a < b \leq l} P(I_{a,b} \geq i) \\ &\leq \sum_{1 \leq a < b \leq l} \frac{1}{2^{i-1}} \\ &\leq l^2 \frac{1}{2^{i-1}}\end{aligned}$$

Now to prove the proposition, we are interested in $\mathbb{E}[I]$. Since I takes values in $\{1, 2, 3, ...\}$, it is the case that

$$\begin{aligned}\mathbb{E}[I] &= \sum_{i=1}^{\infty} \mathbb{P}(I \geq i) \\ &= \sum_{i=1}^{\log(l^2)} \mathbb{P}(I \geq i) + \sum_{i=\log(l^2)+1}^{\infty} \mathbb{P}(I \geq i) \\ &\leq \log(l^2) + l^2 \sum_{i=\log(l^2)}^{\infty} \frac{1}{2^i} \\ &= 2 \log(l) + l^2 \cdot \frac{2}{l^2} \in O(\log(l))\end{aligned}$$

This completes the proof. ■

To conclude, the methods described in this section provides a simple yet highly effective factorization method that works well for general prime fields F_p. The complexity of Algorithm 4 as shown in Proposition 5.2.2 matches that of the fastest known polynomial factorization algorithms. As a final note, observe that the underlying method for splitting polynomials naturally provides a highly effective method of finding just a single root. If one is not interested in the complete set

of roots, but only a single root, then every time a non-trivial split $f = h_1 h_2$ is produced one may simply recurse on the smaller factor. This idea for finding roots is also explored in [Rab80] and [Ben81]. Due to Proposition 5.2.1, such non-trivial splits occur after $O(1)$ steps. The degree is at least halved in each such step, using $O(\log(l))$ steps overall in expection. Similar to before, a polynomial f can be split using $O(M(\deg(f))\log(p))$ operations, and

$$\sum_{i=0}^{\log(l)} M(\frac{l}{2^i})\log(p) = \log(p)\sum_{i=0}^{\log(l)} \frac{l}{2^i}\frac{M(\frac{l}{2^i})}{\frac{l}{2^i}} \leq \log(p)\frac{M(l)}{l}\sum_{i=0}^{\log(l)} \frac{l}{2^i} \leq 2\log(p)M(l)$$

Therefore, using this method, a single root of f can be found using an expected $O(M(l)\log(p))$ field operations. In practice, the method might produce roots even faster since not all splits will be even.

5.2.2 Deterministic Algorithm

The only aspect of Algorithm 4 which is probabilistic is the choice of random $\alpha \in F_p$. In order to construct a deterministic variant, we may instead start at $\alpha = 1$ and increment it by one every time. This is also briefly suggested in [Ber70], although details are limited. In the previous section, Proposition 5.2.1 could be used to establish an effective upper bound on the expected running time of the algorithm. Unfortunately, no such results are available for the deterministic case at this time, and thus analyzing the deterministic variant of the algorithm requires more work. Nevertheless, the following results due to [Sho90a] do provide a means of establishing reasonable bounds on the running time:

Lemma 5.2.1. *Let $x, y \in F_p$ with $x \neq y$ be given, and suppose that for $N \in \mathbb{N}$*

$$\chi_p(xy) = \chi_p((x+1)(y+1)) = ... = \chi_p((x+N)(y+N)) = 1$$

Then $N < p^{1/2}\log(p)$.

Lemma 5.2.2. *Consider the finite field F_p. The number of pairs $(x, y) \in F_p \times F_p$ such that*

$$\chi_p(x+i) = \chi_p(y+i)$$

for $i = 0, ..., \log(p) - 1$ is bounded from above by $p\log(p)^2$.

Algorithm 5: Factor-Berlekamp-Deterministic

Input : prime field F_p, monic completely splitting square-free polynomial $f \in F_p[X]$
Output: Set of distinct linear factors f_i

```
1  H ← {f};
2  F ← ∅;
3  α ← 0;
4  while |H| > 0 do
5  |  for h ∈ H do
6  |  |  d ← gcd(h, (X + α)^((p−1)/2) − 1);
7  |  |  if d ≠ 1 ∧ d ≠ h then
8  |  |  |  Remove h from H;
9  |  |  |  if deg(d) = 1 then F ← F ∪ {d};
10 |  |  |  else H ← H ∪ {d};
11 |  |  |  if deg(d/h) = 1 then F ← F ∪ {d/h};
12 |  |  |  else H ← H ∪ {d/h};
13 |  |  end
14 |  end
15 |  α ← α + 1
16 end
17 return F;
```

Proposition 5.2.3 (Worst-case cost)**.** *Algorithm 5 runs using at most $O(M(l)p^{1/2}\log(p)^2)$ operations in F_p.*

Proof. Using the same arguments as in the proof of Proposition 5.2.2, one iteration of the outer while-loop of Algorithm 5 uses $O(M(l)\log(p))$ operations in F_p. Now, suppose that the outer while-loop requires $N+2$ iterations, such that in the end the element $\alpha = N+1$ produced a split. This implies that there exist two factors $(X-r_a)$ and $(X-r_b)$ of f such that $\alpha = 1, 2, ..., N$ failed to produce a split. This implies that

$$\chi(r_a r_b) = \chi((r_a+1)(r_b+1)) = ... = \chi((r_a+N)(r_b+N)) = 1$$

and the result follows from Lemma 5.2.1 ■

This only establishes an exponential bound on the running time of the algorithm, but unfortunately no sub-exponential bounds for the worst-case cost appear to be within reach at the moment. The problem is further discussed in [BKS15]. Nevertheless, in the next result it can be seen that Algorithm 5 behaves well on average and should be effective in practice. This is further confirmed by the experimental results provided in Section 5.4.

Proposition 5.2.4 (Average-case cost)**.** *Let f be uniformly distributed over the monic polynomials of degree l with l non-zero distinct roots. Algorithm 5 runs using an expected $O(M(l)\log(p)(\log(p)+\frac{l^2(\log(p))^3)}{p^{1/2}}))$ operations in F_p.*

Proof. Let N be a random variable denoting the total number of iterations of the outer while-loop. From Lemma 5.2.2, it follows that if $r_a, r_b \in F_p$ are random, the probability that $(X-r_a)$ and $(X-r_b)$ take more than $\log(p)$ iterations to split is bounded from above by $\frac{\log(p)^2}{p}$. The input f has l such factors and thus

$$\mathbb{P}(N > \log(p)) \leq \frac{l^2\log(p)^2}{p}$$

Moreover, $N < p^{1/2}\log(p)$ as a consequence of Lemma 5.2.1. Since N takes value in $\{1, 2, 3, ...\}$, it is the case that

$$\begin{aligned}\mathbb{E}[N] &= \sum_{i=1}^{\infty} \mathbb{P}(N \geq i) \\ &= \sum_{i=1}^{\log(p)} \mathbb{P}(N \geq i) + \sum_{i=\log(p)+1}^{\infty} \mathbb{P}(N \geq i) \\ &\leq \log(p) + \sum_{i=\log(p)+1}^{p^{1/2}\log(p)} \frac{l^2 \log(p)^2}{p} \in O(\log(p) + l^2 \frac{\log(p)^3}{\sqrt{p}})\end{aligned}$$

which completes the proof. ■

5.3 Moenck

The factorization method of the previous section is essentially based on splitting a polynomial based on quadratic residuosity. This can be generalized to splitting based on 2^ith residuosity, which is suggested in [Moe77]. Let s, t be such that $p - 1 = 2^s t$, with t odd. For $0 \leq i \leq s$, the polynomial $c_i \in F_p[X]$ defined as

$$c_i = X^{(p-1)/2^i} - 1$$

is the product of those linear factors $X - r$ such that r is an 2^ith power residue as a consequence of Proposition 2.2.1. The main idea is to use this family of polynomials (c_i) to split a polynomial f.

5.3.1 Reduction

Let $0 \leq i < s$, and suppose that $h \in F_p[X]$ is some polynomial with the property that all of its roots are 2^ith residues. Define $u_i \in F_p[X]$ as:

$$u_i = \gcd(h, c_{i+1})$$

This polynomial contains those roots that are also 2^{i+1}th power residues. On the other hand, the polynomial $v_i = \frac{h}{u_i}$ contains those roots of h that are 2^ith power residues but 2^{i+1}th power non-residues. Now if g is a primitive root, this means that v_i can be written as

$$v_i = \prod (X - g^{j2^i})$$

where the j are odd. Now consider the polynomial $\hat{v_i}$, defined as v_i with all of its roots multiplied by g^{2^i}, i.e.

$$\hat{v_i} = \prod (X - g^{(j+1)2^i}) = \prod_{j \text{ odd}} (X - g^{2^{i+1}(j+1)/2}) \tag{5.4}$$

The roots of $\hat{v_i}$ are all 2^{i+1}th power residues, and moreover, if we find a root of $\hat{v_i}$, then we also obtain a root of v_i by dividing out the factor g^{2^i}. Overall, we have reduced the problem of factoring a polynomial h with roots that are 2^ith power residues, to the problem of factoring two polynomials with roots that are 2^{i+1}th power residues. This procedure can be applied repeatedly, until eventually we need to factor polynomials where all roots are 2^sth residues. This collection of elements is generated by $g^{(p-1)/t}$ and consists of t elements. If t is small, say $t \in O(\log(p))$, a brute-force approach becomes a feasible approach for this last step. Alternatively, the algorithmic ideas presented in [Rón89] could be used to obtain a more efficient yet more complex method for this last step.

5.3.2 Algorithm

An important step of the method described in the previous section is the construction of the polynomial $\hat{v_i}$ based on v_i. While the relation (5.4) is expressed in terms of the roots, this transformation can in fact be performed when v_i is represented as a coefficient list as a consequence of the following proposition:

Proposition 5.3.1. *Let $v \in F_p[X]$ be a monic polynomial of degree k represented as $v = \sum_{j=0}^{k} X^j a_j$, and let $\alpha \in F_p^*$ be arbitrary. Suppose that v has k distinct roots in F_p. If we define the polynomial $\hat{v} \in F_p[X]$ as*

$$\hat{v} = \sum_{j=0}^{k} X^j \alpha^{k-j} a_j$$

then $\hat{v}$ has k distinct roots, and these roots $\hat{r_j}$ are related to the roots r_j of v as $\hat{r_j} = \alpha r_j$ for $j = 0, ..., k-1$.

Proof. This follows directly from Vieta's formulas relating the coefficients and roots of a polynomial. ■

The method described in the previous section, combined with Proposition 5.3.1, would lend itself well to a recursive implementation. Nevertheless, for the sake of comparison with earlier factorization algorithms and also to ease analysis of later experimental results, an iterative version of the algorithm is presented here. For the time being, we will assume that a primitive root g is

simply provided to the algorithm. This will be addressed in the next section.

Algorithm 6: Moenck

Input : prime field F_p, monic completely splitting square-free polynomial $f \in F_p[X]$, primitive root $g \in F_p$
Output: Set of distinct linear factors f_i

Let s, t be such that $p - 1 = r^s t$, with $\gcd(r, t) = 1$.
$H \leftarrow \{(f, 1)\}$;
$F' \leftarrow \emptyset$;
Compute the powers $g^2, g^{2^3}, ..., g^{2^s}$ and store these in a table G.;
Set $\xi \leftarrow g^{2^s}$, compute $\xi^2, \xi^3, ..., \xi^{t-1}$ and store these in table T;
for $i \leftarrow 0$ *to* $s - 1$ **do**
  for $(h, d) \in H$ **do**
    $u \leftarrow \gcd(h, X^{(p-1)/2^{i+1}} - 1)$;
    Compute $v \leftarrow \frac{h}{u}$, and construct $\hat{v}$ by multiplying all of the roots of by g^{2^i} using Proposition 5.3.1
    Remove (h, j) from H;
    if $\deg(u) = 1$ **then** $F' \leftarrow F' \cup \{(u, d)\}$;
    else if $\deg(u) > 1$ **then** $H \leftarrow H \cup \{(u, d)\}$;
    if $\deg(v) = 1$ **then** $F' \leftarrow F' \cup \{(v, d \cdot g^i)\}$;
    else if $\deg(v) > 1$ **then** $H \leftarrow H \cup \{(\hat{v}, d \cdot g^i)\}$;
  end
end
for $(h, d) \in H$ **do**
  Find all linear factors of h by trying elements of T, and insert these together with d into F'.
end
$F \leftarrow \emptyset$;
for $(X - r, d) \in F'$ **do**
  $F \leftarrow F \cup \{X - \frac{r}{d}\}$
end
return F;

Proposition 5.3.2 (Worst-case complexity)**.** *Algorithm 6 runs using at most* $O(lt + M(l)\log(p)s)$ *field operations in* F_p.

Proof. It is assumed that the decomposition $p - 1 = r^s \cdot t$ is given. The powers of g for table G can be computed using $O(s)$ squarings. Next, the powers of ξ for table T can be then be computed using $O(t)$ multiplications.

In each iteration of the inner for-loop, one can first compute $X^{(p-1)/2^{i+1}}$ modulo h in $O(M(\deg(h))\log(p))$ operations using fast modular exponentation. Next, the GCD can be computed in $O(M(\deg(h))\log(\deg(h))$ operations, while v can be computed in $O(M(\deg(h))$ operations. Based on v, one can compute $\hat{v}$ using Proposition 5.3.1 in $O(\deg(v)) \subseteq O(\deg(h))$ operations. Finally, observe that $d \cdot g^i$ can be computed in $O(1)$ multiplication by using table G. Overall, one iteration of the inner for-loop uses $O(M(\deg(h))\log(p))$ operations. Through a similar argument as was used in the proof of

Proposition 5.2.2, we find

$$\begin{aligned}\sum_{h\in H} M(\deg(h))\log(p) &= \log(p)\sum_{h\in H}\deg(h)\frac{M(\deg(h))}{\deg(h)}\\ &\leq \log(p)\frac{M(n)}{n}\sum_{h\in H}\deg(h)\\ &= \log(p)M(n)\\ &\leq \log(p)M(l)\end{aligned}$$

and thus one iteration of the first outer for-loop uses $O(M(l)\log(p))$ operations, and $O(sM(l)\log(p))$ for the first outer for-loop overall. For the second for-loop, observe that a polynomial h can be evaluated at a point using $O(\deg(h))$ operations using Horner's method such that one iteration uses $O(\deg(h)t)$ operations. Moreover

$$\sum_{h\in H}\deg(h)t = t\sum_{h\in H}\deg(h) \leq t\cdot l$$

and therefore in this way, overall the second for-loop uses $O(lt)$ operations. Finally, the last for-loop uses $O(l)$ operations to transform the roots. ■

This result shows that the particular prime p being used has a significant impact on the efficiency of Algorithm 6. If t large, the exponential-time exhaustive search approach could potentially make the algorithm infeasible to use. However, for certain fields, for instance when $t \in O(\log(p))$, the algorithm provides an effective deterministic polynomial-time option for factoring polynomials. Proposition 5.3.2 also shows that Algorithm 6 becomes faster as s increases and thus t decreases. However, notably the exact opposite is true for the Adleman-Manders-Miller algorithm discussed in Section 3.2, which requires small s in order to be competitive.

If a primitive root g is given, Algorithm 6 could be optimized by performing several pre-computations. Specifically, tables G and T could be pre-computed beforehand. This could improve the practical speed of the algorithm, although asymptotically the result from Proposition 5.3.2 remains unchanged. In addition, the exhaustive search procedure could be replaced by using the efficient multipoint evaluation techniques described in Section 2.1. This way, the $O(lt)$ term in Proposition 5.3.2 would be changed to $O(M(n)\log(n))$ where $n = \max\{l, t\}$.

5.3.3 Primitive roots

In the previous section, a primitive root in F_p was assumed to be available for the factorization method. If t is small, say $t \in O(\log(p)$, the complete prime factorization of $p-1$ can be computed quickly and primitive roots are readily available. On the other hand, if $p-1$ contains large prime factors, fully factoring $p-1$ becomes infeasible and instead the methods described in Chapter 4 can be used, which is explored in this section.

First, consider the case in which t is small. Using Algorithm 6 as a substep the following algorithm can fully factor polynomials in $F_p[X]$ without requiring any given elements.

Algorithm 7: Factor-Moenck-1

Input : prime field F_p, monic completely splitting square-free polynomial $f \in F_p[X]$
Output: Set of distinct linear factors f_i

1 Compute the prime factorization $p-1 = q_1^{e_1}\cdot\ldots q_k^{e_k}$;
2 $a \leftarrow 1$ **for** $i = 1$ *to* k **do**
3 | Try $b_i = 1, 2, ...$ until a q_ith power non-residue is found;
4 | $a \leftarrow a b_i^{(p-1)/q_i^{e_i}}$
5 **end**
6 return MOENCK(f, a)

As a consequence of Proposition 2.2.1, the search for a b_ith power non-residue will succeed after at most $2(\log p)^2$ attempts. In each attempt, Euler's criterion could be used to test for non-residuosity in $O(\log p)$ operations. Combined with Proposition 5.3.2, it is clear that Algorithm 7 uses at most $O((\log p)^3 + lt + M(l)\log(p)s)$ field operations and therefore, if $t \in O(l + \log p)$, polynomials can be factored in deterministic polynomial time. This is an improvement as compared to the result obtained for Algorithm 5, which could only be shown to be polynomial-time when considering the average over all polynomials.

Through the methods for obtaining primitive root candidates described in Section 4, this algorithm can be extended to deal with finite fields in general. One may repeatedly compute a candidate primitive root and attempt the factorization. This strategy is given in Algorithm 8.

Algorithm 8: Factor-Moenck-2

Input : prime field F_p, monic completely splitting square-free polynomial $f \in F_p[X]$
Output: Set of distinct linear factors f_i

Let a be computed as in Algorithm 3
$F \leftarrow \emptyset$
$h \leftarrow f$
$b \leftarrow 1$
while $h \neq 1$ **do**
  $L \leftarrow \text{MOENCK}(f, ab^{(p-1)/t})$
  $F \leftarrow F \cup L$
  $h \leftarrow h / \prod_{l \in L} l$
  $b \leftarrow b + 1$
end
return F;

While this algorithm does work for all finite fields F_p, the exhaustive search approach used in Algorithm 6 can potentially incur a significant computational cost if t is large, which is unacceptable in certain situations. However, as will be demonstrated in the experimental results in the next section, many polynomials can be completely factored through the first component of Algorithm 6, avoiding this costly step entirely. In certain applications, one could even opt to remove the final exhaustive search and return a "Failure" result, attempting factorization with some other algorithm such as for instance Algorithm 5.

Finally, it should be noted that the method described in this section allows for several future improvements. Specifically, both Algorithm 7 and Algorithm 8 compute several prime factors $q_1, q_2, ..., q_k$ of $p-1$ beyond 2. Instead of immediately opting for an exhaustive search near the end of Algorithm 6, one could instead refine the existing factorization even further by splitting roots based on q_ith residuosity for $i = 1, ..., k$. This could further increase the practical effectiveness of Algorithm 8, although polynomials will remain for which an exhaustive search is unavoidable.

5.4 Experimental Results

In this section, experimental results are presented for the algorithms discussed in Sections 5.3 and 5.2. These have been implemented in Python using the MPyC library [Sch18]. In order to measure the computational speed of these algorithms, a collection of random monic polynomials of degree l is first generated. The various factorization algorithms are used on this collection and the average computation times are reported. All results were obtained with a desktop system equipped with an Intel i5-4670K processor and 32GB of memory.

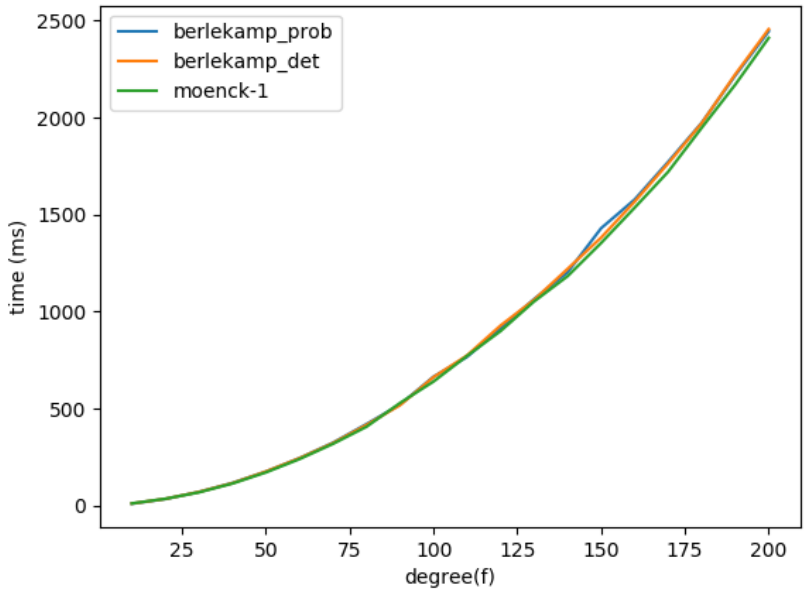

Figure 5.1: Computation times in milliseconds needed to factor polynomials for the prime $p = 34803817920319193089$ with $p - 1 = 2^{56} \cdot 483$. The horizontal axis denotes the degree l of the input polynomials.

	20	40	60	80	100	120	140	160	180	200
Berl-Prob										
Time (ms)	34.53	117.02	245.05	420.68	664.68	914.83	1203.17	1578.11	1973.43	2445.16
# steps	8.94	11.24	12.18	13.11	13.6	14.16	14.47	14.74	15.14	15.71
Berl-Det										
Time (ms)	35.11	116.32	244.80	416.60	662.66	928.76	1220.73	1567.23	1971.29	2456.49
# steps	9.07	10.95	11.95	13.05	13.51	14.06	14.74	14.83	15.15	15.64
Moenck-1										
Time (ms)	35.56	114.16	239.99	406.72	639.89	899.89	1182.91	1535.88	1947.61	2411.29
# steps	9.1	11.13	12.08	13.05	13.75	14.07	14.46	15.26	15.41	15.43

Table 5.1: Computation times in milliseconds and number of steps needed to factor polynomials for the prime $p = 34803817920319193089$ with $p - 1 = 2^{56} \cdot 483$.

Figure 5.1 and Table 5.1 suggest that when s is large, such that the first variant of the Moenck algorithm is applicable, all of the factorization algorithms behave in a similar manner in terms of computation speed. Note that the polynomial arithmetic implemented in MPyC is of the order $M(l) \in \Theta(l^2)$, which explains the quadratic curve visible in Figure 5.1. While for large s, the Moenck algorithm appears to perform as well as the Berlekamp algorithms, further analysis of the experimental results reveals that the exhaustive search step included in the algorithm is never actually invoked. On probabilistic grounds, one would indeed expect the three algorithms to behave similarly in this case. In particular, they are all expected to produce fairly balanced splits such that the number of steps needed is approximately $O(\log(l))$. If s is small, the exhaustive search starts to contribute significantly the computation time of the Moenck algorithm, which can be seen in Figure 5.2. Note that the measurements for the Moenck shown in Figure 5.2 do not include the cost of computing the prime factorization of $p - 1$, in order to demonstrate the effect of the exhaustive search procedure.

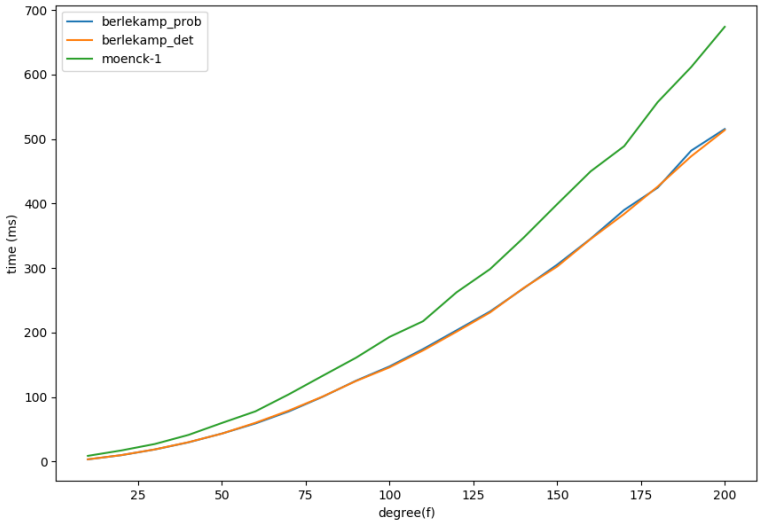

Figure 5.2: Computation times in milliseconds needed to factor polynomials for the prime $p = 6753281$ with $p - 1 = 2^{10} \cdot 6595$. The horizontal axis denotes the degree l of the input polynomials. The time needed to fully factor $p - 1$ is not included for the Moenck algorithm.

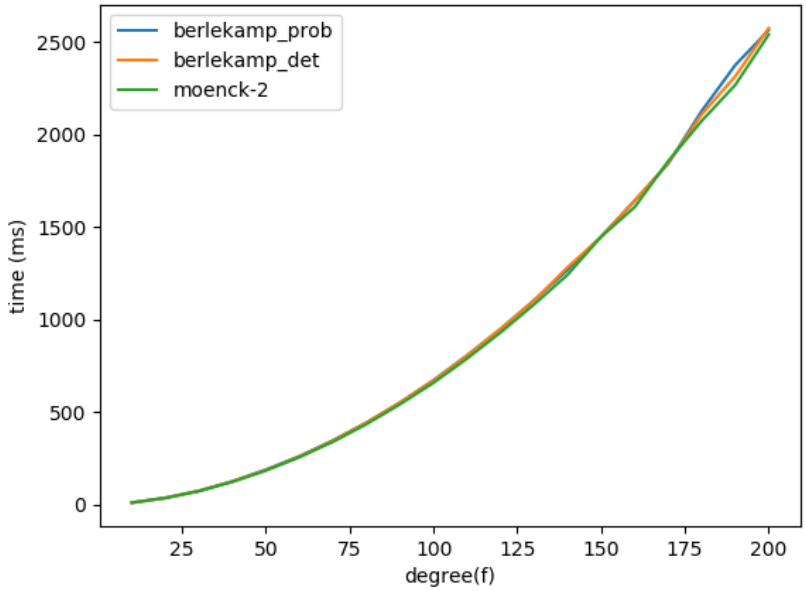

Figure 5.3: Computation times in milliseconds needed to factor polynomials for the prime $p = 18630133447762378753$ with $p - 1 = 2^{33} \cdot 2168832981$. The horizontal axis denotes the degree l of the input polynomials.

The results in Figure 5.3 and Table 5.2 demonstrate that even when t is larger, the factorization method based on Moenck can still be effective when combined with a proper scheme for generating primitive root candidates. In this case, computing the complete prime factorization of $p - 1$ is infeasible and thus the Moenck-2 algorithm should be used instead of Moenck-1. Note however that similar to the results shown in Figure 5.1, the exhaustive search procedure is never invoked.

	20	40	60	80	100	120	140	160	180	200
Berl-Prob										
Time (ms)	38.07	126.81	262.63	444.47	673.16	949.63	1269.25	1642.30	2126.71	2563.72
# steps	9.17	10.9	12.26	12.72	13.89	13.93	14.37	15.01	15.05	15.59
Berl-Det										
Time (ms)	37.26	126.75	260.78	443.40	670.65	950.17	1283.38	1642.22	2108.38	2574.23
# steps	8.72	11.02	11.8	12.57	13.6	14.21	14.57	14.69	15.23	15.32
Moenck-2										
Time (ms)	36.94	124.54	257.40	435.85	659.16	930.96	1242.33	1607.33	2073.03	2539.52
# attempts	1	1	1	1	1	1	1	1	1	1

Table 5.2: Computation times in milliseconds and number of steps or attempts needed to factor polynomials for the prime p with $p - 1 = 1863013344776237875 3 = 2^{33} \cdot 2168832981$. Averaged over 100 random polynomials.

The results in Table 5.2 also show that in practice, the two variants of the Berlekamp algorithm have similar behaviour with respect to the number of steps needed. While currently the best known bound on the worst-case complexity of the deterministic Berlekamp variant is exponential, these results show that the algorithm is effective in practice.

The factorization methods discussed in this chapter all essentially use divide and conquer approach. In each step, the factorization is further refined until at last only linear factors remain. Most of the factors in the last few steps many of the remaining factors will be of a low degree, e.g. 2, 3 or 4, and thus one potential improvement could be to handle such cases using the quadratic, cubic and quartic formulas to directly compute the roots. These can be expressed in terms of square roots and cube roots, which can in turn be computed using the techniques and algorithms from Chapter 3.

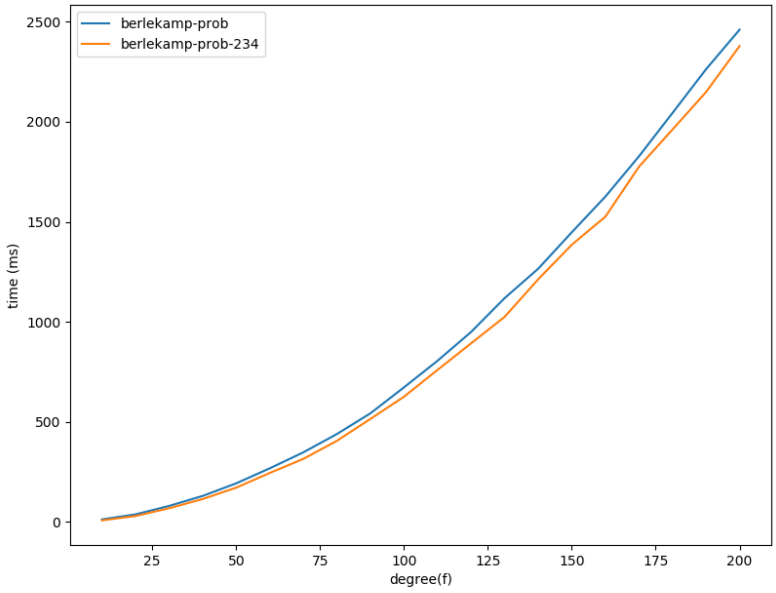

Figure 5.4: Computation times in milliseconds needed to factor polynomials for the prime $p = 34997513354692853761$ with $p - 1 = 2^{16}3^2 \cdot 5 \cdot 11867103866473$. The horizontal axis denotes the degree l of the input polynomials.

Figure 5.4 shows the effect of incorporating the quadratic, cubic and quartic formulas into the probabilistic Berlekamp algorithm. Square roots and cube roots are computed using the Adleman-Manders-Miller algorithm. Figure 5.4 suggests that the direct formulas can contribute to improving the efficiency of practical factorization methods, albeit only slightly.

Finally, note that these formulas could also be used in the Moenck factorization algorithms, including the exhaustive search procedure. However, one should take care to choose the appropriate algorithms for computing square roots and cube roots. The complexity analysis and experimental results presented in Chapter 3 for the Adleman-Manders-Miller algorithm indicate that the algorithm performs significantly worse as s increases. This is the opposite of the Moenck-1 algorithm, which requires a sizeable s to function properly. Instead, the Peralta algorithm discussed in Section 3.3 could be used for square roots, along with an appropriate cube root algorithm. For instance, the Padró-Sáez algorithm introduced in [PS02] and later refined in [Heo+14] could be used for this purpose. This is generalization of the Peralta algorithm for the case of cube roots.

Chapter 6

Application: Secure shuffle

In this chapter, we discuss an interesting application of polynomial factorization in secure multiparty computation (MPC). In [Vre20], the author presents MPC protocols for a variety of tasks, such as shuffling a list of secret-shared values and sorting a list of secret-shared values. The former task can in fact be divided into two independent subtasks, which are of interest in their own right as well:

- Generating a random, secret-shared permutation
- Rearranging a list of secret-shared values according to a secret-shared permutation

In [Vre20] the author also suggests applications for the secure shuffle protocol, such as in online card games. We will briefly describe both of these protocols and their use of polynomial factorization. The conventions and notations from [Vre20] are used, with $[\![x]\!]$ denoting a secret-shared variable x and reveal($[\![x]\!]$) denoting the act of revealing the value of x to all involved participants. We will assume that basic MPC primitives, such as multiplying or adding two secret-shared numbers, are given.

6.1 Random permutations

Let $N > 0$ be some integer, and let $[\![\hat{r}]\!]$ be a list of N distinct, secret-shared elements in F_p. Define the secret-shared polynomial $f \in F_p[X]$ as

$$f = \prod_{i=1}^{N} (X - \hat{r}_i)$$

Since multiplication in $F_p[X]$ is commutative, revealing f to all parties does not reveal any information about the original order of the secret-shared elements $[\![r_i]\!]$. If all participants fully factor f, they obtain a new list $r = [r_{i_j}]$ containing the same values but with the original order lost. By mandating a common ordering of r between participants, the pair $(r, [\![r]\!])$ establishes a random secret-shared permutation. This is the main idea behind the protocol, which is provided

in algorithmic form in Algorithm 9:

Algorithm 9: Random-Permutation

Input : Integer $N > 0$
Output: list r of length N, secret-shared list $[\![\hat{r}]\!]$ of length N

Generate N secret-shared random values, and store these in a list $[\![\hat{r}]\!]$
$[\![T]\!] \leftarrow \prod_{i=1}^{N}(X - [\![\hat{r}_i]\!])$
$f \leftarrow \text{reveal}([\![T]\!])$
$r \leftarrow \text{Find-Roots}(f)$
if $r[i] = r[j]$ *for any* $i \neq j$ **then**
  return "Failure"
else
  return (r, $[\![\hat{r}]\!]$)
end

Here Find-Roots() denotes any suitable algorithm for determining the roots of a monic polynomial, such as those presented in Chapter 5. In order for Algorithm 9 to function correctly, the involved parties need to ensure that their lists r are matching, i.e. have the same order. In [Vre20], the author indicates that any arbitrary order will work and proposes that at the end of the protocol, participants locally sort r through a regular sorting algorithm. Using results from Chapter 5, an alternative solution which would avoid this last step would be to use the deterministic factorization algorithm presented in Section 5.2.2 for obtaining r. Although this algorithm has an exponential worst-case complexity, the average-case complexity is comparable to that of existing probabilistic algorithms which is further supported by the experimental results in Section 5.4. This eliminates the need for a dedicated sorting step (which typically runs using $O(N \log(N))$ comparisons), since all participants necessarily obtain the same order. The resulting protocol is faster in practice, although asymptotically the computational complexity is still dominated by the factorization step.

During the execution of Algorithm 9, the same work on factoring the polynomial f is effectively repeated by all of the participants. A potential optimization to the protocol would therefore be to assign the work to one of these participants, which could share the resulting roots with the remaining participants when finished. If necessary, all of the participants can also easily verify this received factorization, which is significantly more efficient than computing the complete prime factorization.

6.2 Shuffling

The remaining components of the Secure Shuffle protocol are provided below (see [Vre20], Chapter 3 for an in-depth derivation and analysis).

Suppose a list $[\![v]\!]$ of N secret-shared values is given. By generating a random permutation through Algorithm 9 and applying the following algorithm for rearranging $[\![v]\!]$ according to this permutation, we can securely shuffle $[\![v]\!]$.

Algorithm 10: Rearrange

Input : Secret-shared list $[\![v]\!]$ of length N, secret-shared permutation $(r, [\![\hat{r}]\!])$ of size N
Output: Secret-shared rearranged list

$[\![f]\!] \leftarrow \text{Interpolate}(r, [\![v]\!])$
return $\text{Eval}([\![f]\!], [\![\hat{r}]\!])$

Here, Interpolate() refers to a polynomial interpolation algorithm, such as those mentioned in Chapter 2, and similarly Eval() to a regular multi-point evaluation algorithm. The complete

shuffling algorithm is now as following:

Algorithm 11: Shuffle

Input : Secret-shared list $[\![v]\!]$ of length N
Output: Secret-shared shuffled list

1 $(r, [\![\hat{r}]\!]) \leftarrow$ Random-permutation(N)
2 return Rearrange($[\![v]\!], (r, [\![\hat{r}]\!])$)

Previously, implementations provided by the author of [Vre20] relied on components written using the NTL C++ library [Sho+01], using the *FindRoots* function from NTL for the factorization step. Using the factorization algorithms previously described in Chapter 5, the complete Secure Shuffle protocol has been implemented in Python using the MPyC library [Sch18]. Below, several experimental results are provided for the speed of this implementation when using the probabilistic Berlekamp factorization method. Here N denotes the size of the list to be shuffled, while M denotes the number of participants in the protocol. These results have been obtained on a desktop computer equipped with Intel i5-4670K processor and 32GB of memory, where multiple parties are simulated with communication running over local tcp connections.

$N \rightarrow$	5	10	15	20	25	30	35
M=2	0.26	0.45	0.98	2.07	4.31	6.62	9.73
M=5	0.51	1.32	3.16	7.44	14.76	27.86	45.45
M=10	2.23	3.52	9.26	23.79	50.43	90.25	131.50

Table 6.1: Running the Secure Shuffle protocol for varying N, and with a varying number of participants. Entries indicate time in seconds needed for the protocol to complete. This implementation uses basic non-optimized Lagrange interpolation methods for Algorithm 10, and similarly, Horner's Rule is used to evaluate $[\![f]\!]$ in each root.

Chapter 7

Conclusion

In this thesis we discussed several algorithms that can be used for the computational problem of polynomial factorization and more specifically root finding. The modular root algorithms discussed in Chapter 3 are shown to be very effective in practice, even on low-cost computer systems.

Using probabilistic algorithms, polynomials can be factored over arbitrary finite fields efficiently in expected polynomial time. In particular, the Berlekamp algorithm discussed in Chapter 5 runs using $O(M(l)\log l \log p)$ expected field operations, which matches the bounds of the most efficient known factorization methods. By making some very minor changes, this algorithm can be converted into a deterministic algorithm. Presently, proving a sub-exponential bound on the worst-case running time of this deterministic algorithm appears to be out of reach, although the method is shown to be effective when the average over all polynomials is considered. A sub-exponential upper bound would have a significant impact, since this would allow us to construct deterministic polynomial-time solutions for a variety of computational problems for which the only known efficient methods are currently probabilistic, or rely on unproven conjectures.

We propose two variants of the Moenck factorization method introduced in [Moe77]. Through the first algorithm, polynomials over certain finite fields can be factored in deterministic polynomial time. Using the techniques for determining primitive root candidates presented in Chapter 4, this method is extended to deal with more general finite fields. Although this algorithm is not guaranteed to be efficient for all polynomials, experimental results demonstrate that the resulting algorithm can still be effective in practice. Finally, we present an interesting application in secure multiparty computation with the recently proposed Secure Shuffle protocol, along with experimental results using the MPyC library [Sch18]. These demonstrate that the protocol can be run even on low-cost computers, although optimizations to the implementation are likely necessary in order for the protocol to be effective in real-world applications such as card games.

7.1 Future work

This thesis presents several possibilities for future research. A few ideas are listed below:

- The factorization algorithms discussed in Chapter 5 use a divide-and-conquer approach and could potentially be parallelized. This would be particularly interesting in secure multiparty computation applications as it could be set up as to divide up the work among the participants. If the deterministic methods from Section 5.2 or Section 5.3 are used, communication between participants could be minimized.
- As part of Algorithm 8, the partial prime factorization of $p-1$ is computed in order to find proper primitive root candidates. Using these prime divisors q_i, the algorithm could be generalized to also split polynomials based on q_ith residuosity, since a (candidate)primitive root is already given. In addition, the exhaustive search procedure could be significantly improved by using the direct root formulas for degree 2, 3 and 4 polynomials.

Bibliography

[Ank52] Nesmith Cornett Ankeny. "The least quadratic non residue". In: *Annals of mathematics* (1952), pp. 65–72.

[Leh59] Emma Lehmer. "On Euler's criterion". In: *Journal of the Australian Mathematical Society* 1.1 (1959), pp. 64–70.

[Ber67] Elwyn R Berlekamp. "Factoring polynomials over finite fields". In: *Bell System Technical Journal* 46.8 (1967), pp. 1853–1859.

[Ber70] Elwyn R Berlekamp. "Factoring polynomials over large finite fields". In: *Mathematics of computation* 24.111 (1970), pp. 713–735.

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[AMM77] Leonard Adleman, Kenneth Manders and Gary Miller. "On taking roots in finite fields". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE Computer Society. 1977, pp. 175–178.

[Moe77] Robert T Moenck. "On the efficiency of algorithms for polynomial factoring". In: *Mathematics of computation* 31.137 (1977), pp. 235–250.

[HW79] Godfrey Harold Hardy and Edward Maitland Wright. *An introduction to the theory of numbers*. Oxford university press, 1979.

[Sha79] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[Rab80] Michael O Rabin. "Probabilistic algorithms in finite fields". In: *SIAM Journal on computing* 9.2 (1980), pp. 273–280.

[Ben81] Michael Ben-Or. "Probabilistic algorithms in finite fields". In: *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*. IEEE. 1981, pp. 394–398.

[CZ81] David G Cantor and Hans Zassenhaus. "A new algorithm for factoring polynomials over finite fields". In: *Mathematics of Computation* (1981), pp. 587–592.

[Yao82] Andrew C Yao. "Protocols for secure computations". In: *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE. 1982, pp. 160–164.

[Per86] Rene Peralta. "A simple and fast probabilistic algorithm for computing square roots modulo a prime number (Corresp.)" In: *IEEE transactions on information theory* 32.6 (1986), pp. 846–847.

[CK87] David G Cantor and Erich Kaltofen. "Fast multiplication of polynomials over arbitrary rings". In: *Acta Inf* 28 (1987).

[Gat87] Joachim von zur Gathen. "Factoring polynomials and primitive elements for special primes". In: *Theoretical Computer Science* 52.1-2 (1987), pp. 77–89.

[Rón89] Lajos Rónyai. "Factoring polynomials modulo special primes". In: *Combinatorica* 9.2 (1989), pp. 199–206.

[Bac90a] Eric Bach. "A note on square roots in finite fields". In: *IEEE Transactions on Information Theory* 36.6 (1990), pp. 1494–1498.

[Bac90b] Eric Bach. "Explicit bounds for primality testing and related problems". In: *Mathematics of Computation* 55.191 (1990), pp. 355–380.

[Sho90a] Victor Shoup. "On the deterministic complexity of factoring polynomials over finite fields". In: *Information Processing Letters* 33.5 (1990), pp. 261–267.

[Sho90b] Victor Shoup. "Searching for primitive roots in finite fields". In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 1990, pp. 546–554.

[CK91] David G Cantor and Erich Kaltofen. "On fast multiplication of polynomials over arbitrary algebras". In: *Acta Informatica* 28.7 (1991), pp. 693–701.

[Atk92] A Oliver L Atkin. "Probabilistic primality testing". In: *INRIA Res. Rep* (1992), pp. 159–163.

[VS92] Joachim Von Zur Gathen and Victor Shoup. "Computing Frobenius maps and factoring polynomials". In: *Computational complexity* 2.3 (1992), pp. 187–224.

[Ros95] Michael I Rosen. "Niels Hendrik Abel and equations of the fifth degree". In: *The American Mathematical Monthly* 102.6 (1995), pp. 495–505.

[BS96] Eric Bach and Jeffrey Outlaw Shallit. *Algorithmic number theory: Efficient algorithms*. Vol. 1. MIT press, 1996.

[Bac97] Eric Bach. "Comments on search procedures for primitive roots". In: *Mathematics of Computation* 66.220 (1997), pp. 1719–1727.

[KS97] Erich Kaltofen and Victor Shoup. "Fast polynomial factorization over high algebraic extensions of finite fields". In: *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*. 1997, pp. 184–188.

[KS98] Erich Kaltofen and Victor Shoup. "Subquadratic-time factoring of polynomials over finite fields". In: *Mathematics of computation* 67.223 (1998), pp. 1179–1197.

[Sho99] Peter W Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". In: *SIAM review* 41.2 (1999), pp. 303–332.

[Ber01] Daniel J Bernstein. "Faster square roots in annoying finite fields". In: (2001).

[Sho+01] Victor Shoup et al. *NTL: A library for doing number theory*. 2001.

[PS02] Carles Padró and Germán Sáez. "Taking cube roots in Zm". In: *Applied mathematics letters* 15.6 (2002), pp. 703–708.

[ZG02] Joachim von Zur Gathen and Jürgen Gerhard. "Polynomial factorization over F2". In: *Mathematics of Computation* 71.240 (2002), pp. 1677–1698.

[Mül04] Siguna Müller. "On the computation of square roots in finite fields". In: *Designs, Codes and Cryptography* 31.3 (2004), pp. 301–312.

[Van05] Christiaan Van de Woestijne. "Deterministic equation solving over finite fields". PhD thesis. Universiteit Leiden, 2005.

[Bog+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter et al. "Secure multiparty computation goes live". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2009, pp. 325–343.

[Sho09] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

[KU11] Kiran S Kedlaya and Christopher Umans. "Fast polynomial factorization and modular composition". In: *SIAM Journal on Computing* 40.6 (2011), pp. 1767–1802.

[Bre13] Richard P Brent. *Algorithms for minimization without derivatives*. Courier Corporation, 2013.

[VG13] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[Heo+14] Geon Heo, Seokhwan Choi, Kwang Ho Lee, Namhun Koo and Soonhak Kwon. "Remarks on the Pocklington and Padró–Sáez cube root algorithm in Fq". In: *Electronics letters* 50.14 (2014), pp. 1002–1003.

[BKS15] Jean Bourgain, Sergei Konyagin and Igor Shparlinski. "Character sums and deterministic polynomial root finding in finite fields". In: *Mathematics of Computation* 84.296 (2015), pp. 2969–2977.

[Sch18] Berry Schoenmakers. *MPyC: Secure Multiparty Computation in Python*. GitHub. github.com/lschoe/mpyc. May 2018.

[Vre20] Niels de Vreede. "Assorted algorithms and protocols for secure computation". PhD thesis. Netherlands: Eindhoven University of Technology, 2020.