Elliptic Curve Cryptography for Developers

Mike Rosing

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Elliptic Curve Cryptography for Developers

Version 6

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *Elliptic Curve Cryptography for Developers.*

Today's blockchain applications utilize advanced protocols which include the pairing of points over elliptic curves. To fully understand and appreciate why this mathematics is so useful is exceptionally challenging. This book explains how to convert the mathematics of elliptic curves over finite fields into code which you can use for your cryptographic applications.

The cryptographic protocols described in this book are used in state of the art blockchain applications. Two specific applications are described in the last two chapters of the book. To get to the point where you can understand those applications requires learning

- elliptic curves over finite fields
- arithmetic of point addition using elliptic curves
- polynomial arithmetic using polynomials as a modulus
- elliptic curves over field extensions
- how to find good elliptic curves which can be used for point pairings
- arithmetic of point addition using elliptic curves over field extensions
- computation of pairing points over field extension elliptic curves

Each chapter in the book teaches some aspect of this list. In each chapter you will learn some math and be introduced to subroutines which can compute that math.

In the process of writing this book, I read many cryptographic papers. Usually over a dozen times. So while I think I'm explaining things correctly, I would really appreciate your comments on how to improve descriptions which seem confusing. I want people to write advanced mathematical software with confidence, so your careful reading of *Elliptic Curve Cryptography for Developers* will help make that happen.

My background includes 40+ years in embedded systems starting with assembler and ending with VHDL. For most 16 and 32 bit processors I wrote code in C. The code in this book is aimed at small embedded systems that require high security. C is not strongly typed so it is trivial to convert integers into bytes and vice versa. This is exceptionally useful when converting a pass phrase on an embedded system into a private key – the same pointer means two different things depending on which subroutines are called.

For those who want to convert the code into Verilog or VHDL, the GNU Multi Precision library are the core routines to replace. Using fixed width integers, and an appropriate choice of prime field, very efficient and highly secure applications can be created in hardware. I would really like to know about your experience with putting elliptic curve mathematics into an FPGA.

Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum.

—Mike Rosing

brief contents

1 Pairings over elliptic curves in cryptography

PART 1: BASICS

2 Description of finite field mathematics

3 Explaining the core of elliptic curve mathematics

4 Key exchange using elliptic curves

5 Prime field elliptic curve digital signatures explained

6 Finding good cryptographic elliptic curves

PART 2: INTERLUDE

7 Description of finite field polynomial math

8 Multiplication of polynomials explained

9 Computing powers of polynomials

10 Description of polynomial division using Euclid's algorithm

11 Creating irreducible polynomials

12 Taking square roots of polynomials

PART 3: PAIRINGS

13 Finite field extension curves described

14 Finding low embedding degree elliptic curves

15 General rules of elliptic curve pairing explained

16 Weil pairing defined

17 Tate pairing defined

18 Exploring BLS multi-signatures

19 Proving knowledge and keeping secrets, zero knowledge using pairings

1 Pairings over elliptic curves in cryptography

This chapter covers

- Defining elliptic curve cryptography
- Where ECC is used
- Public key cryptography
- Who this book is for

I first became interested in elliptic curve cryptography (ECC) in the mid 1990s. I was involved with an activist organization working to legalize marijuana and we wanted to ensure our member list was secure and that we could email each other with encrypted messages. While I was aware of PGP (pretty good privacy), as a scientist at Argonne National Laboratory I wanted something more state of the art. So I dug into the papers and went to a few CHES (Cryptographic Hardware and Embedded Systems) conferences to learn how to write my own code.

Today ECC is ubiquitous. For example, EMVCo (Europay, Mastercard, and Visa) support ECC in their cryptographic interface for credit card transactions. In addition to sharing secret keys as in an SSL/TLS handshake it is also used for signing certificates to authenticate web pages and HIPPA documents. A blockchain holds globally accessible information shared among many peers. The blockchain has found use of elliptic curve pairings to enable aggregate group signatures as well as zero knowledge proofs. The impetus for

blockchain may be cryptocurrency, but the technology has many more uses. Blockchain ledgers are used in healthcare and supply chain environments.

ECC has been used for random number generation too. However, this takes a lot more resources than linear feedback shift registers or hardware sources like zener diodes. So there are a few places where ECC can be replaced by a better alternative.

Learning ECC math is still challenging to understand, especially for those who do not have a Ph.D. in mathematics. The mathematics behind pairings at first glance appear exceptionally deep. Most developers find ways to not include pairings in their products. Once the basics are understood, pairing based elliptic curve cryptography will also become ubiquitous. The learning curve is not steep if you know the right path. This book follows a very narrow path that is essential to follow from beginning to end. You will understand how to compute elliptic curve pairings used in the last two chapters by following each chapter one at a time.

1.1 What is elliptic curve cryptography?

First - there are no ellipses. Second - there are no curves. So why is it called an "elliptic curve"? The primary reason is history. Performing integrals of the elliptical orbits of planets gave rise to formulas that were labeled elliptic curves. These formulas were then used in other areas of mathematics and the label stuck.

The areas of mathematics used in public key cryptography involve number theory (the properties of integers), combinatorics (the study of counting) and finite fields (sets with finite objects and specific rules). The areas of mathematics where elliptic curves are used is just about everything. This makes studying elliptic curve math very difficult because it is hard to determine what you really need to know and what is just really interesting.

For example, elliptic curves have been used for factoring numbers and for solving Fermat's Last Theorem. In chapter 3 I explain how elliptic curves on the complex plane are used to understand elliptic curves over finite fields. It's all interesting, but not necessarily applicable to cryptography.

1.2 Why use elliptic curve cryptography

Public key cryptography started with the RSA system, which uses exponentiation modulo very large primes. Algorithms for breaking RSA are subexponential, so several thousand bit primes are needed for standard security. The advantage of using elliptic curve mathematics is the reduced size of the numbers involved for the same level of security. Larger numbers require more memory, more processing time, more gates on an FPGA or more processors in a GPU. That means the resources required (or the cost of the system) is higher for other methods. That reduced cost is one of the main drivers for the use of elliptic curve public key cryptography.

In the past decade, other properties of elliptic curves have allowed applications which are not even possible with other methods. Aggregate digital signatures are a primary example. And while zero knowledge proofs were first introduced with other mathematics, the elliptic curve versions are much shorter to transmit.

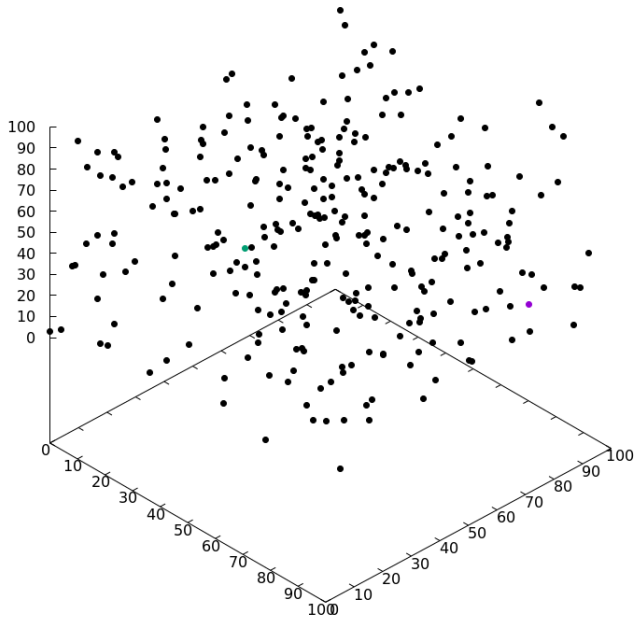

Figure 1.1 Conceptual elliptic curve over finite field, points are scattered in a multidimensional, astronomically large space

Figure 1.1 is a conceptual view of an elliptic curve over a finite field. All the points are "on the curve". That means they all satisfy the elliptic curve equation. One point is colored red, one is colored green. These points are mathematically related to each other using algorithms described in this book.

Discovering the mathematical relationship is called breaking the algorithm or cracking the code. For elliptic curves of high enough field size and dimension the cracking effort would require ten million 4 GHz processors about one billion years to find the relationship. In part 3 of this book I dive into the details of what high enough field size and dimension means.

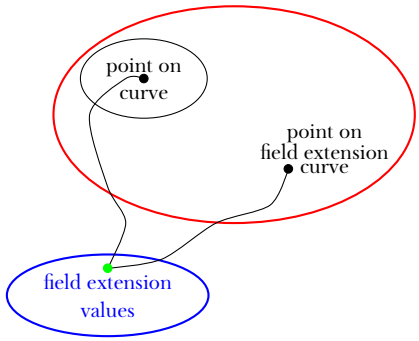

Figure 1.2 Schematic of elliptic curve pairing with one point on base curve, one point on field extension curve, and result in field extension values

Figure 1.2 is an outline of elliptic curve pairings, which I cover in part 3 of the book. The red area represents an elliptic curve on a field extension. Buried within that are points on a base curve. A pairing of points on this curve results in a field extension value (technically, an n^{th} root of unity). Only elliptic curve mathematics gives rise to cryptographically secure pairings. The relationship between pairing values and points on curves is more difficult to crack than the relationship between points alone. This is a fundamental reason for using elliptic curves in cryptography - it is very hard to break the algorithms.

1.3 Elliptic curves come to public key cryptography

The use of elliptic curves for cryptography is a very new development in the history of keeping secrets. From ancient Rome to the present day secure messages are sent using a secret key which both the sender and receiver know. Transmission of this secret key to both parties was always a major problem. In the 1970s the idea of using one way trap door functions which are easy to compute and essentially impossible to unravel introduced the concept of public key cryptography.

The fundamental idea behind public key cryptography is the use of a private key held by one person or organization and a public key transmitted openly by that person. Two people can create a shared secret that no one else can determine so encrypted messages can be exchanged. This is especially useful if they have never met.

Additional applications in the digital age have also been developed. A person can sign a document with their private key and anyone can verify they signed it using the signer's public key. The combination of multiple signatures is also possible where many people or computer servers sign the same document. Zero knowledge proofs allow transactions to be anonymous and verifiable at the same time which also requires a private key to prove existence and a public key to verify. In the following sections I will expand on these concepts and the rest of the book will describe how to implement this in reality using elliptic curve mathematics.

1.3.1 General description of Key exchange

To securely send messages over a public network we want to use a very strong method of encryption. The National Institute of Standards and Technology (NIST) defined the Advanced Encryption Standard (AES) with several levels of security depending on the length of the secret key. A key exchange between two parties is performed using public key cryptography so that a shared secret can be created using open networks.

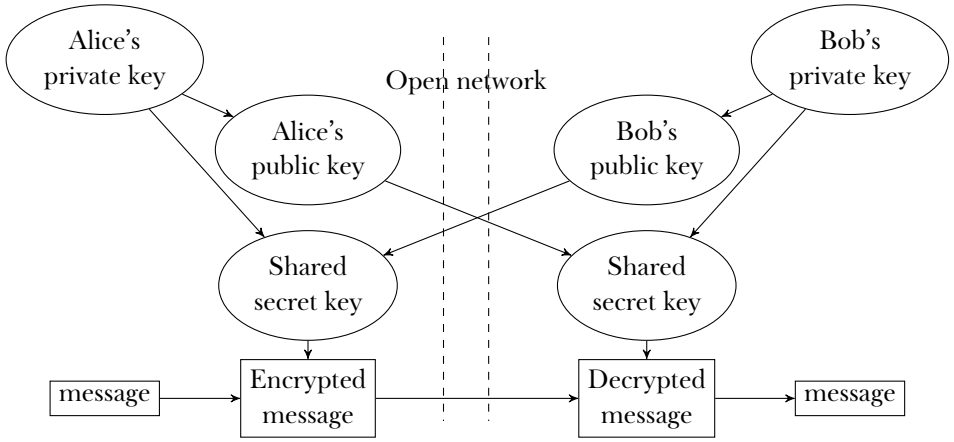

Figure 1.3 Creating secret key by exchanging public keys

Suppose Alice and Bob want to securely exchange messages. They each create a private key which they hold secretly so no one else can know it. Using the private key, they each create a public key, which they share with each other. In figure 1.3 we show how Alice and Bob can send their public keys over an open network. Alice combines her private key with Bob's public key to create a single secret key for a system like AES used to encrypt a message. The advantage of AES is that it is fast and can encrypt large amounts of data easily. The disadvantage is that it requires a single secret key.

Bob uses Alice's public key along with his private key to create the same shared secret that Alice used to encrypt the message. What is seen on the public network are Alice's public key, Bob's public key and the encrypted message.

The point of this system is that Alice and Bob don't have to agree on what secret key to use before they decide to communicate. As we'll see later, in addition to creating a shared secret, we can make the secret ephemeral so that even if someone were capable of breaking the message key, they would not be able to break Alice's or Bob's public keys to discover their private key.

1.3.2 Digital signature algorithms explained

To prove that she created a document and to ensure that no one else can modify it, Alice can create a digital signature of the document. She uses her private key to sign the document and anyone else can use her public key to verify that she signed it. If anyone changes even one bit in the document the signature will not verify.

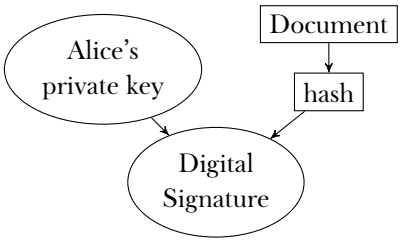

Figure 1.4 Processes involved with creating a digital signature

Figure 1.4 shows how digital signatures are signed and figure 1.5 how they are verified. Alice uses her private key along with a hash of the document to create a digital signature that is posted along with the document in a public place.

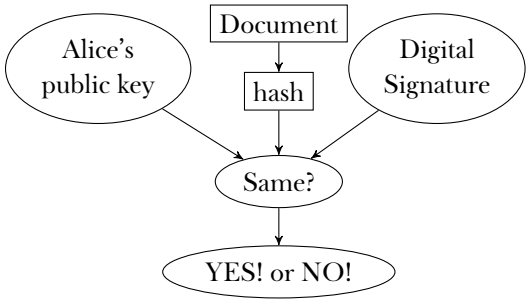

Figure 1.5 Process and data required to verify a digital signature

Anyone can then use Alice's public key, the Document and the digital signature to verify that Alice is indeed the person who created the document. Anyone else attempting to impersonate Alice would need to know her private key. As figure 1.5 shows, a valid document outputs positive result for the correct public key.

1.3.3 How multiple people can sign the same document

If a document requires several people to sign it, a simple digital signature of each person signing the same document becomes complicated. Each person's key has to be used to verify their signature. This increases the storage area required because every digital signature needs to be attached to the document.

Figure 1.6 looks complicated, but there are really only three steps.

1. Aggregate all the public keys into one block.
2. Each person digitally signs the document plus the aggregated keys into their own signature data.
3. Mathematically combine all the signatures into one final signature.

Figure 1.7 shows how the aggregated signature is verified. The hash of the public keys can be recreated or saved with the document. That is a system level time versus space argument, we will assume it as an input. An elliptic curve pairing operation is done using the public keys, the document and the combined final signature to determine if everyone

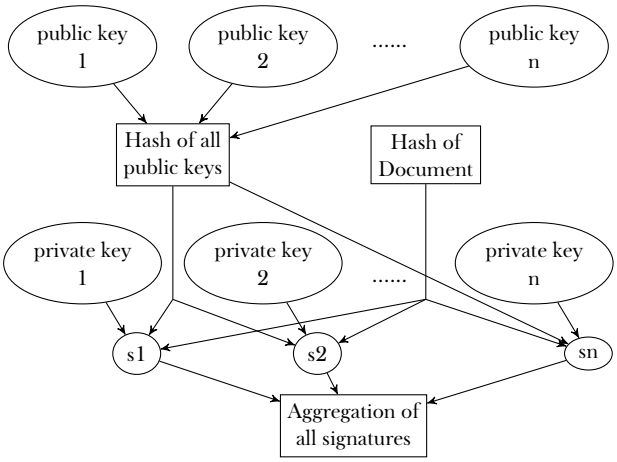

Figure 1.6 Aggregated digital signature signing combines multiple users public keys and individual signatures to create a single signature

did in fact sign the same document. When the computations match each other (we'll get into the details in later chapters) the composite signature verifies.

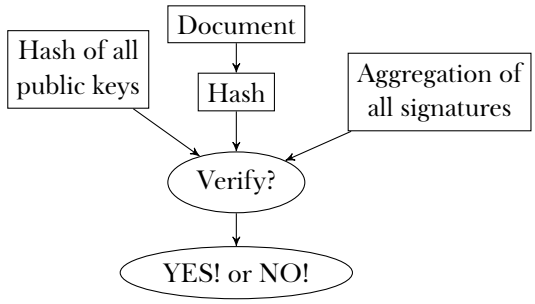

Figure 1.7 Aggregated digital signature verification process

The ability to verify an aggregated digital signature requires the use of elliptic curve pairings. Parts 2 and 3 of this book build up the mathematical background required to understand all the details of elliptic curve pairings.

1.3.4 Zero knowledge, or how to keep a secret and prove you have one

A digital certificate is used to verify data is authentic. An example is a web page that has been certified as genuine by some trusted authority. It can also contain personal data which someone may not wish to be exposed. The idea that we can prove an entity has some private information but not expose that information is called a zero knowledge proof.

A huge surveillance state was envisioned if every digital certificate was traceable to every issuer and owner. To eliminate this lack of privacy a way to verify that a prover actually knows what they claim without giving away any information was developed. So the prover knows what they want to keep private in the certificate and the verifier wants to check the

certificate is valid.

Today we want transactions on a blockchain to be anonymous as well as keeping amounts of money transferred a secret. The original method of zero knowledge proof was interactive with the prover and verifier sending messages back and forth. If the probability of being correct was 50%, after 20 tries the chances of the prover misleading the verifier would be less than one in a million.

The fundamental problem with an interactive method is the communication between prover and verifier. So the next step in solving the problem was to introduce non-interactive zero knowledge proofs. Figure 1.8 shows how a non-interactive proof is set up using public data and a public common reference string. The public data is usually information

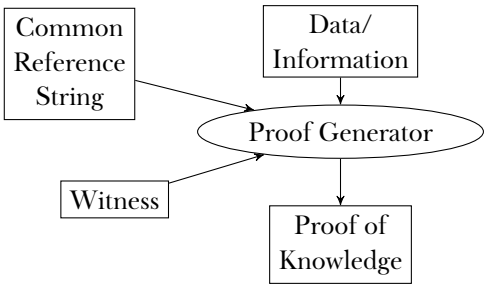

Figure 1.8 Requirements involved to create zero knowledge proof

contained on a blockchain. The common reference string refers to the elliptic curve parameters used to hide the data. The witness shown in figure 1.8 is private information, which can be a combination of public keys and coin data and/or address key information. The combined mathematical result is called a proof of knowledge.

The verification process uses the public information, the public common reference string and the public proof of knowledge to verify that the information is correct. This is shown diagrammatically in figure 1.9. If the data and proof are from the same blockchain, the verification will be a YES result, otherwise one gets a NO result. The idea is to make generation of the proof a time-consuming process (but not too long) and the verification to be a very short time. We also want the proof to be small and not take up too much space on the blockchain.

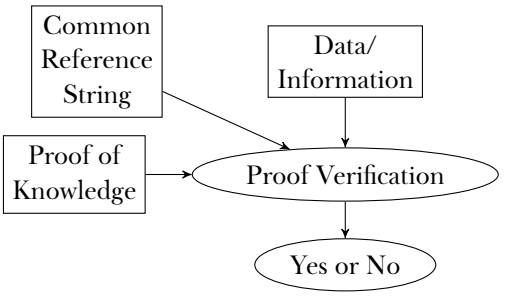

Figure 1.9 Process of verifying a zero knowledge proof

The acronym developed for zero knowledge proofs is SNARK, which stands for Succinct Non-interactive ARgument of Knowledge. After reading many mathematical papers on SNARK's, I found Lewis Carroll's "The Hunting of the Snark" (Carroll & Singh, 2010) to make about as much sense. It was not a surprise to find a cryptographic article with the same title (Bitansky *et al.*, 2014). These constructions use elliptic curve pairing operations which allow the succinct part to happen.

1.4 Who this book is for

If you don't have an advanced degree in mathematics and you want to understand enough elliptic curve math to implement common algorithms, then this book is for you. If you don't have a good grasp of linear algebra and are not familiar with manipulation of polynomials, then this book might be too much. The learning process will be step by step, but the first step assumes you know how to manipulate equations with several variables.

This book will teach the basics of elliptic curves over finite fields to show how the math applies to cryptography. The book will also teach the fundamentals of elliptic curve pairing mathematics. Pairings on elliptic curves are a deep subject with many tangents that have nothing to do with cryptography and much to do with mathematics. Some topics were tried for cryptography, and have been rejected because the system was "broken" by careful analysis.

By the time you finish this book you will be able to implement secure transfer of keys using elliptic curves of your own design as well as create pairing based digital signature schemes for use on blockchain systems. You will have confidence in your ability to test the cryptographic underpinnings of your code.

So, let's get started!

1.5 Summary

- Finite fields are sets with specific rules and a finite number of objects. The mathematics of finite fields determines how we manipulate elliptic curves.
- Elliptic curves over finite fields are good for cryptography because they use smaller numbers for the same level of security than other public key methods.
- Key exchange and digital signatures are straightforward using elliptic curve mathematics through the use of simple formulas. This makes programming the mathematics easy.
- Aggregation of many digital signatures using pairings over elliptic curves allows for smaller storage on a blockchain than other methods. This reduces the amount of data transmitted between peers which decreases over all transaction time.
- Zero knowledge proofs can certify knowledge of information without exposing that information. Zero knowledge proofs use reduced amounts of data compared with other methods when using elliptic curve pairing mathematics.

Chapter Bibliography

Bitansky, Nir, Canetti, Ran, Chiesa, Alessandro, Goldwasser, Shafi, Lin, Huijia, Rubinstein, Aviad, & Tromer, Eran. 2014. *The Hunting of the SNARK*. Cryptology ePrint Archive, Paper 2014/580. https://eprint.iacr.org/2014/580. 9

Carroll, L., & Singh, M. 2010. *The Hunting of the Snark*. Melville House. 9

Part 1

Basics

The chapters in part 1 of the book cover finite field arithmetic, elliptic curve mathematics and cryptographic primitives. Finite fields based on prime numbers are the essence of elliptic curve cryptography. They form the basis of everything else that follows.

Large integers consisting of 160 to more than 500 bits make up the finite fields used in cryptography. Large integer libraries have been around for a long time and the one I chose for this book is called the GNU Multiple Precision Arithmetic Library or GMP. There are many routines not included within GMP which are required for elliptic curve implementations. These are discussed in chapter 2. One of the major routines covered includes taking square roots using a modulus. There are several routines which will be mentioned that use only a few calls to the GMP library. These routines are used throughout all the code in this book, so while they are simple, they are very important.

Chapter 3 dives into the elliptic curve mathematics. The idea of algebra on an elliptic curve is described along with some abstract pictures. The ideas are important and having some mental image can help you understand the mathematics. I discuss the idea of embedding values onto a curve, then describe code to add and multiply points.

This part includes two chapters on applications of elliptic curves over finite fields. Chapter 4 covers key exchange and chapter 5 covers digital signatures. There are dozens of key exchange and signature algorithms to choose from. Each chapter only discusses two of the most commonly used algorithms. Once you understand how one algorithm works with elliptic curve cryptography, you will have no trouble implementing similar algorithms.

The last chapter in this part goes into the process of finding good curves. For the large prime fields this book is interested in, this process is not that difficult to do using the mathematical tools which have been developed over the last few decades. I also have a short discussion on how to avoid bad curves. For all the methods shown in this part, the odds of finding a good curve are reasonably high if you search long enough (as in 24 hour computer runs).

2 Description of finite field mathematics

This chapter covers

- Fundamentals of finite fields
- Subroutines for modular operations
- Concept of quadratic residues
- Computing square roots mod n

Fields are mathematical objects which obey specific rules. Finite fields have a fixed number of objects. This chapter introduces finite fields over prime numbers and the code we will need in rest of the book to manipulate these objects.

In this chapter I start at the bottom of finite field mathematics by using prime numbers to define the core concept of a finite field. Every formula in this book depends on prime numbers to create a finite field. I'll first go over the rules of finite fields we need to know and then introduce simple subroutines that exploit the GNU Multiple Precision Arithmetic Library (GMP library) to implement some of those rules. The library is exceptionally useful for very large integers used in cryptographic protocols. You can learn about retrieving the GMP library and its documentation from appendix A.

The general equation of an elliptic curve is given by

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

Fortunately for us, we don't need the general curve because we work over finite fields. The ordinary equation used throughout this book has $a_1 = a_2 = a_3 = 0$.

To find the y value from the equation of an elliptic curve we need to compute square roots. After the simple code, I discuss a more complicated problem of taking square roots modulo a prime. This involves finding quadratic residues to determine if a square root even exists for a number. GMP has a library routine to help with that. Then I will discuss the algorithm used to find square roots once we know it can be found. With that description the code is straightforward to implement.

2.1 Basic mathematics of finite fields

In this section I dive into the fundamentals of finite fields over prime numbers.

A field is a very special term in mathematics. It describes objects which can be added, multiplied and divided. Two special objects **0** and **1** are the identity elements for addition and multiplication respectively. Identity just means $a + \mathbf{0} = a$ for addition and $b * \mathbf{1} = b$ for multiplication.

The types of objects which make up fields include numbers, polynomials and polynomials of polynomials! So a field is a mathematical abstraction which has very specific rules that apply to all these things.

A finite field has a finite number of objects. The field of integers is infinite. We can always add 1 and get the next number. Because objects in a field can multiply and add any object to any other object to get yet another object in the field, a finite field must be cyclic.

Your first introduction to a finite field was learning how to tell time on a clock. As shown in figure 2.1, adding 6 hours to 9 o'clock you get 3 o'clock. On a 24-hour clock you would get 15 o'clock. We say that 15 is congruent to 3 modulo 12. So the size of the field determines what the results are. We say a 12-hour clock is modulo 12 and a 24-hour clock is modulo 24. The term congruent will also show up in many places, and it means "equal to" given the modulus.

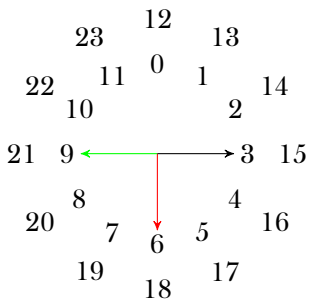

Figure 2.1 Adding 6 + 9 modulo 12 and modulo 24

For cryptography, the size of the field is astronomical because a field size of 256 bits is a number 2^{256} which is roughly the number of atoms in the observable universe.

The problem with a clock is that the number 12 has three factors: $2 * 2 * 3$. We can

create sets of numbers that are even, or factors of 4, or factors of 3, or factors of 6. Each of those sets maintains a cyclic relationship. So the set of numbers {2, 4, 6, 8, 10, 12} form a cyclic group. But because it does not have the identity element, it is not technically a field.

Figure 2.2 illustrates the idea of a prime field using 43 as the prime number. It is drawn like a clock so when counting to 43 we return to 0. Since both 0 and 1 are in the set of numbers, addition and multiplication are possible in this field. Computing an inverse modulo a prime number is also possible. That is what makes a prime number base into a field.

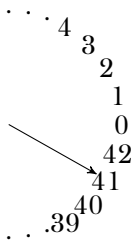

Figure 2.2 Finite field over prime numbers viewed as a clock ($p = 43$)

For cryptography, we want finite fields with really large prime numbers. For embedded systems we want numbers that are easy to compute with. Later on I'll give examples where I choose primes like $43 \times 2^{252} + 1$. Because this set of numbers contains both 0 and 1 it creates a finite field with $43 \times 2^{252} + 1$ numbers in it. A secret key will be one of those numbers. If code is written properly - good luck to an adversary finding it!

2.1.1 Elliptic curves form groups of points over a finite field

In this section I cover how an elliptic curve over a finite field creates a new kind of finite field.

The points on an elliptic curve over a finite field form cyclic sets. Sometimes these sets are disjoint with two groups that do not overlap. I go over this type of curve in chapter 13 on field extensions. Sets of points in each cyclic group will have many subgroups which depend on the number of factors making up the number of points on the curve. The different possible combinations of factors create the number of groups which can be formed.

As an example, suppose the total number of points on the curve has three factors. The rules about these sets of points create seven total groups of cyclic points. One group for each combination of the factors. So factors a, b and c create group sizes a, b, c, $a \cdot b$, $a \cdot c$, $b \cdot c$ and $a \cdot b \cdot c$. Many of the points will be in multiple groups.

Typically, if we pick a point at random it will be in the largest group. We can move that point to a smaller subset group by multiplying by the other factors. For example, suppose c is a really large prime with $a = 2$ and $b = 5$. If we multiply the random point by 10 the result will be a point in group c. This is one of the finite field group rules we take advantage of with elliptic curve mathematics. Many more details will be spelled out in chapter 3.

Exercise 1

Given an elliptic curve with 86 points, list the orders of all the groups on this curve.

2.2 Basic subroutines for finite field arithmetic

In this section I describe subroutines which make the use of the GMP library easier for the remainder of the book.

The fundamental, lowest level routines are placed in a file called `modulo.c`. The header showing all the routines is given in listing 2.1. There are two groups of routines. The first group requires a modulus as part of the API. The second group assumes a local static variable has been set for the modulus.

Listing 2.1 Header: `modulo.h`

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

Basic include for all GMP polynomial functions

```
void mod_add(mpz_t a, mpz_t b, mpz_t c, mpz_t n);
void mod_sub(mpz_t a, mpz_t b, mpz_t c, mpz_t n);
void mod_mul(mpz_t a, mpz_t b, mpz_t c, mpz_t n);
void mod_div(mpz_t a, mpz_t b, mpz_t c, mpz_t n);
void mod_neg(mpz_t a, mpz_t b, mpz_t n);
```

requires modulus on input

```
void minit(mpz_t m);
void mget(mpz_t mod);
```

setup for internal modulus

```
void mset(mpz_t prm);
void madd(mpz_t a, mpz_t b, mpz_t c);
void msub(mpz_t a, mpz_t b, mpz_t c);
void mmul(mpz_t a, mpz_t b, mpz_t c);
void mdiv(mpz_t a, mpz_t b, mpz_t c);
void minv(mpz_t a, mpz_t b);
void mneg(mpz_t a, mpz_t b);
int msqrt(mpz_t x, mpz_t a);
void mrand(mpz_t rand);
int msqr(mpz_t x);
void mpowi(mpz_t a, mpz_t b, long i);
```

requires modulus initialized

Since this header is included in all other files, the standard C includes are listed along with GMP. As you can see, all the subroutines begin with `m` which means they belong to the modulo group of routines.

The GMP library has many types of functions. I exclusively use the `mpz_*` functions, which are for large integers. The type `mpz_t` is GMP's integer type. The majority of arguments used in listing 2.1 type `mpz_t`.

The built-in routines to GMP include integer functions for add, subtract, multiply and divide. There are many versions with a lot of different combinations of arguments. Chap-

ter 5 of the GMP manual goes into this in detail. One of the GMP division routines is `mpz_mod()`. I combine this with add, subtract, multiply and divide to get `mod_*()` routines. All my routines follow the same format as GMP. The output is listed first and the input arguments to the function follow. This comes from thinking about the function as $a = b < op > c$ modulo n.

Let's take a look at the modulo division routine in listing 2.2. This has a few more lines than all the others because division by zero should never happen if your code is correct. If it does happen we want to know about it so we can find the bug.

Listing 2.2 Modular division: `mod_div()`

```
void mod_div(mpz_t a, mpz_t b, mpz_t c, mpz_t n)
{
  mpz_t rslt;

  mpz_init(rslt);
  if(!mpz_invert(rslt, c, n))   ← division by zero?
  {
    printf("division by zero in div_mod!\n");
    mpz_clear(rslt);
    exit(-1);
  }
  mpz_mul(rslt, b, rslt);
  mpz_mod(a, rslt, n);          common to
  mpz_clear(rslt);              all m* routines
}
```

Because the result could be one of the input arguments, we need to create a place for the result to go so the inputs are not clobbered as we do computations (you can bet I learned that the hard way.) The `mpz_init()` routine creates space for the variable. On an embedded system or an FPGA you may want to use fixed field sizes and specialized routines which take advantage of the modulus. These lowest level routines are the best place to do this because all the number crunching gets down to this level sooner or later.

The next function called is `mpz_invert()`. The inversion of c modulo n uses Euclid's algorithm which we will get into later when we deal with polynomials. If c is zero, the function prints an error and kills the program. Division by zero implies a serious problem somewhere, and we need to find out why.

The last three lines of the code are similar in all the `mod_*` routines. A call to `mod_<op>()` followed by a call to `mpz_mod()` and then clearing the temporary variable space. Another efficiency step might be to set up the temporary space during initialization and just let it be global to all the routines. This saves a lot of `mpz_init()` and `mpz_clear()` calls.

Most of the time the modulus used in all the routines is fixed at a single prime number. Rather than have to list that with every call, we can store it as a static variable local to all the modulo routines. As we get higher up the chain, we sometimes need to know what that prime number is for different operations. So the routines `minit()` and `mget()` respectively initialize and get the modulus for all the `m*()` functions. At the top of the modulo.c file I

put the globals:

```
static mpz_t modulus;
static gmp_randstate_t state;
```

In the middle of the `modulo.c` file are the initialization routines as shown in listing 2.3.

Listing 2.3 Modular initialization routines

```
void minit(mpz_t m)
{
  mpz_init_set(modulus, m);    | set modulus
  gmp_randinit_mt(state);      | and random state
}

void mget(mpz_t mod)
{
  mpz_init_set(mod, modulus);  ← return local modulus
}
```

You'll notice there is no `mclear()` function. If your system needs to clear the modulus, it should be obvious how to just call `mpz_clear(modulus)` with a void function. The example code does not require this function, but real life might.

In addition to setting the local modulus value, the `mpz` random number generator is initialized. This is used to pick random values for probabilistic algorithms like the square root routine shown later.

There is a negate operation which simply performs $a = -b \bmod n$. The advantage of this happens when b is larger than n (or could be) and we have the chance to reduce it in size before using it in other operations.

2.3 Computing quadratic residues over a prime field

In this section I describe testing if a square root is possible to compute for a number in a finite field.

Taking square roots of real numbers is easy. Newton's method has worked for hundreds of years. But taking square roots in a finite field is quite a bit different. The idea is simple: $x^2 = a \bmod n$, we know a, we want to find x. But some numbers will not have square roots mod n. Numbers that have square roots are called quadratic residues. All the remaining numbers are simply called nonresidues.

Number theory has been studied for a very long time. In 1640 Fermat wrote down what is now called Fermat's Little Theorem. It is one of the most fundamental properties of number theory and was used as the first attempt at finding square roots mod n. We can paraphrase the version I.3.2 in (Koblitz, 1994): for any prime p and any integer $a < p$ we have

$$a^{p-1} \cong 1 \bmod p \tag{2.1}$$

More than 100 years later, Legendre wrote down a symbol for quadratic residues. Today

we call this the Legendre symbol, which is defined in II.2.3 of (Koblitz, 1994) to be

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue mod } p \\ -1 & \text{if } a \text{ is a nonresidue mod } p \end{cases}$$

A number which is a nonresidue mod p will not have a square root with respect to p. As an example the square root of 2 mod 23 is 5 because $5^2 \bmod 23 = 2$. The Legendre symbol is

$$\left(\frac{2}{23}\right) = 1.$$

Similarly the Legendre symbol

$$\left(\frac{17}{23}\right) = -1$$

says there is no square root of 17 mod 23.

Fortunately for us, GMP has a Legendre symbol routine which we can use to check for residue or nonresidue as the case may be. Using the local `modulus` variable, we have the trivial routine shown in listing 2.4.

Listing 2.4 Legendre symbol

```
int msqr(mpz_t x)
{
  return mpz_legendre(x, modulus); ← return Legendre symbol of input
}
```

2.4 Computing the square root mod n

In this section I go into the computation details of square roots over finite fields. The theory introduced here will also be used in chapter 12.

After checking if a number is a quadratic residue, we can compute the square root. We can use the last two bits of the modulus to determine how to take the square root. Since a modulus is odd, the last bit is always 1. The second to last bit can be 0 or 1, which means the modulus is congruent to 1 mod 4 or congruent to 3 mod 4 respectively.

If the modulus is congruent to 3 mod 4, we can use Fermat's little theorem (2.1) to power our way to the result. The formula is found in section 1.5 of (Cohen, 2000)

$$x = a^{(p+1)/4} \bmod p.$$

You can think about this as $p+1$ converts the last two bits from 1 to 0, and then the divide by 4 results in an exact power. Since $a^{p+1} \cong a^2 \bmod n$, when we divide by 4 we end up with $a^{1/2}$.

When the prime modulus p is congruent to 1 mod 4 life is a touch more complicated. The following description comes from section 1.5.1 of (Cohen, 2000), it is called "The Algorithm of Tonelli and Shanks".

The first step is to determine how many zeros there are after we subtract 1 from the prime. We know there are at least two zero bits because the prime is congruent to 1 mod 4. But there could be a lot more zero bits in the binary representation of p. As we scan past the zero bits, eventually we get to a set bit. That becomes the least significant bit of a number, which is labeled q.

Therefore, we take

$$p - 1 = 2^e \cdot q$$

where q is odd and e is at least 2. We then choose a number n at random which is a quadratic nonresidue. Since on average half the numbers modulo a prime p are in this group that does not take very long.

The initialization process then proceeds with the following setup:

$$\begin{aligned} y &\leftarrow n^q \bmod p \\ r &\leftarrow e \\ x &\leftarrow a^{(q-1)/2} \bmod p \\ b &\leftarrow ax^2 \bmod p \\ x &\leftarrow ax \bmod p \end{aligned} \tag{2.2}$$

Where a is the number we are attempting to find the square root of, x is going to be our result and b will go to 1 when we finish.

The loop process is: find the smallest m such that $b^{2^m} \cong 1 \bmod p$. If $m = r$, output a message that a is a nonresidue (which should not happen because we test for that to begin with).

We then perform the following operations mod p:

$$\begin{aligned} t &\leftarrow y^{2^{r-m-1}} \\ y &\leftarrow t^2 \\ r &\leftarrow m \\ x &\leftarrow xt \\ b &\leftarrow by \end{aligned}$$

and test to see if $b \cong 1 \bmod p$.

Exercise 2

Would taking square roots modulo 27213068317 use the power method or Tonelli and Shanks method? HINT: convert to hex and look at last two bits.

OK, now let's turn all this math into code. Listing 2.5 shows the first thing I do is check if we can take a square root. If not, just bail without doing anything else.

Listing 2.5 Square root mod n: entry

```
int msqrt(mpz_t x, mpz_t a)
{
  mpz_t n, q, z, y, b, t, t1;
  long e, i, r, cmp, m;

  if(!msqr(a))
    return 0;      | exit immediately if nonresidue
```

The second step is to check if this is an easy calculation. If the last two bits of the modulus are set, we can use the direct power operation to compute the square root. This uses the GMP `mpz_tstbit()` function as shown in listing 2.6.

Listing 2.6 Square root mod n: $p \cong 3 \bmod 4$

```
  if(mpz_tstbit(modulus, 0) && mpz_tstbit(modulus, 1))   | last 2 bits set?
  {
    mpz_init_set(q, modulus);
    mpz_add_ui(q, q, 1);
    mpz_divexact_ui(q, q, 4);      | compute a^((q+1)/4)
    mpz_powm(x, a, q, modulus);
    mpz_clear(q);
    return 1;
  }
```

If our modulus has bit 1 clear, we dive into the Tonelli and Shanks algorithm shown in listing 2.7. After subtracting 1 from the modulus, I use the `mpz_scan1()` function to count the number of zeros at the end of q. I then brute force divide by 2 for each clear bit to get q to be odd.

The next section of code looks for a quadratic nonresidue by starting with the assumption that the return value is 1. Once it finds a nonresidue, the value of `i` will be -1 and the loop will exit.

The initialization code follows the math step by step as outlined above. There is no `mpow()` routine that uses the modulus because the only call for it outside `modulo.c` has not yet set the modulus when it is called. There is an `mpowi()` routine used outside `modulo.c` to take advantage of the `mpz_powm_ui()` routine. This is shown in listing 2.8.

Listing 2.7 Square root mod n: Tonelli and Shanks

```
  mpz_inits(n, q, y, b, t, t1, NULL);

  mpz_sub_ui(q, modulus, 1);   <- q = p - 1
  e = mpz_scan1(q, 0);         | find number of binary zeros
  i = e;                       | break down p - 1 into 2^e*q
  while(i)
  {
    mpz_divexact_ui(q, q, 2);
```

```
    i--;                                   remove one factor of two at a time
  }

  i = 1;                                   find a generator
  while(i >= 0)                            randomly search for nonresidue
  {
    mrand(n);
    i = msqr(n);
  }

  mpz_powm(y, n, q, modulus);  ← y = n^q   initialize working components
  r = e;
  mpz_sub_ui(q, q, 1);
  mpz_divexact_ui(q, q, 2);                x = a^((q-1)/2)
  mpz_powm(x, a, q, modulus);
  mmul(b, x, x);                           b = ax^2
  mmul(b, b, a);
  mmul(x, x, a);               ← x = ax

  loop on algorithm until finished or failure

  cmp = mpz_cmp_ui(b, 1);
  while(cmp)   ← terminate when b==1
  {
    m = 1;
    mpz_set(t1, b);
    while(m < r)
    {
      mpowi(t1, t1, 2);
      if(!mpz_cmp_ui(t1, 1))               minimum m such that b^(2^m) ≅ 1 mod p
        break;
      m++;
    }
    if(r == m)                             should never happen because a is quadratic residue
    {
      mpz_clears(n, q, y, b, t, t1, NULL);
        return 0;
    }
    i = r - m - 1;
    mpz_set(t, y);
    while(i)                               t = y^(2^(r-m-1))
    {
      mpowi(t, t, 2);
      i--;
    }
    mmul(y, t, t);   ← y = t^2
    r = m;
    mmul(x, x, t);   ← x = xt
    mmul(b, b, y);       ← b = by
    cmp = mpz_cmp_ui(b, 1);   ←  is b == 1?
  }
  mpz_clears(n, q, y, b, t, t1, NULL);
```

```
  return 1;
}
```

The final block of code performs the loop section described before listing 2.5. The first step is to find the minimal `m` such that $b^{2^m} \cong 1 \bmod p$. This means we square a temporary variable `t1` until it goes to 1. If $r = m$ when we exit the loop, it means the input was a non-residue. This should never happen because we check on entry to the subroutine. If it does, there is a serious problem somewhere which must be debugged.

Once I have m, I set the value t to y and compute $t = y^{2^{r-m-1}}$ using a squaring loop. Then y is set to t^2, x is set to $x \times t$ and b is set to $b \times y$. Finally, the loop end test variable `cmp` is evaluated to determine if the routine is finished. When b equals 1 the variable `x` contains the square root of input `a` modulo the static prime `modulus`.

The final routine I want to show in the `modulo.c` file is the $b^i \bmod n$ routine in listing 2.8. This routine checks to see if the input power is negative. If it is, I first invert the input and then raise that to a positive power. If the input power is zero, the return value is simply 1. Otherwise, it just computes the direct result with the modulus.

Listing 2.8 Small integer power mod n

```
void mpowi(mpz_t a, mpz_t b, long i)
{
  if(i < 0)
  {                                          | negative power
    minv(a, b);                              | is inverse
    mpz_powm_ui(a, a, -i, modulus);          | to positive power
  }
  else if(!i)
    mpz_set_ui(a, 1);                        | b^0 = 1
  else
    mpz_powm_ui(a, b, i, modulus);           | a = b^i mod n
}
```

2.5 Summary

- A finite field allows addition, multiplication and inversion with every element to another element in the field.
- Elliptic curve groups are finite fields consisting of points. The order of each point is some combination of the factors from the total number of points on the curve.
- Subroutines to compute over prime fields include addition, subtraction, multiplication, inversion and division. These routines will be used throughout the book.
- A quadratic residue is an element in a finite field which is a perfect square.
- Computing square roots uses a power function for primes congruent to 3 mod 4 and Tonelli-Shanks algorithm for primes congruent to 1 mod 4.

Chapter Bibliography

Cohen, Henri. 2000. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag. 8

Koblitz. 1994. *A course in number theory and cryptography*. 2 edn. Springer. 7, 8

2.6 Answers to exercises

- 1) 86 = 2 x 43 so all possible group sizes are 2, 43, and 86.
- 2) 27213068317 decimal = 65606781D in hexadecimal. Since D = 1101 in binary, the last two bits are not both set. Therefore the Tonelli and Shanks method is required to take square roots modulo 27213068317.

3 Explaining the core of elliptic curve mathematics

This chapter covers

- Elliptic curve fundamentals
- Algebra using elliptic curves
- Code for adding and multiplying points
- Embedding data on a curve

This chapter begins the journey into the mathematics of elliptic curves over finite fields. I'll start with elliptic curves over the real numbers, so we can visualize what the mathematics does. Then we'll see how elliptic curves on the complex plane map to elliptic curves over finite fields with very crude graphics. Then we'll dive into the detailed mathematics of how we can perform addition using two points on an elliptic curve and finally expand that to the concept of multiplication.

3.1 Elliptic curve algebra

In this section I describe elliptic curves in a more visual way. It is useful to keep this visualization in mind when covering elliptic curves over finite fields because the formulas are the same. The plots over finite fields are not as pretty so there are fewer of those.

To start off I want to show the standard plot of an elliptic curve in the (x, y) plane. The equation $y^2 = x^3 - 5x + 5$ is pretty when plotted using real numbers. Figure 3.1 shows a

graph of this equation in blue.

What makes elliptic curves so useful is the ability to do algebraic manipulation of points the same way we manipulate numbers. To add two points P and Q we draw a line through them and find the place where the line intersects the elliptic curve. As seen in figure 3.1, the result is actually taken as the point on the opposite side of the curve.

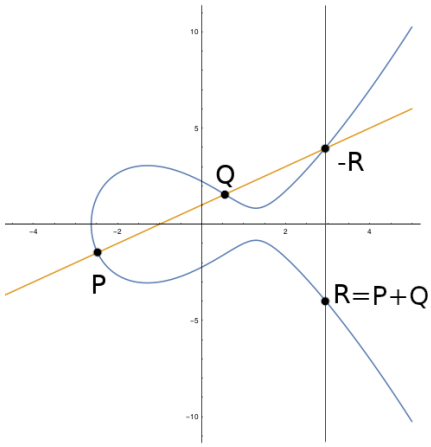

Figure 3.1 Elliptic Curve $y^2 = x^3 - 5x + 5$

Using the vertical line between R and $-R$ for addition of those two points should give us the identity for addition. That is $R - R = \mathbf{0}$. Clearly the point $\mathbf{0}$ is not on the graph. The identity element $\mathbf{0}$ is called the point at infinity. It is not on the curve but is essential for the points on an elliptic curve to be called a field. Different text books use different symbols. I will stick with $\mathbf{0}$ because the point at infinity acts like zero: any point plus zero is that point itself, any point minus itself is zero. The point at infinity is special and will be treated as a special case in the code.

3.1.1 Point representation

In this section I show how finite field elliptic curve points are stored in the computer.

There are a few ways to represent points on an elliptic curve which changes the formulas for point addition. These representations can reduce computation time. Chapter 1 of (Silverman, 2013) goes into these forms and chapter VI of (Blake *et al.*, 1999) goes into details of a representation I won't cover here. In this book a simple set of two values (x, y) will represent a point.

All the curves we are interested in are called ordinary. They are defined with the equation:

$$y^2 = x^3 + a_4x + a_6 \tag{3.1}$$

In chapter 2 I gave the complete equation for an elliptic curve. For ordinary curves over finite fields we only need a_4 and a_6, the other coefficients in the general curve are 0.

Looking at figure 3.1 we can see that for every x value on the curve, there are two y values. Since a square root can have either a positive or negative result, that makes sense. The same is true over a finite field. The negative of a point from equation 3.1 always follows the rule:

$$-(x, y) = (x, -y)$$

For all the curves we are interested in, coefficient a_6 is never zero. This is exceptionally fortunate because y can never be 0. This gives us a way to represent $\mathbf{0}$ in the computer. Undefined is not "mathematically" correct, but for our purposes it works really well:

$$\mathbf{0} = (0, 0)$$

that is, the point at infinity has $x = 0$ and $y = 0$ which is very easy to test for.

3.1.2 Elliptic curves over Finite fields

This section goes into some esoteric math connecting finite fields and the complex plane. The idea is to show the depth and beauty of the mathematics. I also want to give a mental picture which is useful with elliptic curves covered in part 3 (field extension curves).

One of the most fascinating aspects of elliptic curves is that we can map curves over finite fields to curves over the complex plane. This is both really cool and very important to keep in the back of your mind when we get into part 3 of the book.

In the process of doing a line integral over an elliptic curve on the complex plane, the complex plane is wrapped into a torus (to understand why look at section VI.1 in (Silverman, 2013)). Figure 3.2 is a very crude attempt to get the idea across without diving into too much math. The elliptic curve function repeats along the drawn vectors. Since the torus is a surface, there are two independent vectors that mark out a range where the values do not repeat.

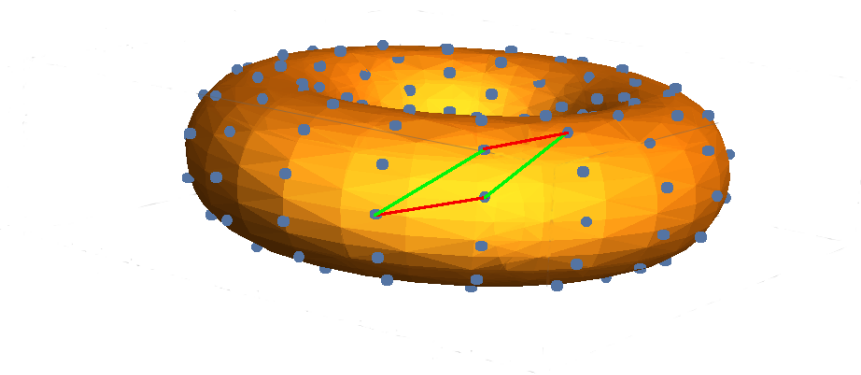

Figure 3.2 Elliptic curve over the complex plane

Figure 3.3 zooms in on the rectangle from figure 3.2. The points form a grid which repeats over the torus. The distance between the points perfectly divides the two vectors,

so there are an integer number of points on the torus. This grid exactly maps to a finite field.

The fundamental importance here are the two cyclic vectors. The red lines represent one cyclic group and the green lines a second cyclic group. In the first part of this book we will look at curves with just a single cycle. Part 3 of the book deals with curves composed of two cyclic groups as this is what makes pairing of points possible.

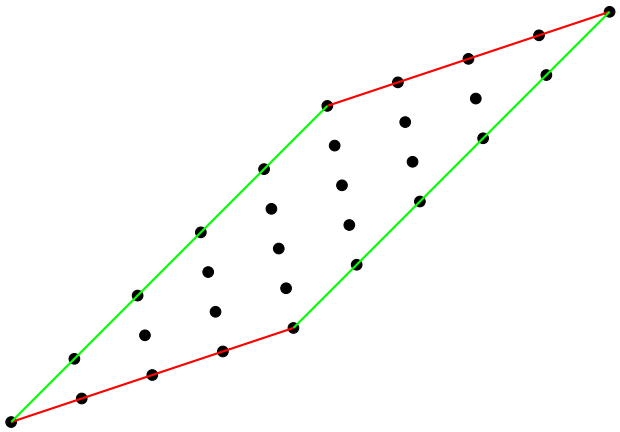

Figure 3.3 Points on curve over complex plane

Since a finite field is cyclic, it repeats similarly. This ability to map finite field math to complex plane math allows all the rules of calculus to be applied to the elliptic curves over finite fields. The rules for point addition are thus the same for real, complex and finite fields. We are interested in finite fields based on large primes, not fields based on powers of 2 or 3. On very tiny processors with limited capabilities, finite fields over powers of 2 are exceptionally useful so check out reference (Rosing, 1999) for those details.

3.1.3 Point addition

In this section I cover the algebra of adding two points on an elliptic curve to get a third point on the same curve.

The first step in adding points is finding the slope of the line between them. While we can come up with formulas for different points, we also need a formula for adding the same point to itself. That combination is a tangent to the curve as shown in figure 3.4.

Since the slope of a tangent is computed differently than the slope between two points we normally have two different formulas. Unfortunately this allows a side channel attack using power analysis which can help an adversary determine the secret key. To avoid this problem I will use the formula at the end of section III.3 in (Silverman, 2013) to compute the slope. This works for both tangents and different points equally. At best an attacker can only learn the number of bits set (called Hamming weight in the literature) in the private key.

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. It is OK for P_2 to equal P_1. Both points are on the

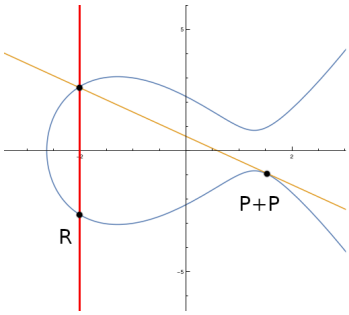

Figure 3.4 Adding point to itself

curve 3.1, so they satisfy that equation. We take the slope λ between the two points to be

$$\lambda = \frac{x_1^2 + x_1 x_2 + x_2^2 + a_4}{y_1 + y_2} \tag{3.2}$$

The resulting point $R = P_1 + P_2 = (x_3, y_3)$ is then found using the formulas

$$x_3 = \lambda^2 - x_1 - x_2 \tag{3.3}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \tag{3.4}$$

For the ordinary equation 3.1 the equations 3.2, 3.3 and 3.4 are the point addition formulas over an elliptic curve.

There is a problem with this only when $y_1 + y_2 = 0$. This special case is rare and will be discussed in detail with the code implementation later.

Exercise 1

Why is $y_1 + y_2 = 0$ a problem in equations 3.3 and 3.4?

3.1.4 Point multiplication

In this section I show how adding a point to itself multiple times is called multiplication.

The next step in the mathematics of using elliptic curves is multiplication. We do this by adding a point to itself multiple times. For a point P and integer k we write

$$Q = kP = \underbrace{P + P + \cdots + P}_{k}$$

with P added to itself k times.

Rather than actually perform this operation, we use the double and add formula. There are many ways to speed this up as shown in section IV.2 of reference (Blake *et al.*, 1999). I paraphrase the method in figure 3.5.

The idea behind double and add is to expand k from most significant bit downward. Multiplying by two for every bit position is similar to a shift left, and then if the next bit is set we add in the original point. This is a simple walk down every bit position. A

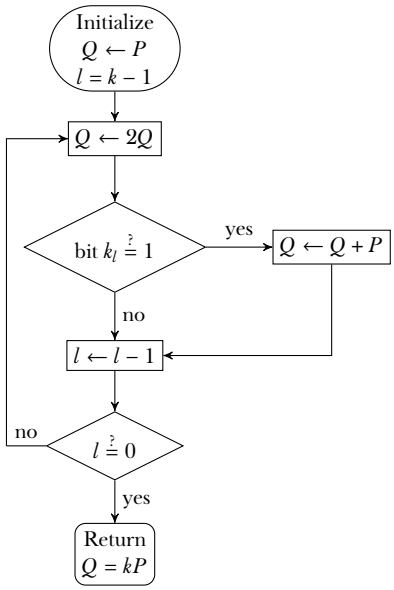

Figure 3.5 Double and Add Method

side channel attack looks at power consumption to tell the difference between the double and the add routine. So while it is a touch slower, using formula 3.2 is more secure on embedded systems.

The points on an elliptic curve over a finite field are finite in number. When a point is added to itself enough times you get back to where you started. This gives us a cyclic group to work with. Along the way you will also hit the point at infinity. The number of times it takes to add a point to itself that gets to the point at infinity is called the order of the point.

As described in section 2.1 the points on elliptic curves form cyclic groups based on the combination of factors making up the number of points on the curve. We can determine what the order of a point is by multiplying each combination of those factors with the point to see if we hit the point at infinity.

Using the example from section 2.1.1 we have two points belonging to a group with factor $a = 2$ and five points with factor $b = 5$. All the points on the curve which have order c are called the c-torsion subgroup. For security, we want c to be a very large prime. There will also be points of order $2 \cdot c$ and $10 \cdot c$, but they are not cryptographically useful because their order is not a prime.

Exercise with answer

Show how the double and add method is similar to elementary school multiplication in binary using values 7 times 5.

7 = 111b and 5 = 101b. Using $k = 5$, start with answer = 111b. Double this to get 1110b. The second bit in 5 is clear, so no further addtion is perfomed. Double again to get answer = 11100b. The last bit in 5 is set, so add in 111b to the answer which gives 10011b = 35.

3.1.5 Embedding data on a curve

In this section I show how arbitrary data can be shifted to allow a value to become a point on an elliptic curve.

Up to this point all descriptions have assumed we have (x, y) pairs which satisfy equation 3.1. To choose a random point on the curve we can start with a random x value. The odds that value of x satisfies equation 3.1 is less than 50%. A look at figure 3.1 shows half the plane has no points, so this is to be expected.

Elliptic curves over finite fields do not plot very well. What we see are individual points. Figure 3.6 shows an example curve used throughout this book. It is the curve $y^2 = x^3 + 23x - 1 \bmod 43$. It is easy to see some places have gaps along the x axis. Those x values do not satisfy the curve equation.

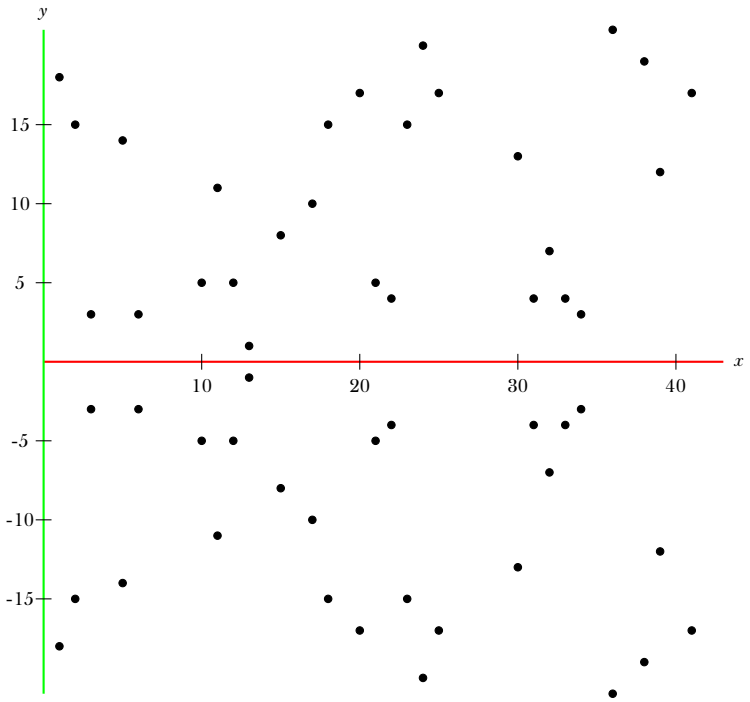

Figure 3.6 All points on tiny elliptic curve $y^2 = x^3 + 23x - 1$ mod 43

Figure 3.7 shows the process for embedding arbitrary data onto an elliptic curve. After computing the right-hand side of equation 3.1, the test in the diamond is the Legendre symbol from chapter 2 referenced to the field prime. If the value of $f(x)$ does not allow a square root, x is incremented by 1. This repeats until a value for x is found that is on the curve. The value of y is found from the square root of $f(x)$. More details will be developed with the code.

As an example, there is a gap on the right hand side of figure 3.6 between 25 and 30. Suppose our random number generator picks an x value of 27. The routine computes $27^3 + 23 * 27 - 1 \bmod 43 \ (= 7)$, then checks if $\left(\frac{7}{43}\right) = 1$. It does not, so the routine computes $28^3 + 23 * 28 - 1 \bmod 43 \ (= 20)$, which also fails the Legendre symbol test and the same happens for $x = 29$. When $x = 30$, the Legendre symbol test does equal 1 and the y values turn out to be 4 and −4.

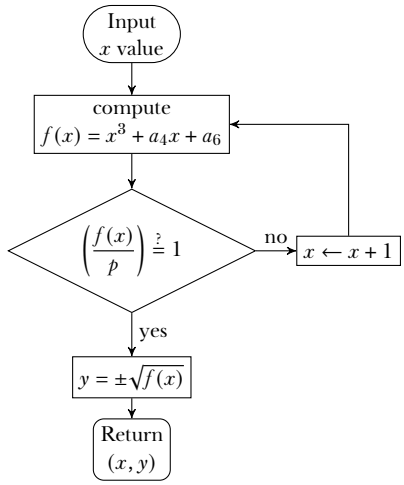

Figure 3.7 Embedding data on curve

While not every x value can be on the curve, there will be a nearby x value that is.

As an aside, note that the plot of figure 3.6 could have used all positive values of y. Instead of being mirrored around 0, it would be mirrored around 21. All negative values $-k$ can be replaced by adding the modulus p to get the equivalent value $p + k$. All the routines in this book will return positive values when a modulus operation is the last step, but might return a negative value when subtraction is the last step.

Exercise 2

Will all values of x such that $\left(\frac{f(x)}{p}\right) = 1$ generate a point on an elliptic curve?

3.2 Elliptic curve subroutines

In the following sections I show how finite field elliptic curve mathematics is implemented in code.

Theory is great because it gives us understanding of what we need to do. Reality requires a lot more detail. So to turn all that math into code I start with structures for curves and points and ways to manipulate those structures. Then I dive into the details of point addition and then use the point addition routine to cover point multiplication. Code for

embedding data on a curve will be described as well along with a few miscellaneous routines for creating random points and help with debugging.

3.2.1 Code to represent curves and points

In this section I define structures which are used in the rest of the book to help manipulate points and curves.

Since both curves and points are used as objects, I start with a set of structures which are defined in a header. The file is `eliptic.h` and it is obvious I can't spell. The structures are shown in listing 3.1.

Listing 3.1 Point and curve structures

```
typedef struct
{
    mpz_t   x;
    mpz_t   y;
}POINT;

typedef struct
{
    mpz_t   a4;
    mpz_t   a6;
}CURVE;
```

point is (x, y) value

curve is a_4, a_6 coefficients

Since these are large integers in the GMP library each component needs to be initialized. For some subroutines we have temporary points, so these variables need to be cleared (`free()` in `malloc()` terms). These routines are shown in listing 3.2.

Listing 3.2 Point and curve initialization

```
void point_init(POINT *P)
{
    mpz_inits(P->x, P->y, NULL);
}

void point_clear(POINT *P)
{
    mpz_clears(P->x, P->y, NULL);
}

void curve_init(CURVE *E)
{
    mpz_inits(E->a4, E->a6, NULL);
}

void curve_clear(CURVE *E)
{
    mpz_clears(E->a4, E->a6, NULL);
}
```

← **initialize point structure**

← **clear point structure**

← **initialize curve structure**

← **clear curve structure**

Two simple routines used on points are one: copy a point from variable A to variable B, and two: test if a point is the point at infinity. These routines are shown in listing 3.3.

Listing 3.3 Point copy and test

```
void point_copy(POINT *R, POINT P)
{
    mpz_set(R->x, P.x);
    mpz_set(R->y, P.y);
}

int test_point(POINT P)
{
    if(!mpz_cmp_ui(P.x, 0) && !mpz_cmp_ui(P.y, 0))
      return 1;
    return 0;
}
```

copy both x and y values

both x==0 and y==0 for point at infinity

3.2.2 Code for point addition

In this section I describe the code which implements point addition for prime field elliptic curves.

Adding two points is a major subroutine using formulas 3.2, 3.3 and 3.4. Since any point added to **0** is the point itself, I first check to see if any of the input points are the point at infinity. This is shown in listing 3.4.

Listing 3.4 Point addition: test for 0

```
void elptic_sum(POINT *R, POINT P, POINT Q, CURVE E)
{
  mpz_t lmbda, ltp, lbt, t1, t2, t3;
  POINT rslt;

  if(test_point(P))
  {
    point_copy(R, Q);
    return;
  }
  if(test_point(Q))
  {
    point_copy(R, P);
    return;
  }
```

if either point at infinity return other point

After we determine that the points are not **0** we compute the slope using formula 3.2. This is shown in listing 3.5. As seen in figure 3.1 points R and $-R$ have the same x value and opposite y values. So if we add two y values we get 0 and the formula 3.2 would then divide by 0.

Listing 3.5 Point addition: computing λ

```
mpz_inits(t1, t2, t3, ltp, lbt, lmbda, NULL);
mmul(t1, P.x, P.x);
mmul(t2, P.x, Q.x);
mmul(t3, Q.x, Q.x);
madd(ltp, t1, t2);
madd(ltp, ltp, t3);
madd(ltp, ltp, E.a4);
madd(lbt, P.y, Q.y);
if(!mpz_cmp_ui(lbt, 0))
{
  msub(lbt, Q.x, P.x);
  if(!mpz_cmp_ui(lbt, 0))
  {
    mpz_set_ui(R->x, 0);
    mpz_set_ui(R->y, 0);
    mpz_clears(t1, t2, t3, ltp, lbt, lmbda, NULL);
    return;
  }
  msub(ltp, Q.y, P.y);
}
mdiv(lmbda, ltp, lbt);
```

top = $x_1^2 + x_1x_2 + x_2^2 + a_4$

compute lambda using general form

← bottom = $y_1 + y_2$

← enter if $y_1 + y_2$==0

← compute $x_2 - x_1$

← enter if $x_2 - x_1$==0

x_2==x_1 results in point at infinity

special case $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$

either case $\lambda = \frac{\text{top}}{\text{bottom}}$

If we draw a horizontal line through point Q in figure 3.1 we would have three y values that are the same. That means there are three matching values on the negative half of the curve. While rare, it is possible that we have y value sums that cancel, but they are not at the same x value. Under this condition we must compute the slope the old-fashioned way using

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

If $x_2 = x_1$ then this formula also goes to 0. The purpose of the two `if()` statements is to check

- $y_1 \stackrel{?}{=} -y_2$
- $x_1 \stackrel{?}{=} x_2$.

The result is the point at infinity **0** if both conditions are true, otherwise we can compute the third point.

Listing 3.6 is the calculation of formulas 3.3 and 3.4. This is straightforward use of the modular subroutines shown in chapter 2. Because we don't want to clobber old points while computing the new one (again, learned the hard way!) the temporary point `rslt` is used, so the final result can be copied back to the desired location.

Listing 3.6 Point addition: x_3 and y_3 calculation

```
  finally compute
  resulting point
    point_init(&rslt);
    mmul(t1, lmbda, lmbda);
    madd(t2, P.x, Q.x);          | x3 = λ^2 − (x1 + x2)
    msub(rslt.x, t1, t2);        |
    msub(t1, P.x, rslt.x);
    mmul(t2, t1, lmbda);         | y3 = (x1 − x3)λ − y1
    msub(rslt.y, t2, P.y);       |
    point_copy(R, rslt);  ← transfer result to output
    mpz_clears(t1, t2, t3, ltp, lbt, lmbda, NULL);
    point_clear(&rslt);
}
```

3.2.3 Code for point multiplication

Multiplying points by a large number is the core of elliptic curve cryptography. The code to add two points is a bit complicated. But once we have that subroutine the jump to point multiplication via figure 3.5 is easy. The point multiplication routine is shown in listing 3.7.

Listing 3.7 Point multiplication

```
void elptic_mul(POINT *Q, POINT P, mpz_t k, CURVE E)
{
  int bit, j;
  POINT R;

  point_init(&R);
  point_copy(&R, P);          | save input point
  j = mpz_sizeinbase(k, 2) - 2; ← bit position index
  while(j >= 0)
  {
    elptic_sum(&R, R, R, E); ← double for each bit position
    bit = mpz_tstbit(k, j);
    if(bit)
      elptic_sum(&R, R, P, E); ← add for each bit set
    j--;
  }
  point_copy(Q, R); ← transfer result
  point_clear(&R);
}
```

The GMP routines `mpz_sizeinbase()` and `mpz_tstbit()` are used to determine how many bits we need to work with and if a bit is set or not. The `sizeinbase` routine is used here to count bits. We subtract 2 from the return value because the starting offset is 1 and we skip the most significant bit. Every time through the loop the result R is doubled, and if the bit k_j is set, the original point is added to the result.

3.3 Code for embedding data on a curve

In this section I describe the detailed method of converting arbitrary data into a point on an elliptic curve.

Suppose I want to use elliptic curves to transmit a short message. The first step is to convert the message into a point (x, y). Unfortunately, not all x values have a point on the curve. We need to adjust the message to find some x' which is close to the message we want to send.

It turns out the odds of not finding an x' close to x go as $\frac{1}{2^{2^{n-1}}}$ where n is the number of bits modified at the end of the message. There is a 50% chance with one bit. With eight bits, the odds of not finding an x' on the curve are astronomically small (approximately 3×10^{-39}), and most likely only five or six bits are needed.

The right-hand side of equation 3.1 is entirely a function of x. This makes a nice subroutine if we take

$$f(x) = x^3 + a_4x + a_6$$

Listing 3.8 shows this calculation given an input value x and elliptic curve E.

Listing 3.8 Embedding data: computing $f(x)$

```
void fofx(mpz_t f, mpz_t x, CURVE E)
{
  mpz_t t1, t2;

  mpz_inits(t1, t2, NULL);
  mmul(t1, x, x);
  mmul(t1, t1, x);          | t1 = x^3
  mmul(t2, E.a4, x);
  madd(f, t1, t2);          | f = x^3 + a4x + a6
  madd(f, f, E.a6);
  mpz_clears(t1, t2, NULL);
}
```

To determine if the value of x can be found on the curve, $f(x)$ must have a square root because $y^2 = f(x)$. If $f(x)$ is not a quadratic residue, $f(x)$ does not have a square root. As described in section 3.1.5, we can increment x and look to find an x value that is on the curve as shown in the algorithm of figure 3.7. Usually this takes 2 to 5 tries, but sometimes it can take over 30. This is where the 5 or 6 bits of noise in x' are useful. I typically use the last byte in an x value as spare when embedding specific data on a curve.

Listing 3.9 shows a way to embed data on a curve. To allow addition by 1 modulo the field prime I create the constant `one` as an `mpz_t` integer. I copy the input value to the output x value of the point and then check if $f(x)$ is a quadratic residue. If it is, the value of x is used, if it is not a quadratic residue, then the point x value is incremented by 1 and the testing continues until a quadratic residue is found.

Because the square root has a positive and negative value, both points are returned. Since it is arbitrary, the first point is set to the smaller y value. With numbers modulo a prime, we can always take a negative value and add the modulus to get a positive result

which is congruent to the original negative number. The GMP manual says the result of `mpz_mod()` is always non-negative. For consistency sake, I chose to put the point with the smaller y value first.

Listing 3.9 Embedding data: finding points

```
void elptic_embed(POINT *P1, POINT *P2, mpz_t x, CURVE E)
{
  mpz_t f, one;
  int done;

  mpz_init(f);
  mpz_init_set_ui(one, 1);   ← mpz constant 1
  mpz_set(P1->x, x);
  done = 0;
  while(!done)
  {
    fofx(f, P1->x, E);
    if(msqr(f) > 0)          | f(x) is quadratic residue
      done = 1;
    else
      madd(P1->x, P1->x, one); ← increment x by 1
  }
  mpz_set(P2->x, P1->x);
  msqrt(P1->y, f);           |
  mneg(P2->y, P1->y);        | two y values
  done = mpz_cmp(P2->y, P1->y);
  if(done < 0)               |
    mpz_swap(P2->y, P1->y);  | smaller y value first
  mpz_clears(f, one, NULL);
}
```

3.4 Miscellaneous routines

This section includes two routines which are useful in the rest of the book but do not fit in a mathematical description.

There are two more routines in the `elliptic.c` file. One creates random points and the other prints points to the console for debugging. The random point routine is shown in listing 3.10. This first finds a random value in the range of the modulus. It then checks the last bit of this random number and embeds the smaller y value to the point if the bit is clear and the larger y value if the bit is set.

Listing 3.10 Random point

```
void point_rand(POINT *P, CURVE E)
{
  mpz_t r;
  POINT mP;
```

```
  mpz_init(r);
  mrand(r);       ← random value
  point_init(&mP);
  if(mpz_tstbit(r, 0))
    elptic_embed(P, &mP, r, E); ← last bit set return smaller y
  else
    elptic_embed(&mP, P, r, E); ← last bit clear return larger y
  mpz_clear(r);
  point_clear(&mP);
}
```

When debugging it is very useful to know what intermediate values are. The point printing routine shown in listing 3.11 requires a string to label the point and the point itself.

Listing 3.11 Print a point

```
void point_printf(char *str, POINT P)
{
  printf("%s", str);
  gmp_printf("(%Zd,  %Zd)\n", P.x, P.y);
}
```

The header file `elliptic.h` includes the structures in listing 3.1 as well as the function definitions described in this chapter. Listing 3.12 is a nice summary of all the code presented in this chapter.

Listing 3.12 Header function definitions

```
void point_init(POINT *P);
void point_clear(POINT *P);
void curve_init(CURVE *E);
void curve_clear(CURVE *E);
void point_copy(POINT *R, POINT P);
int test_point(POINT P);
void fofx(mpz_t f, mpz_t x, CURVE E);
void elptic_sum(POINT *R, POINT P, POINT Q, CURVE E);
void elptic_embed(POINT *P1, POINT *P2, mpz_t x, CURVE E);
void point_printf(char *str, POINT P);
void elptic_mul(POINT *Q, POINT P, mpz_t k, CURVE E);
void point_rand(POINT *P, CURVE E);
```

3.5 Summary

- The fundamental formula for ordinary elliptic curves is

$$y^2 = x^3 + a_4x + a_6.$$

- The sum of two points on an elliptic curve is the negative point of the intersection

point to the curve of a line drawn between them.

- The point at infinity is not on the curve. It is the identity element and for ordinary curves can be represented as (0, 0) in code.
- Elliptic curves over finite fields have a 1 to 1 mapping to curves over the complex plane.
- The point addition formula has a special form for use in secure applications. Using a standard form can leak key information when side channel attacks are applied.
- Point multiplication uses a double and add algorithm to rapidly compute extremely large prime multipliers.
- When the point mP is the point at infinity $\mathbf{0}$ then P is order m and is a member of the m-torsion subgroup.
- For coding, curves and points have simple structures which include (a_4, a_6) and (x, y) respectively.
- The point addition routine must check for the point at infinity on input and output.
- Point multiplication code uses the same addition routine for doubling and adding to enhance security.
- To embed data on a curve the x value is incremented until $x^3 + a_4x + a_6$ is a quadratic residue. When sending a message on a curve point the last six to eight bits should be considered noise.
- Random points are found by embedding random x values on the curve. This will be useful for many routines described throughout the book.

Chapter Bibliography

Blake, I., Seroussi, G., & Smart, N. 1999. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press. 15, 18

Rosing, M. 1999. *Implementing Elliptic Curve Cryptography*. Manning Pubs Co Series. Manning. 17

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York. 15, 16, 17

/

3.6 Answers to exercises

- 1) The variable λ is divided by $y_1 + y_2$. When $y_1 + y_2 = 0$, $\lambda = \infty$. Equations 3.3 and 3.4 are then useless.
- 2) Yes. In fact it will have two matching points for $\pm y$.

4 Key exchange using elliptic curves

This chapter covers

- Create a shared secret using private and public keys.
- The Diffie-Hellman key exchange using elliptic curves.
- Implementing the NIST Full ECC MQV algorithm.

In this chapter I describe two methods of secure key exchange. As described chapter 1, elliptic curve cryptography is used to create a secret key for an efficient encryption algorithm. No one but the two parties exchanging public keys can compute the secret key.

Now that we have the basic elliptic curve mathematics routines for point addition and multiplication, we can begin to look at algorithms which use these techniques to implement public key cryptography. The private key is a large integer and the public key is a point. Since the private key can be anything, a hash of a pass phrase which is never stored can be really secure. At the system level this might be a problem if the phrase is forgotten, but there is nothing to be done about that here.

All key exchange algorithms are based on the Diffie-Hellman process. The process involves the sender's public key and the receiver's private key. For many peer-to-peer transactions that do not happen very often this is sufficient. For common transactions between

two users (like an employee to their company) it might allow an attack on the shared secret. To prevent this a more sophisticated method called the Menezes-Qu-Vanstone (MQV) key exchange algorithm can be used.

The following sections will describe the Diffie-Hellman key exchange algorithm followed by the MQV key exchange algorithm. The MQV algorithm uses ephemeral keys which change every time two parties communicate in addition to the static public keys used in the Diffie-Hellman key exchange algorithm.

The use case for one over the other depends on your environment. If two systems communicate on a regular basis then choosing MQV makes sense. If two systems will only rarely communicate (say a customer registering a product) then using Diffie-Hellman makes sense.

4.1 Diffie-Hellman algorithm description

In this section I describe the simplest algorithm for key exchange using elliptic curves.

As pointed out in chapter 3, we can embed a message on a curve if the message size is less than the modulus used to hold an x value. If our modulus is 256 bits, our message size can be 250 bits or at most 31 bytes. This is not very useful for most communications.

If instead we use the x values to create a secret key for a symmetric encryption algorithm such as Advanced Encryption Standard (NIST, 2001) (AES) our message can be as long as we want. This is the idea behind public key cryptography: two people send their public keys to each other and create a secret no one else can find. Figure 4.1 is a copy of figure 1.3 which shows how the Diffie-Hellman method works. Each person multiplies their private key with the other person's public key to create a shared secret. For a more in depth description of Diffie-Hellman key exchange look at https://livebook.manning.com/book/real-world-cryptography/chapter-5/point-16981-1-224-1.

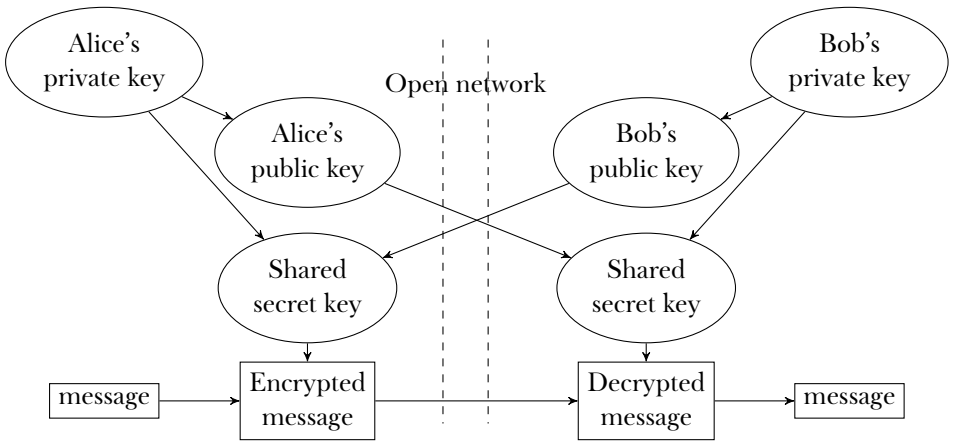

Figure 4.1 Creating shared secret key by exchanging public keys

4.1.1 Elliptic curve math

In this section the details of elliptic curve Diffie-Hellman key exchange are described.

We start a key exchange with a secure elliptic curve as described in chapter 3 – a curve with a large prime factor in the cardinality. Since we are using ordinary curves over finite fields it is possible to find curves which have a prime number of points as the finite number of points on the curve. In most books and papers the number of points on the curve is called the cardinality of the curve and is written mathematically as $\#E$ where the curve E is the equation 3.1 repeated here

$$E: \ y^2 = x^3 + a_4x + a_6.$$

Most curves over finite fields have a cardinality with many factors, but as we will show in chapter 6 it does not take too long with modern computers to find good secure curves which have prime cardinality with no cofactors. An adversary has no choice but to hunt over every point on the curve, there are no possible shortcuts.

For the rest of this book, capital variables will refer to points, and lower case variables will refer to a field value. The point G has values (x, y) which satisfy the equation of the curve that the point G is on. This saves a lot of writing. It means we operate at a higher level of abstraction. So the multiplication of a point by a value to get a new point must grind through all the equations of chapter 3.

Once we have a curve, we pick a base point G. The G stands for generator. Since we are interested in curves with prime factor cardinality, every point on the curve can be a generator because the order of every point is the same prime number. When we get into field extension curves in chapter 13 prime factor cardinality is not possible, so we need to find curves with a very large prime as one of the factors in the cardinality. We call the remaining small factors a cofactor. For useful curves I assume the size of the cofactor fits in a `long` which implies it is very small compared with the large prime factor.

For simple key exchange a secure curve over a finite field has prime order and once chosen is a public parameter. We make the base point G public as well, so everyone can use it. This public data is usually built into the program used for key exchange because it is required to create new keys.

To prepare for a key exchange, each person creates a private key. This is usually a hash of some pass phrase, but it can be any set of bits turned into an integer. We'll call Alice's private key k_A and Bob's private key k_B.

Alice creates a public key by computing

$$A = k_AG$$

and Bob creates a public key

$$B = k_BG$$

To communicate Alice and Bob send each other their public keys.

The security comes from the inability of using the knowledge of points G and A to find the value of k_A. This is called the elliptic curve discrete log problem (ECDLP). Since the

ability to solve this problem goes as the $\sqrt{\#E}$ we need twice as many bits in our prime factor as the level of security we are attempting to reach. That is, 128-bit AES level of security requires 256 bits of $\#E$ cardinality.

Once Alice and Bob have exchanged keys, they create a shared secret by multiplying the received public key with their own private key. This gives

$$S = k_A B = k_B A = k_A k_B G \tag{4.1}$$

They each use the x component of point S (or some chunk of bits from it) as the secret key for a symmetric algorithm such as AES.

Exercise 4.1

Clare wants to securely message Alice and Bob. Can she create one secret key between all three of them?

4.1.2 Hash function

In this section the important concept of hashing is described. This will be used in all example programs including the routines in chapters 18 and 19.

A hash function is an algorithm that takes an arbitrary length of bytes and smooshes out a fixed length of random looking bytes. A secure hash function changes half its output bits if one single input bit changes. There are many secure hash functions available, so I chose one whose core was approved by National Institute of Standards and Technology (NIST) and has since been improved.

The chosen hash software is called KangarooTwelve and is available from https://github.com/XKCP/K12/archive/refs/heads/master.zip or use Git to download us-ing https://github.com/XKCP/K12.git. While there are many subroutines in this package I only use one of them. If you want to know more details you can check out the inven-tor's home page: https://keccak.team/kangarootwelve.html. Once downloaded the source is straightforward to build into a library. I've added instructions for this in appendix A.

The routine has two input strings and one output string. The lengths of each are a separate argument. While the internal core function is a fixed size, the routine I call creates an extendable output. (NIST, 2016) defines an extendable output function as a function on bit strings in which the output can be extended to any desired length.

The reason we want extendable output is described in a document written by the In-ternet Engineering Task Force (IETF): "Hashing to Elliptic Curves". This can be found at URL
https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-11.

That document describes how a problem occurs when we convert a hashed output down to a prime value using a modulus n. Since the modulus is less than a full power of two, the odds are high our hashed output modulo n will be in a small range. To avoid this problem IETF advises we add the number of bits we require for security to the output length.

Listing 4.1 takes an arbitrary length input string and outputs a value modulo an input prime which should be the order of the base point. This routine has a fixed "customize"

string used in KangarooTwelve. The IETF standard requires a domain separation tag (dst) which allows the use of the same hash function in different routines. By changing the dst the same input string will output a unique result.

Listing 4.1 Hash to finite field

```
void hash(mpz_t hsh, unsigned char *dat, long len, mpz_t prm)
{
  unsigned char *outp, *dst;
  int m, k, b;

  dst = (char*)malloc(24);
  sprintf(dst, "Hash_b pring&sig");          setup domain separation tag

  m = mpz_sizeinbase(prm, 2);
  if(m < 208)
    k = 80;
  else if(m < 320)
    k = 128;                                 determine security level
  else if(m < 448)                           add appropriate
    k = 192;                                 extra bit length
  else
    k = 256;
  b = m + k;
  if(b & 7)
    b = (b >> 3) + 1;   <- round to next byte
  else
    b >>= 3;
  outp = (unsigned char*)malloc(b + 2);
  KangarooTwelve(dat, len, outp, b, dst, 16);   generate hash of input

  mpz_import(hsh, b, -1, 1, 0, 0, outp); <- convert bits to integer
  mpz_mod(hsh, hsh, prm);     <- force to be mod n
  free(outp);
  free(dst);
}
```

The output length is first computed in bits. If any of the last three bits are set the byte count is one more than a division by eight. The hash is then converted to an integer using the GMP `mpz_import()` function. The argument `-1` means little endian input, the next `1` signifies `b` is in bytes and the two zeros mean processor endianness output and no skipped bytes respectively. Using `mpz_mod()` with p the result is then the correct size.

4.1.3 Key generation

In this section I show how an individual's keys can be created. The advantage of elliptic curve key exchange is that keys do not need to be stored, they can be computed every time they are needed.

Once we have a hash function which converts a pass phrase to an integer mod p, generating the public key is easy. Assuming we use the random point function (listing 3.10)

to create the base point, a nice structure to hold the public parameters is shown in listing 4.2.

Listing 4.2 Base key structure

```
typedef struct
{
  long  cofactor; ← assume tiny cofactor
  mpz_t order;    | order of base point
  POINT Base;     | base point (x, y) value
  CURVE E;        | curve (a4, a6) coefficients
}BASE_SYSTEM;
```

Note that this includes a cofactor. For our example curves the cofactor will usually be 1, but for generality sake it needs to be included. I call this structure the `BASE_SYSTEM` because the curve and base point are field prime size. When we get to field extensions points and curves will become polynomials with field prime coefficients.

Listing 4.3 shows that we don't need to do much to create a public key from a pass phrase. Tracking the private key is system dependent. The security of the entire system rests on the private key staying secret. That is one advantage of going from a pass phrase directly to a key, the private key never has to be placed in non-volatile storage.

Listing 4.3 Key generation

```
void gen_key(mpz_t sk, POINT *PK, unsigned char *phrase, BASE_SYSTEM bse)
{
  long np;

  np = 0;                    | could have used
  while(phrase[np]) np++;    | np = strlen(phrase)
  hash(sk, phrase, np, bse.order); ←—— private key modulo
                                       base point order
  elptic_mul(PK, bse.Base, sk, bse.E);
}
```

4.1.4 Computing shared keys

In this section I show the code for computing a secret shared key.

In this section I'm going to assume that both Alice and Bob have traded their public keys over an open network. Once that has been done listing 4.4 shows how they both generate the same shared secret key. The output is the shared secret and the inputs are their own private key and the other person's public key. The curve that the points are defined to be on is also an input, so the multiplication can take place.

Listing 4.4 Diffie-Hellman shared key

```
void diffie_hellman(mpz_t keyshare, mpz_t my_key, POINT Their_key, CURVE E)
{
  POINT Tmp;
```

```
  point_init(&Tmp);
  elptic_mul(&Tmp, Their_key, my_key, E); ← compute (their public key)* (my private key)
  mpz_set(keyshare, Tmp.x);
  point_clear(&Tmp);
}
```

Of course, they each have the other person's public key and their own private key. As shown in equation 4.1 they both compute the same shared secret.

4.2 MQV algorithm

In this section a more involved key exchange algorithm is described.

In 1995 Menezes, Qu, and Vanstone proposed a key agreement scheme that provided perfect forward security (Menezes *et al.*, 1995). That means even if you figured out the shared secret key for that message, you have no way of finding information about any other message. Over the past 25 years the MQV algorithm has been studied and modified.

Today, NIST has a published a document on all the various ways to implement MQV. You can find it here (NIST, 2018). I will go over one of those methods called Full ECC MQV.

What I really like about this method is the use of ephemeral public and private keys in addition to static public and private keys. Ephemeral keys are generated once per key agreement session. So while static public keys may be stored, the ephemeral keys must be transmitted by both sides before a shared secret can be computed.

4.2.1 Elliptic curve math for MQV algorithm

This section goes into the details of the MQV method as described by NIST.

For notation I'll use subscript $_e$ for ephemeral and subscript $_s$ for static. As with Diffie-Helmann, the base point is G on the curve E. The order of the base point n and cofactor of the curve r are known. Alice has a static private key $k_{s,A}$ and ephemeral private key $k_{e,A}$. Similarly, Bob has static and ephemeral private keys $k_{s,B}$ and $k_{e,B}$ respectively. The public keys are

$$P_{s,A} = k_{s,A}G$$
$$P_{e,A} = k_{e,A}G$$
$$P_{s,B} = k_{s,B}G$$
$$P_{e,B} = k_{e,B}G.$$

Alice sends Bob her public keys and Bob sends Alice his public keys. The static keys could have been transferred prior to initial contact and the ephemeral keys when contact is initiated. This is system dependent. To prevent duplicating all the equations I'm going to look at Alice's side of the calculation in the following description. The A and B subscripts swap for Bob's side.

Once both sides have each other's public keys Alice computes what NIST calls an im-

plicit signature with her own keys using the formula

$$s_A = k_{e,A} + avf(P_{e,A}.x)k_{s,A}$$

where $P_{e,A}.x$ is the x coordinate of the point $P_{e,A}$.

NIST defines the routine $avf()$ as the associate value function. This chops an x value in half and then sets a bit at the halfway point. Since the security of the system goes as $\sqrt{n}$ this sort of makes sense.

$$avf(x) = x \bmod 2^{\lceil \log_2(n)/2 \rceil} + 2^{\lceil \log_2(n)/2 \rceil}$$

where the symbol $\lceil\ \rceil$ means next largest integer (i.e. round up) and n is the order of the base point G.

Next each side computes a point using the other sides public keys. This formula is

$$U_B = P_{e,B} + avf(P_{e,B}.x)P_{s,B}.$$

The shared secret is then computed as

$$z = (s_A U_B).x$$

and this value will be in the range $\{0 \cdots p-1\}$. The NIST standard also multiplies by the cofactor of the curve. That seems like a security overkill to me since all the points are in the prime order group. See section 5.7.2.3 of NIST (2018) for details.

To see how both sides get the same value let's follow both sides through the above description as in table 4.1. Alice's column duplicates the above equations. Bob's column swaps the subscripts A and B because he has his private keys and Alice's public keys. The public keys are the private keys multiplied with the base point G. Expanding the middle line using the public key formulas we see that the last line in each column shows the final result is identical.

Table 4.1 MQV expanded calculation

Alice	Bob
$s_A = k_{e,A} + \mathsf{avf}(P_{e,A}.x)k_{s,A}$	$s_B = k_{e,B} + \mathsf{avf}(P_{e,B}.x)k_{s,B}$
$U_B = P_{e,B} + \mathsf{avf}(P_{e,B}.x)P_{s,B}$	$U_A = P_{e,A} + \mathsf{avf}(P_{e,A}.x)P_{s,A}$
$s_A U_B = (k_{e,A} + \mathsf{avf}(P_{e,A}.x)k_{s,A})(k_{e,B} + \mathsf{avf}(P_{e,B}.x)k_{s,B})G$	$s_B U_A = (k_{e,B} + \mathsf{avf}(P_{e,B}.x)k_{s,B})(k_{e,A} + \mathsf{avf}(P_{e,A}.x)k_{s,A})G$

Since all the points are related to the base point G the effective computation has both sides computing the same result. The impossible task of finding the private key values from the public information is what allows this to work.

Exercise 4.2

Assuming the static and ephemeral key points exist, how many point multiplications are required to compute the MQV algorithm for one person?

4.2.2 MQV code

In the following sections, the code to implement MQV is described.

Now it is time to turn all that math into code. I start with the generation of ephemeral keys using pseudo random numbers. I'll then describe the simple associate value function. The full MQV code is shown after that. Converting the math into code is actually now simple because we have the hard part of summing and multiplying points behind us.

EPHEMERAL KEYS

This section describes a routine to create a random ephemeral key for use with the MQV algorithm.

In addition to the static key the MQV algorithm uses an ephemeral key. These are randomly generated. A cryptographically secure random number generator should use hardware. Radioactive decay or thermal junction noise on a diode are typical sources available. To really dive deep check out https://csrc.nist.gov/projects/random-bit-generation. We'll just use the pseudo random generator from GMP as shown in listing 4.5.

The function takes the system parameters as input. The private and public keys are output. The private key is just a random number. The public key is that random number multiplied with the base point. While this function is cryptographically simple making it secure in a system might be more challenging.

Listing 4.5 Ephemeral key generator

```
void mqv_ephem(mpz_t ephm, POINT *Eph, BASE_SYSTEM bse)
{
  mrand(ephm);   ← random number for ephemeral private key
  elptic_mul(Eph, bse.Base, ephm, bse.E);
}
```

ASSOCIATE VALUE FUNCTION

This section describes the code which implements the NIST associate value function.

The implementation of the `avf()` is shown in listing 4.6. The computation of $x \bmod 2^{\lceil \log_2(n)/2 \rceil}$ is a simple mask of the lower half of the number of bits in the order of the base point. Since the order is always prime, the ceiling function takes us to the next integer after dividing by two.

The mask is created by brute force. Setting one bit at a time is slow, but obvious. Using lower level routines this can be accomplished far more quickly. After the input value is masked, the bit at the halfway point is set.

Listing 4.6 Associate value function

```
void avf(mpz_t z, mpz_t x, BASE_SYSTEM bse)
{
  long f, i;
  mpz_t mask;
```

```
  mpz_init(mask);
  f = (mpz_sizeinbase(bse.order, 2) » 1) + 1; ← f = n/2 + 1
  for(i=0; i<f; i++)
    mpz_setbit(mask, i);      | mask has f bits set
  mpz_and(z, x, mask);        | apply mask to input
  mpz_setbit(z, f); ← bit f is always set
  mpz_clear(mask);
}
```

FULL ECC MQV

This section describes what NIST calls the full ECC MQV algorithm.

The NIST Full ECC MQV routine is shown in listing 4.7. There are basically two steps. The first is to compute the value s with the local sides private data modulo the order of the base point. The second is to compute the point U using the far sides public key values. The multiplication then acts like a Diffie-Hellman operation. The NIST requirement also adds the curve cofactor. Here, I check to see if the cofactor is greater than 1 because there is no point doing the multiply unless it is.

Listing 4.7 Full ECC MQV

```
void mqv_share(mpz_t keyshare, mpz_t my_key, POINT MY_KEY,
               mpz_t my_ephm, POINT My_Ephm,
               POINT Their_key, POINT Their_Ephm,
               BASE_SYSTEM bse)
{
  POINT U;
  mpz_t s, z;

  mpz_inits(s, z, NULL);      | compute s = k + R.x * sk
  avf(z, My_Ephm.x, bse);     | s = k_e + avf(P_e.x)k_s
  mod_mul(s, z, my_key, bse.order);
  mod_add(s, s, my_ephm, bse.order);

  point_init(&U);             | compute U = R' + R.x' * P'
  avf(z, Their_Ephm.x, bse);  | U = P_e + avf(P_e.x)P_s
  elptic_mul(&U, Their_key, z, bse.E);
  elptic_sum(&U, U, Their_Ephm, bse.E);

  elptic_mul(&U, U, s, bse.E); ← shared secret = sU | compute key share value = r*s*U (x component)
  if(bse.cofactor > 1)        | cofactor > 1 requires multiply
  {
    mpz_set_ui(z, bse.cofactor);
    elptic_mul(&U, U, z, bse.E);
  }
  mpz_set(keyshare, U.x);     | return x component for shared secret
  point_clear(&U);
  mpz_clears(s, z, NULL);
}
```

Using the Full MQV algorithm requires more communication than Diffie-Hellman.

Both sides transmit their ephemeral public key to the other side in addition to their static public key. The other versions all require some communication of ephemeral keys, so even if you don't use the Full version some initial transfer of data is required. That is one reason this key sharing method is so useful - you create a new key every time you connect. In terms of cryptographic security an outside attack is essentially impossible. Just make sure your implementation deals with other problems such as unknown keyshare attack.

4.3 Example test code

In the following few sections, example code to show how Diffie-Hellman and MQV algorithms can be used are presented.

To test the code we first require curves to work with. And the choice of curve depends on the prime chosen for the finite field. In chapter 6 I will explain the choice of primes. For now, I will use curve parameters found using methods which I'll describe later. The obvious missing component in these tests is the actual transmission of the key data over the network.

In the following description I first list all the curves and base point parameters which are input files to the test program. I wrote a trivial routine to skip text lines from the input file, so I can easily convert numbers from a human-readable file. The test program uses both the Diffie-Hellman routines and the MQV routines so the same private and public keys are used for each case. The two sides of the transmission are called "my side" and "their side". Private keys for my side are input phrases and private keys for their side are fixed strings of random data.

4.3.1 Test curves

This section describes curves found using the method in chapter 6. It also describes an example program which executes the key exchange subroutines.

Testing the above routines requires a curve and base point. One can use NIST approved curves, or you can find your own as shown in chapter 6. If you find your own, you should make sure they are not susceptible to various attacks. For these examples I found 4 curves at different security levels and then checked that each subroutine worked at that level. The subroutines didn't work the first time!

Listings 4.8 through 4.11 show 4 different files with curve and point parameters. Listing 4.8 is a 160 bit curve using field prime $43 \cdot 2^{158} + 1$.

Listing 4.8 Test curve: secure 160 bit

```
File: Curve_160_params.dat
prime
ac0000000000000000000000000000000000000001
order
ac0000000000000000006543ba11adf8eb6345c77
cofactor
1
curve(a4    a6)
```

```
1
782e
basepoint(x    y)
1680bbdc87647f3c382902d2f58d2754b39bca877
a08957b09764ae59da8fb3058efef9c428e497268
```

Listing 4.9 is a 256 bit curve using field prime $43 \cdot 2^{252} + 1$.

Listing 4.9 Test curve: secure 256 bit

```
File: Curve_256_params.dat
prime
2b0000000000000000000000000000000000000000000000000000000000000001
order
2b0000000000000000000000000002e7f521c85bba055a6e2161b956a47f69
cofactor
1
curve(a4    a6)
1
a87
basepoint(x    y)
2310115d283e49377820195c8e67781b6f112a625b14b747fa4cc13d06eba0919
51277aeb91946f0cb83053a10f67c5a9ef00a4f0cf2466b3bedf4fdcd774b574
```

Listing 4.10 is a 384 bit curve using field prime $23 \cdot 2^{381} + 1$.

Listing 4.10 Test curve: secure 384 bit

```
File: Curve_384_params.dat
prime
2e0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000001
order
2e0000000000000000000000000000000000000000000000002275cc5f2f7fcc15352a2c993900a851b
3a75365a9ac54733
cofactor
1
curve(a4    a6)
1
310
basepoint(x    y)
23c0d9fcfaa3dc18b1eff7e89bf7678636580d17dd84a873b14b9c0e1680bbdc87647f3c382902d2f
58d2754b39bca874
28d7205f1be0a725d2aa7c3386f2e0b0ea7c558ca19f9770cdc72f91a1cbc262687810d4c5bd53681
8ccfa49aae2ed0cc
```

Listing 4.11 is a 512 bit curve using field prime $113 \cdot 2^{509} + 1$.

Listing 4.11 Test curve: secure 512 bit

```
File: Curve_512_params.dat
prime
e200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000001
order
e200000000000000000000000000000000000000000000000000000000007788830d091dc57e3
af7d7bbd15386ee9414602d88d1e6489cd056336922bbf4d
cofactor
1
curve(a4   a6)
1
41
basepoint(x   y)
518f204fe6846aeb6f58174d57a3372363c0d9fcfaa3dc18b1eff7e89bf7678636580d17dd84a873b
14b9c0e1680bbdc87647f3c382902d2f58d2754b39bca875
c9fe7223aca476cde61f206be285898475f1dcbaefeda90057d3b8bae5146f3016ebf2139daa73f39
417193e8609a4229cd4c58389e4b9095fafcd68362b310fe
```

One of these files is read in at the top of the test routine as shown in listing 4.13. The command line argument is one of `160, 256, 384` or `512`. That parameter file is read in and then parsed into the correct arguments. The parsing process includes skipping text lines in the data file. The line skipping subroutine is shown in listing 4.12. The index is maintained, so I track where each line is in the buffer.

Listing 4.12 Key exchange text code: skip line subroutine

```
int skipln(char *bfr, int strt, int skp)
{
  int i;

  i = strt;
  while(skp)
  {
    while(bfr[i] != '\n') i++;
    i++;
    skp--;
  }
  return i;
}
```

index start

number of lines to skip

index points to start of next line

The `prime` value is the field over which the curve is computed. The `order` value is the order of the base point. All these curves have a `cofactor` of 1, so this is also the cardinality of the curve. The equation of the curve (coefficients a_4 and a_6) as well as the (x, y) coordinates of the base point are included. Once the values are converted and placed into the `BASE_SYSTEM` structure the test is ready to begin.

Listing 4.13 Key exchange test code: input curve parameters

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

```
{
  FILE *parm;
  mpz_t prm, sk, sok;
  long lvl;
  BASE_SYSTEM base;
  POINT Pk, Pok, Rk, Rok;
  char filename[256], *bufr, *ptr;
  int i, k;
  mpz_t myshare, theirshare, myrand, theirrand;

  if(argc < 2)
  {
    printf("Use: ./base_test <level>\n");
    exit(-1);
  }
  lvl = atol(argv[1]);
  sprintf(filename, "Curve_%ld_params.dat", lvl);
  mpz_inits(prm, base.order, NULL);
  parm = fopen(filename, "r");
  if(!parm)
  {
    printf("can't find file %s\n", filename);
    exit(-2);
  }
  bufr = (char*)malloc(1024*4);
  i = 0;
  while((!feof(parm)) && (i < 1024))
  {
    bufr[i] = fgetc(parm);
    i++;
  }
  fclose(parm);

  i = skipln(bufr, 0, 1);
  gmp_sscanf(&bufr[i], "%Zx", prm);
  gmp_printf("%Zd\n", prm);
  minit(prm);
  i = skipln(bufr, i, 2);
  gmp_sscanf(&bufr[i], "%Zx", base.order);
  gmp_printf("%Zd\n", base.order);
  i = skipln(bufr, i, 2);
  sscanf(&bufr[i], "%ld", &base.cofactor);
  i = skipln(bufr, i, 2);
  curve_init(&base.E);
  gmp_sscanf(&bufr[i], "%Zx %Zx", base.E.a4, base.E.a6);
  gmp_printf("E: %Zx  %Zx\n", base.E.a4, base.E.a6);
  i = skipln(bufr, i, 3);
  point_init(&base.Base);
  gmp_sscanf(&bufr[i], "%Zx %Zx", base.Base.x, base.Base.y);
  point_printf("Base point: ", base.Base);
```

verify input level exists

get data file else complain

read in whole file to buffer

convert text to big numbers

field prime initialized

base point order and cofactor initialized

curve parameters initialized

base point initialized

The next step is common to both Diffie-Helmann and MQV. The private key is created using a pass phrase. This code is shown in listing 4.14. I then create "the other side's" keys using random numbers.

Listing 4.14 Key exchange test code: secret key generation

```
printf("Input pass phrase to generate secret key: ");
fflush(stdout);
i = 1024;                                          read in
getline(&bufr, (size_t*)&i, stdin);                key phrase
mpz_init(sk);
point_init(&Pk);
gen_key(sk, &Pk, bufr, base); ← convert phrase to public, private key
gmp_printf("secret key: %Zx\n", sk);
point_printf("Public key: ", Pk);

mpz_init(sok);
point_init(&Pok);                                  random data for "other side" key
sprintf(bufr,
"Secret Key Test For Other Side 157  164 218 149 124 108 253 26 40 ");

gen_key(sok, &Pok, bufr, base); ← convert random data to public, private key
point_printf("Other side Public key: ", Pok);
```

The "o" in `sok` and `Pok` is for "other side".

4.3.2 Diffie-Hellman test routines

This section describes the test code to simulate Diffie-Hellman key exchange.

With the secret and public keys generated for each side, we can now call the Diffie-Hellman routine to see that both sides get the same shared secret. The communications can't be done in this test, but we can ensure the math works.

Listing 4.15 shows how both sides calculate the same thing. "My side" uses the secret key created from the pass phrase and the "other" sides public key. "Their side" uses their randomly generated secret key and "my" public key. We then check to see if the results match.

Listing 4.15 Key exchange test code: Diffie-Hellman

```
mpz_inits(myshare, theirshare, NULL);
diffie_hellman(myshare, sk, Pok, base.E);          swap private and public keys
diffie_hellman(theirshare, sok, Pk, base.E);       for each side
if(!mpz_cmp(myshare, theirshare))
  printf("Keys match.\n");
else
  printf("Keys DON'T match!\n"); ← go find bugs!
gmp_printf("my keyshare:    %Zx\n", myshare);
gmp_printf("their keyshare: %Zx\n", theirshare);
```

Here's an example run with `256` bit security:

```
./base_test 256
:
:
Input pass phrase to generate secret key: this is another test
:
:
Keys match.
my keyshare:    df6363311da84770ea7779d9d2bf3991fc41548347041af34dcbb71b6abe72cf
their keyshare: df6363311da84770ea7779d9d2bf3991fc41548347041af34dcbb71b6abe72cf
```

4.3.3 MQV test routine

This section shows a simulation of the MQV key exchange code.

Once we have static keys for a Diffie-Hellman type exchange, we can easily generate ephemeral keys using a random number generator. This process is shown in listing 4.16.

Listing 4.16 Key exchange test code: MQV ephemeral keys

```
generate random secret and public keys for MQV test
    mpz_inits(myrand, theirrand, NULL);
    point_init(&Rk);                          initialize
    point_init(&Rok);                         ram space
    mqv_ephem(myrand, &Rk, base);             create
    mqv_ephem(theirrand, &Rok, base);         random keys
```

The last part of the MQV test is to compute each side's shared secret. This is shown in listing 4.17. Each side only has to call the `mqv_share()` function once. But for this test we want to see that both sides actually do get the same value, so the routine is called twice.

Listing 4.17 Key exchange test code: MQV full shared secret

```
Each side sends the
other the public key,
and then computes
the shared secret
    mqv_share(myshare, sk, Pk, myrand, Rk, Pok, Rok, base); ← "my" side key

    mqv_share(theirshare, sok, Pok, theirrand, Rok, Pk, Rk, base); ← "their" side key
    if(!mpz_cmp(myshare, theirshare))
      printf("MQV Keys match.\n");
    else
      printf("MQV Keys DON'T match!\n"); ← go find bugs!
    gmp_printf("my keyshare:    %Zx\n", myshare);
    gmp_printf("their keyshare: %Zx\n", theirshare);
```

The output from the same `256` bit test is shown here:

```
MQV Keys match.
my keyshare:    9a2b4135e9e39f0daf8aa49a37d22ee551c96a3e6bf0f2c7e6056782d7a8166c
their keyshare: 9a2b4135e9e39f0daf8aa49a37d22ee551c96a3e6bf0f2c7e6056782d7a8166c
```

4.4 Summary

- In elliptic curve key exchange both sides compute the same point and use the x value for the shared secret.
- Diffie-Hellman uses one person's private key and another person's public key to create a shared secret key. The secret key is used with a standard single key encryption algorithm.
- A hash function can be used to generate random bits from a text phrase. This creates a private key which does not require storage. The public key, private key pair is always the same for a given curve and base point.
- The Menezes-Qu-Vanstone (MQV) algorithm uses ephemeral private, public keys in addition to the users static private, public key pairs to create perfect forward secrecy.
- The NIST version of MQV includes an associate value function which outputs half an x value.
- MQV combines two private keys and two public keys which are multiplied to form a shared secret point. The x value is then used as the shared secret key.
- Once the ephemeral keys are created they must be communicated between both sides. The actual calculations are quick.

Chapter Bibliography

Menezes, Alfred, Qu, M., & SA, Vanstone. 1995. Some key agreement protocols providing implicit authentication. *2nd Workshop on Selected Areas in Cryptography (SAC '95)*, April, 22–32. 37

NIST. 2001 (Nov.). *Federal Inf. Process. Stds. (NIST FIPS) - 197*. https://www.nist.gov/publications/advanced-encryption-standard-aes. 32

NIST. 2016. *NIST Special Publication 800-185, SHA-3 Derived Functions*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf. 34

NIST. 2018. *NIST Special Publication 800-56A Revision 3*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf. 37, 38

4.5 Answers to exercises

- 4.1) No. Clare's public key is $C = k_C G$ which can only be used to create two secret keys. One between Clare and Bob is $k_C B$ and the secret between Clare and Alice is $k_C A$.
- 4.2) Two. The value s_A in an integer modulo a prime. The value U_B has one point multiplication. The secret is found using $s_A U_B$ which is the second point multiplication.

5 Prime field elliptic curve digital signatures explained

This chapter covers

- Digital signature using private key
- Verification using public key
- Schnorr algorithm
- NIST ECDSA algorithm

In this chapter I describe two algorithms used for digital signatures.

A digital signature creates proof of authorship using mathematics. A person's private key is used along with a hash of a document to create a signature. The public key can then be used with the local hash of the same document to verify the signature. The connection between a person and their keys can be checked with key exchange methods or certificates in a database. Here we assume that a private key, public key pair only applies to one person, and their signature can be verified or rejected depending on whether they actually digitally signed a document or not.

Two methods of digital signature will be discussed in detail here.

- Schnorr signature
- NIST elliptic curve digital signature algorithm (ECDSA)

The Schnorr signature algorithm is short and concise. The ECDSA algorithm is a bit more

involved. The Schnorr algorithm hashes the combination of a document with a point and verifies using that point. ECDSA goes through more math and outputs two values. Those values are used along with public points to do the verification. Schnorr requires more space to hold the data than ECDSA but takes a touch less time to compute. Each has a place depending on circumstances.

5.1 Schnorr digital signature

In this section the Schnorr digital signature is explained.

The original version of the Schnorr digital signature scheme is over exponentials of numbers modulo a prime. By modifying it for use on elliptic curves we can increase security and decrease the size of the primes. Fewer resources required implies lower cost.

5.1.1 Schnorr elliptic curve math

This section describes the mathematics used to compute the Schnorr digital signature.

Figure 5.1 shows the similarity of the Schnorr algorithm to the key exchange algorithms described in chapter 4. A (public key, private key) pair is generated over a secure curve. We'll take the private key to be s_p and the curve to be the same as equation 3.1 reproduced here

$$y^2 = x^3 + a_4x + a_6$$

with base point B which has prime order n. Then the public key is

$$P_p = s_pB.$$

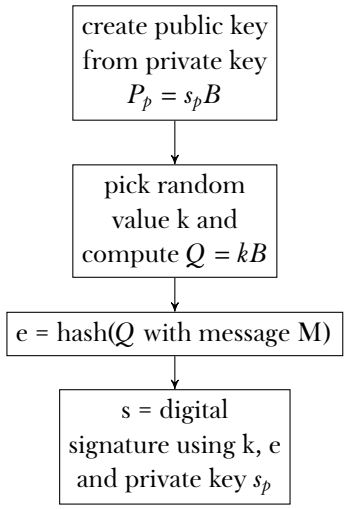

Figure 5.1 Schnorr signature steps

The generation of a signature begins by choosing a random value k in the range of the order of the base point $\{1 \cdots n - 1\}$. For security, it's advisable to choose a new k for

each signature. Once a signature is created, the value of k can be forgotten. Low Hamming weight (i.e. only a few bits set) for k is discouraged. For example, a value of k like 0x000A0008000050001 would be a poor choice as would 0xFFFF5FFFAFFF7FFE. If the number of bits set is about half the number of bits in the field size it has maximal security.

A new public point is computed using

$$Q = kB.$$

The document M is concatenated with the point Q and then hashed to a value which is the order of the base point. We write this as

$$e = \text{hash}(Q.x||Q.y||M) \bmod n. \tag{5.1}$$

The symbol || means to place the bit strings in sequence. Because it is a hash the endianness does not matter so long as everyone is consistent.

This value is combined with the random value k and the private key s_p using

$$s = k - e \cdot s_p \bmod n.$$

The signature is then the point Q and the value s.

To verify that the document M actually was signed by public key P_p the verifier has to compute e as in formula 5.1 and then check that

$$Q \stackrel{?}{=} sB + eP_p. \tag{5.2}$$

If it doesn't match, the signature is rejected.

To see why this works, put the expanded value of s into 5.2. Multiply through with B to get

$$kB - e \cdot s_pB + eP_p.$$

Since $P_p = s_pB$ the last two terms cancel and the result matches.

Exercise 5.1

In equation 5.2, how many times does the random value k appear? Hint: twice in equation 5.1.

5.1.2 Schnorr sign subroutine

The Schnorr digital signature subroutine is described in this section.

Since digital signatures use multiple components, it makes sense to create a structure to hold both of them. Listing 5.1 shows the structure I used for Schnorr signatures.

Listing 5.1 Schnorr structure

```
typedef struct
```

```
{
  POINT Q;
  mpz_t s;
}SCHNORR;
```

The tricky part of the Schnorr routine is converting the x and y components into bytes. Leaving `NULL` bytes can lead to problems if different machines have different word sizes. Listing 5.2 shows how I solve this using the `mpz_sizeinbase()` with size 16 which counts nibbles. Dividing that by two gives the correct number of bytes for each component. In a way it is kind of silly because I then multiply by two again to create space for each component. But this way it is clear what is going on. The extra two bytes in the `malloc` call are to prevent accidents from happening.

Listing 5.2 Schnorr sign

```
void schnorr_sign(SCHNORR *sig, mpz_t sk, POINT Pk,
                  unsigned char *msg, long len, BASE_SYSTEM base)
{
  mpz_t k, e, tmp;
  unsigned char *cat;
  int xsz, i;

  mpz_inits(e, k, tmp, NULL);
  mrand(k);                                          random number
  elptic_mul(&sig->Q, base.Base, k, base.E);         point for signature
  xsz = (mpz_sizeinbase(sig->Q.x, 16) + 1)/2;  ← get number bytes
  cat = (unsigned char*)malloc(len + 2*xsz + 2);
  mpz_export(cat, NULL, -1, 1, 0, 0, sig->Q.x);      convert
  mpz_export(&cat[xsz], NULL, -1, 1, 0, 0, sig->Q.y); x and y
  for(i=0; i<len; i++)                               to strings
    cat[2*xsz + i] = msg[i];  ← add document to string buffer
  hash(e, cat, len+2*xsz, base.order);
  mod_mul(tmp, sk, e, base.order);                   s = k + e · s_k mod n
  mod_sub(sig->s, k, tmp, base.order);
  mpz_clears(e, k, tmp, NULL);
}
```

The function `mpz_export()` is used to convert a large integer into a string of bytes. The first argument is the buffer where the bytes are placed. The second argument is the output number of words, which we don't need, so this is simply NULL. The next two arguments are order and size. `order` = -1 means little endian, `size` = 1 means use byte size values. The next two zeros are not used because we are using bytes for output (these values deal with multibyte results). The last argument is the large integer input in GMP format.

Note the message is placed after the x and y components in the `cat` buffer. Once everything is in place the `hash()` function is called and the s component of signature is computed.

5.1.3 Schnorr verify subroutine

The Schnorr digital verification subroutine is described in this section.

A digital signature is usually attached to a digital document as part of the same file. The file header explains where the different parts of a document reside within the file. A way to find the public key for the signer should also be in the file. If the public key is attached to the file then you would still have to verify that the public key was real and not faked. This is where security becomes an independent issue from cryptography, and it is important to get right in real world applications.

In listing 5.3 we are going to assume the document and signature data are already separated. In addition, we are going to assume that the public key of the signer has been acquired securely.

The verify routine is similar to the sign routine because the hash of the point Q with the message is the same process. Listing 5.3 shows the verify subroutine.

Listing 5.3 Schnorr verify

```
int schnorr_verify(SCHNORR sig, POINT Pk, unsigned char *msg,
                   long len, BASE_SYSTEM base)
{
  mpz_t e;
  unsigned char *cat;
  int xsz, i;
  POINT U, V, Qck;

  mpz_init(e);
  xsz = (mpz_sizeinbase(sig.Q.x, 16) + 1)/2;   <- get number bytes
  cat = (unsigned char*)malloc(len + 2*xsz + 2);
  mpz_export(cat, NULL, -1, 1, 0, 0, sig.Q.x);      convert
  mpz_export(&cat[xsz], NULL, -1, 1, 0, 0, sig.Q.y); x and y
                                                     to strings
  for(i=0; i<len; i++)
    cat[2*xsz + i] = msg[i];   <- add document to string buffer
  hash(e, cat, len+2*xsz, base.order);
  point_init(&U);
  point_init(&V);    <- compute hash
  point_init(&Qck);
  elptic_mul(&U, base.Base, sig.s, base.E);   Q' = sB + eP_k
  elptic_mul(&V, Pk, e, base.E);
  elptic_sum(&Qck, U, V, base.E);
                     verify computed Q matches
                     signature Q
  if((!mpz_cmp(sig.Q.x, Qck.x)) && (!mpz_cmp(sig.Q.y, Qck.y)))
    i = 1;
  else                save verify
    i = 0;            result so we
  mpz_clear(e);       can clear
  point_clear(&U);    variables
  point_clear(&V);
```

```
  point_clear(&Qck);
  return i;
}
```

The computation of e is clearly the same as the signature routine. Then I use three points to build the check. The first point is

$$U = sB$$

the second point is

$$V = eP_k$$

and the check point is their sum

$$Q_{ck} = U + V.$$

If both the x and y components match between the check point Q_{ck} and input signature point Q, the signature is verified.

5.1.4 Schnorr test example

A simulated example of how to use the Schnorr digital signature and verify routines are described in this section.

I added the Schnorr test code to the file with key exchange testing from chapter 4. To perform this test I searched my drive for a text file that was reasonably small. I copied it over to my working directory as `sign_test.txt`. Listing 5.4 shows how I read in the sample.

Listing 5.4 Test message input

now create a test for a digital signature

```
parm = fopen("sign_test.txt", "r");
if(!parm)
{
  printf("sign_test.txt not found??\n");
  exit(-7);
}
k=0;
while((!feof(parm)) && (k < 4*1024))
{
  bufr[k] = fgetc(parm);
  k++;
}
k -= 2;
```

always check for errors

read document into RAM

Using the same public and private keys generated during the test shown in listing 4.14 and the curve parameters from listings 4.8 through 4.11, the Schnorr algorithm test is very simple as seen in listing 5.5.

Listing 5.5 Schnorr test

```
snr_init(&snr);
schnorr_sign(&snr, sk, Pk, bufr, k, base); ← Schnorr sign document

if(schnorr_verify(snr, Pk, bufr, k, base)) ← verify signature
  printf("Schnorr message verifies!\n");
else
  printf("Schnorr message does not match original signed.\n");  ← go look for bugs!
```

When run with any of the curves the output every time is `Schnorr message verifies!` The trick to signatures is making sure the message is the same every time. Even one bit wrong will cause the verify to fail. Packaging the message is a system level problem.

5.2 NIST ECDSA

In this section the NIST ECDSA algorithm is explained.

Another digital signature form is the standard from NIST. It is simply called the elliptic curve digital signature algorithm (ECDSA). It is similar to Schnorr in that a hash of the message is used. I'll follow the draft standard (NIST, 2019) which has a ton of details not included here. Figure 5.2 shows the steps involved in computing the signature.

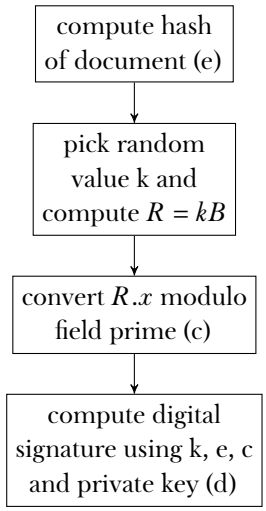

Figure 5.2 ECDSA signature steps

The first step in computing the signature is to compute the hash of the message and convert that to a value modulo the order of the base point n. This is the hash function described in listing 4.1 so we take

$$e = \text{hash}(M) \bmod n. \tag{5.3}$$

The second step is to pick a random number $k \pmod n$ and compute the point

$$R = kB.$$

The x component is the size of the field prime, so we convert it to the order of the base point using

$$c = R.x \bmod n. \tag{5.4}$$

Assuming the signers private key is s_p and their public key is $P_p = s_p B$ we finally compute

$$d = k^{-1}(e + c \cdot s_p) \bmod n. \tag{5.5}$$

The signature is then the pair of values (c, d).

To verify the signature, the hash of the document is done the same as in equation 5.3 to find the value e'. Then we compute

$$h = d^{-1} \bmod n \tag{5.6}$$

$$h_1 = e'h \bmod n \tag{5.7}$$

$$h_2 = ch \bmod n \tag{5.8}$$

From these values we find the point

$$R' = h_1 B + h_2 P_p \tag{5.9}$$

and then check that

$$R'.x \stackrel{?}{=} c.$$

So let's see how this works. Expanding h we have

$$h = \frac{k}{(e + c \cdot s_p)}.$$

The first term in R' is

$$h_1 B = e' \cdot hB = \frac{e' \cdot kB}{(e + c \cdot s_p)}.$$

The second term is

$$h_2 P_p = c \cdot hP_p = \frac{c \cdot k \cdot s_p B}{(e + c \cdot s_p)}$$

Adding these together we have

$$R' = k\frac{e' + c \cdot s_p}{e + c \cdot s_p} B.$$

So as long as the message has not been altered, $e' = e$, the fraction divides out, and we end up with $R' = R$ which implies $R'.x = c$.

Exercise 5.2

Equations 5.3 through 5.8 are all modulo n. Is n the field prime or the largest prime in the elliptic curve cardinality?

5.2.1 ECDSA sign subroutine

This section describes the code which implements ECDSA signatures.

Like the Schnorr signature, a structure for the ECDSA signature is very useful. The structure is shown in listing 5.6 where the `c` value is from equation 5.4 and the `d` value is from equation 5.5.

Listing 5.6 ECDSA structure

```
typedef struct
{
  mpz_t c;
  mpz_t d;
}ECDSA;
```

Listing 5.7 shows the signing subroutine. The inputs are private and public keys, the message and its length and system parameters (`BASE_SYSTEM` structure shown in listing 4.2).

Listing 5.7 ECDSA signing subroutine

```
void ecdsa_sign(ECDSA *sig, mpz_t sk, POINT Pk,
                unsigned char *msg, long len, BASE_SYSTEM base)
{
  mpz_t k, e, tmp;
  POINT R;

  mpz_inits(e, k, tmp, NULL);
  hash(e, msg, len, base.order);
  mrand(k);
  point_init(&R);
  elptic_mul(&R, base.Base, k, base.E);
  mpz_mod(sig->c, R.x, base.order);
  mod_mul(tmp, sk, sig->c, base.order);
  mod_add(tmp, tmp, e, base.order);
  mod_div(sig->d, tmp, k, base.order);
  point_clear(&R);
  mpz_clears(e, k, tmp, NULL);
}
```

e = hash of document

random value and matching point

$c = R.x \bmod n$

$d = \frac{e + s_k c}{k}$

Since the modulo arithmetic is done on the order of the base point, the `mod_*()` routines are used instead of the `m*()` routines. Because the `mod_div()` routine computes the inverse for us, there is no need to compute it separately. I just divide by k mod n.

5.2.2 ECDSA verify subroutine

The ECDSA verify routine is described in this section.

The verification routine for ECDSA is a straightforward calculation of equations 5.8 through 5.9. Like Schnorr, this routine evaluates two points and then sums them to get the final point used to check the signature.

Listing 5.8 ECDSA verify subroutine

```
int ecdsa_verify(ECDSA sig, POINT Pk, unsigned char *msg, long len,
                 BASE_SYSTEM base)
{
  mpz_t h, h1, h2, e;
  POINT R, S, T;
  int rtn;

  mpz_inits(h, h1, h2, e, NULL);
  hash(e, msg, len, base.order);
  mpz_invert(h, sig.d, base.order);
  mod_mul(h1, e, h, base.order);
  mod_mul(h2, sig.c, h, base.order);
  point_init(&T);
  elptic_mul(&T, Pk, h2, base.E);
  point_init(&S);
  elptic_mul(&S, base.Base, h1, base.E);
  point_init(&R);
  elptic_sum(&R, T, S, base.E);
  mpz_mod(h, R.x, base.order);
  if(!mpz_cmp(h, sig.c))
    rtn = 1;
  else
    rtn = 0;
  point_clear(&R);
  point_clear(&S);
  point_clear(&T);
  mpz_clears(h, h1, h2, e, NULL);
  return rtn;
}
```

e = hash of message

$h = \frac{1}{d}$

$h_1 = eh$

$h_2 = ch$

$R = h_1 B + h_2 P_k$

convert R.x mod n

save verify result so we can clear variables

As with the Schnorr algorithm, the message must not change at all between signing and verifying. The main difference between ECDSA and Schnorr is that the latter uses a full point for comparison. With ECDSA, we must reduce the field element from a point to the order of the curve. The advantage of ECDSA is the size of the signature is two `mpz_t` elements, with Schnorr signature taking up three.

5.2.3 ECDSA example code

This section shows a simulated ECDSA sign and verify test routine.

The test of the subroutines uses the same key generation as the previous tests as well as using the same message text in the Schnorr example. Listings 4.8 through 4.11 have the base system data which is selected as shown in listing 4.13. Listing 5.9 shows how to call the signing subroutine using these same arguments.

Listing 5.9 ECDSA signing example

```
sig_init(&sig);
```

```
ecdsa_sign(&sig, sk, Pk, bufr, k, base); ← sign message with private key
```

Similarly, the verification is really simple as shown in listing 5.10.

Listing 5.10 ECDSA verify example

```
if(ecdsa_verify(sig, Pk, bufr, k, base)) ← verify signature with message and public key
    printf("message verifies!\n");
else
    printf("message does not match original signed.\n"); ← go look for bugs!
```

Running this test with every one of the example curves and some arbitrary pass phrase with the same test file used in the Schnorr example gives the same output `message verifies!`

5.3 Summary

- Schnorr digital signature combines a random value times a base point (called Q)with a document and computes the hash of the combination.
- With the Schnorr digital signature, the signer's private key is combined with the hash value and random value to create the other part of the signature.
- Verification of Schnorr uses the signer's public key with the signature to check for a match. Since this requires a hash of the point Q and document, even one bit difference will fail.
- NIST created a draft standard for the elliptic curve digital signature algorithm (ECDSA) in 2019 (NIST, 2019). ECDSA uses the hash of a message, a random value and the signer's private key to create one value of the signature. The x component of the random value times the base point is the second value.
- To verify with ECDSA several calculations are combined with the hash of the document and the signature. These values are multiplied with the base point and signer's public key whose sum is then compared with a signature value. A match verifies and one bit error will fail.

Chapter Bibliography

NIST. 2019 (Oct.). *Federal Inf. Process. Stds. (NIST FIPS) - 186-5(draft)*. https://doi.org/10.6028/NIST.FIPS.186-5-draft. 55, 59

5.4 Answers to exercises

- 5.1) Six times! The values $Q.x$ and $Q.y$ in 5.1 each count for one k appearance. s contains one direct value of k plus two from e. So there are 5 appearances of k on the left of equation 5.2 and one on the right for a total of six.
- 5.2) The value n is the largest prime in the elliptic curve cardinality. This is chosen as the order of the base point to maximize security.

6 Finding good cryptographic elliptic curves

This chapter covers

- Using PARI/gp command line
- PARI library programming
- A program to find the number of points on an elliptic curve
- What constitutes a good curve

In this chapter I show how to find good cryptographic curves using mathematicians software tools. The ability to find and use many different cryptographically secure curves increases security by forcing attackers to work hard to find breaks on every possible curve. The resulting curves are good for the applications shown in chapters 4 and 5.

Up to this point we have assumed we know the cardinality of a curve (see section 4.1.1). Unfortunately the mathematics of computing the number of points on an elliptic curve over finite fields is really deep. For those who want to dig into the details I suggest starting with chapter 7 in (Blake *et al.*, 1999). For this chapter I am just going to use the mathematician's tool PARI/gp which has the point counting algorithms built in. In addition, the calculations are done as efficiently as the authors of PARI/gp know how.

Appendix A describes how to get hold of PARI/gp. Remember that if you are a Python advocate you can get SageMath which has PARI as just one of the options you can use.

PARI/gp has a great deal of mathematical tools. We only care about a small, but very important subset of those tools related to elliptic curve mathematics. I will describe the important mathematics and then explain how PARI/gp allows us to compute elliptic curve cardinalities.

6.1 PARI/gp for elliptic curves

In this section I describe the mathematicians tool PARI/gp. It is very useful for checking elliptic curve code.

Before getting into programming details I am going to introduce interactive use of the PARI/gp tool. The startup and how to set up elliptic curves over finite fields is described first. This is exceptionally useful for debugging code. Mistakes in computations can be found by duplicating programming steps using copy and paste entries from `print_point()` subroutines into PARI/gp.

The programming side of PARI using libpari will also be explained. This is what I use to find good curves. The method I used to eliminate poor curves will be explained and then the actual code to accomplish the task of finding good curves follows.

6.1.1 Starting PARI/gp

In this section I cover the start up of PARI/gp.

The program `gp` is a mathematics calculator with elliptic curve functions including finding the cardinality of curves and the order of points. When we first start `gp` we get output that looks like this:

```
$ gp
Reading GPRC: /home/drmike/.gprc
GPRC Done.

 ⋮

PARI/GP is free software, covered by the GNU General Public License, and comes
WITHOUT ANY WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?18 for how to get moral (and possibly technical) support.

PARIsizemax = 10000003072, primelimit = 500000
?
```

The line about "moral (and possibly technical) support" is very real. I have asked many stupid questions on the PARI mailing list and gotten a lot of very helpful answers. The `?` is the `gp` prompt.

The file `.gprc` is used to set up the PARI environment which is different from the default. In this case the file is simply:

```
PARIsizemax = 10000000000
read "PARI/funcs.gp";
```

The first line sets the heap to 10 GB, which on a 64 GB machine is reasonable. The

second line reads in a predefined function that I find useful. That file is simply

```
numbits(x)={floor(log(x)/log(2))+1}
```

which tells me how many bits are in a value x. When looking at a 50-digit number it is nicer to let PARI tell you how many bits it has.

6.1.2 PARI/gp elliptic curves over finite fields

In this section I describe how PARI/gp works with finite field elliptic curves.

PARI/gp has a function for creating elliptic curves using just a_4 and a_6. The function is `ellinit()` whose first argument is a single vector input $[a_4, a_6]$. The brackets [ ] are not optional because they tell gp that items contained within are components of a vector.

The second argument to `ellinit()` is the field. For the moment this will be a prime number which is the finite field we are using. This will be something more complicated when we dive into field extensions.

The manual says "The precise layout of the ell structure is left undefined and should never be used directly." They do define the first thirteen values as the common elliptic curve parameters

$$a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j$$

which you can find described (in the same order) in section III.1 of reference (Silverman, 2013). If the output we see is `[]` (referred to as "null") it means there is no curve which can be constructed with the given inputs.

Since we are working with finite fields there are several automatic values which PARI has access to when an `ell` structure is created. These are

- .no the number of points on the curve
- .cyc the cyclic structure of the curve
- .gen the generators of the curve
- .group the first three items as a vector [.no, .cyc, .gen]

As a simple example let's look at a curve over the field $p = 1187$. I pick $a_4 = 1$ and $a_6 = 17$. gp gives the result

```
? E=ellinit([1, 17], 1187)
%1 = [Mod(0, 1187), Mod(0, 1187), Mod(0, 1187), Mod(1, 1187), Mod(17, 1187), Mod(0,
1187), Mod(2, 1187), Mod(68, 1187), Mod(1186, 1187), Mod(1139, 1187), Mod(743, 1187),
Mod(910, 1187), Mod(95, 1187), Vecsmall([3]), [1187, [109, 236, [6, 0, 0, 0]]],
[0, 0, 0, 0]]
```

I then request the entire group information

```
? E.group
%2 = [1148, [1148], [[Mod(702, 1187), Mod(1007, 1187)]]]
```

This tells me that there are 1148 points on the curve, it is a simple cyclic curve and a point which can generate all other points on the curve is (702, 1007). The form `Mod(702,`

`1187)` means "702 modulo 1187" which is how PARI tracks a finite field number.

Exercise 6.1

How many points are on the curve $y^2 = x^3 + x + 97 \mod 95289871302753755165078396311$?

6.1.3 LibPARI with elliptic curves

In this section I show how to use PARI library subroutines with C programming.

While it is possible to write scripts in PARI/gp I find it easier to link directly with the PARI library, which is called libpari. The code initialization process is a touch different from the interactive command line program.

To initialize libpari we first specify the stack size and number of primes to precompute. In my code examples I use

```
pari_init(1024*1024*1024, 5*1024*512);
```

which gives 1 GB of stack and 2.5 million primes.

Libpari uses the type `GEN` which is a pointer to a set of longs. Internally it knows what each `GEN` object is. If you give a routine the wrong kind of object libpari will bail with an error. The debugging process is to first find the place where the error occurred and then attempt to figure out which argument was wrong.

6.2 General ordinary curves

In this section I describe an algorithm to find the number of points on an elliptic curve to determine if it is cryptographically useful.

The general equation for an ordinary elliptic curve over a large prime field is

$$E : y^2 = x^3 + a_4 x + a_6 \mod p \tag{6.1}$$

Choosing a_4 and a_6 at random will give us a random curve. Many curves are isomorphic which means they have the same number of points. For cryptographic purposes we want the same points for each user, but we don't really care which curve we pick so long as the cardinality of the curve is a large prime.

The relationship between the field p and the cardinality $\#E$ is given by Hasse's Theorem:

$$\#E = p + 1 - t \tag{6.2}$$

where the value of t is limited to

$$|t| \leq 2\sqrt{p} \tag{6.3}$$

(See chapter V in (Silverman, 2013).) This is an important relationship between cardinality and the field prime. If we can find a negative t which is odd the cardinality might be a prime which would be larger than the field. Since t has half as many bits as p that means the cardinality is still the same bit size as p. The variable t is called the trace of Frobenius.

Figure 6.1 shows the generic algorithm for seeking good curves. PARI is used to compute the cardinality. Then powers of 2 are removed one at a time until an odd number is

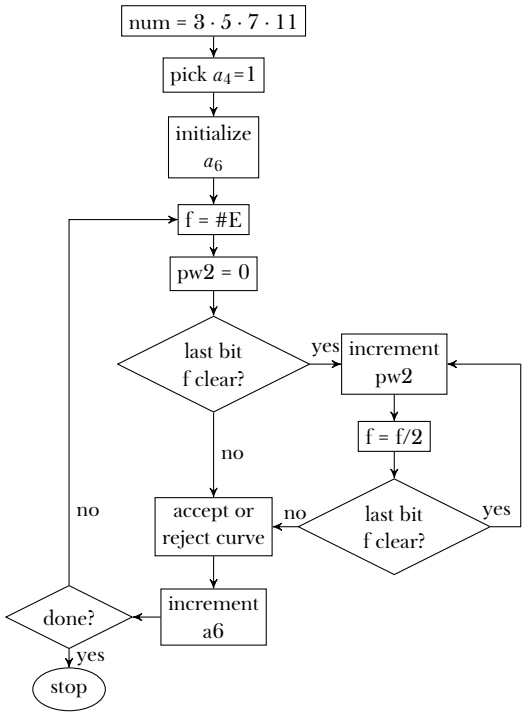

Figure 6.1 General curve finding algorithm

found. The algorithm for accept or reject is shown in figure 6.2. An accepted curve is output, a rejected curve is ignored. Either way, the next curve is investigated by incrementing the a_6 value.

My first requirement was to find curves with very small cofactors. So I created a number with the factors $3 \cdot 5 \cdot 7 \cdot 11$ and used the `gcdii()` function to test if any of those primes were present in the cardinality.

If #E has too many primes I want to ignore the curve. Figure 6.2 shows a flow chart of the logic. If the cardinality only has powers of two, which I track with variable $pw2$, and a remaining prime, then I output it. If the cardinality is not prime and there are no small factors after removing 2^{pw2}, the number can be ignored because the factors are not good enough for a secure curve.

Small factors are then removed one power at a time. The array pw[] holds each set of removed factors. As long as the array length is less than four, and we find a prime then all factors are output. Otherwise, the number is ignored because it does not have a large enough prime factor to use.

To create the elliptic curve parameters in listings 4.8 through 4.11, I modified one program into four separate versions, so I could run them simultaneously on one desktop. The smaller primes gave a lot more results to choose from after an over night run.

The primes used for the base field were taken from reference (Riesel, 2013). I used the table **Primes of the Form:** $h \cdot 2^n + 1$ and looked for field sizes as close to 160, 256, 384

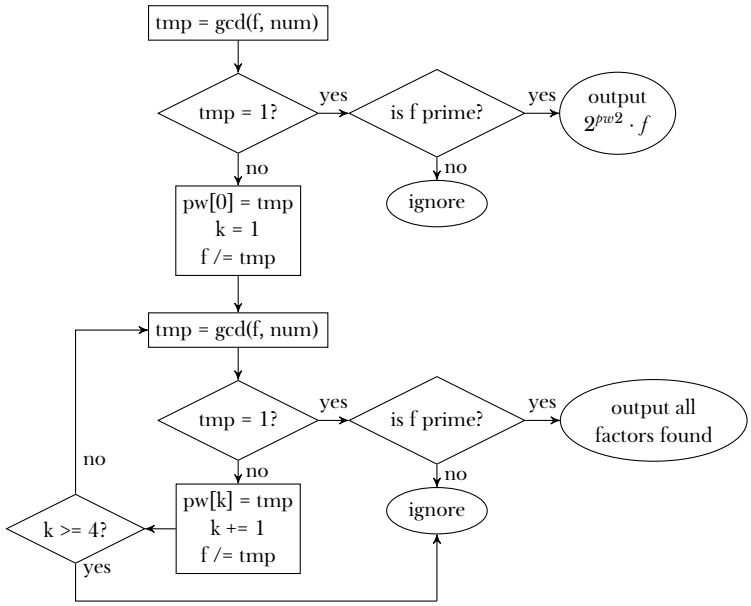

Figure 6.2 Accept or reject curve

and 512 with as small an h as I could get. Table 6.1 shows the values chosen. The average density of good curves is about the same for every field prime. If you find a prime that is more useful for faster base field operations most likely you will find just as many good curves as these choices provided.

Table 6.1 Primes for base field

field size	prime number
160:	$43 \cdot 2^{158} + 1$
256:	$43 \cdot 2^{252} + 1$
384	$23 \cdot 2^{381} + 1$
512	$113 \cdot 2^{509} + 1$

Exercise 6.2

In the curve of excercise 1, what is the value t from equation 6.2?

6.2.1 *Variables and initialization*

In this section the initial code to find good curves is presented.

Listing 6.1 shows the start of the program used to find 160 bit curves. The constant `z` is the first entry from table 6.1.

The `ellinit()` library function in libpari requires a vector with five values. For general elliptic curves this makes sense. For ordinary curves over large prime finite fields only two values are required.

Listing 6.1 Finding curves: initialize

```
int main()
{
  GEN y, E, f, z, a1, a3, a2, ell5, num, tmp;
  int k, pw2, pw[4];
  unsigned long a4coef, a6coef;
  PARI_sp av;

  PARI_init(1024*1024*1024, 5*1024*512);      | initialize PARI and finite field
  z = gp_read_str("0xac0000000000000000000000000000000000000001");
  y = ffgen(z, -1);  ← create FFELT GEN object
                                  hex value 43 · 2^158 + 1
  a1 = gen_0;
  a3 = gen_0;   | constant 0 for unused values
  a2 = gen_0;
  ell5 = zerovec( 5);
  gel(ell5, 1) = a1;
  gel(ell5, 2) = a2;        | initialize array to integer values
  gel(ell5, 3) = a3;
  a4coef = 1;
  a6coef = 0x01;   | inside loop variables

  num = muluu(3, 5);
  num = mului(7, num);
  num = mului(11, num);   | gcd constant num = 3 · 5 · 7 · 11
```

The three variables `a1, a2, a3` are assigned the constant value zero in `GEN` format. The other PARI constants available are 1 (`gen_1`) and 2 (`gen_2`). The `ell5` variable is set up to be a length five vector whose first three values are all zero. The names of the variables are there simply to remind me what those index positions stand for in the general elliptic curve equation.

Originally I had changed the `a4coef` as part of the sweep of curves. But this was pointless because the curves were isomorphic to all the others found with $a_4 = 1$. The last part of the initialize is to create the constant with the four small primes $3 \cdot 5 \cdot 7 \cdot 11$.

6.2.2 Main loop

In this section code for deciding to keep or reject a curve is described.

The search for good curves requires calling the `ellinit()` function and then getting the cardinality of the curve. I then want to find the factors of the cardinality to determine if there is a very large prime. The first factor to eliminate are powers of two. Using the `bittst()` function it is very simple to determine how many factors of two are in the cardinality. This is shown in listing 6.2.

The variable `av` marks the PARI stack. Every new variable in a loop increases the stack

use. To remove temporary stack variables the routine `gerepileall()` is called at the bottom of the loop. If you don't do this the stack will overflow and an attempted overnight run will only last an hour. More details can be found in the "User's Guide to PARI Library" (see appendix A).

Listing 6.2 Finding curves: top of main loop

```
while(a6coef < 0xfffff) ←— arbitrary limit, reduce for larger primes
{
  printf("%0lx  %0lx", a4coef, a6coef);      | monitor
  fflush(stdout);                            | progress
  av = avma;       ← mark top of PARI stack
  gel(ell5, 4) = stoi(a4coef);               | convert integers
  gel(ell5, 5) = stoi(a6coef);               | to GEN
  E = ellinit(ell5, y, 0);                   | and create curve

  f = ellcard(E, NULL); ← compute cardinality of curve
  k = bittest(f, 0);
  pw2 = 0;
  while(!k)                                  |
  {                                          | track and
    pw2++;                                   | remove all
    f = gdivexact(f, gen_2);                 | powers of 2
    k = bittest(f, 0);                       |
  }
```

The `stoi()` function converts a `long` to a PARI `GEN`. The variable `y` is already specified as a finite field using a prime as shown in listing 6.1. The result from `ellcard()` is a PARI integer. The loop on `k` removes the powers of two if the last bit is a 0 in the cardinality value `f`.

The first seven lines of Listing 6.3 shows how the gcd is used to test for only powers of two. If the gcd(f, num) is 1 and f is a prime then only powers of two can be factors of $\#E$. The section of code following the `else` clause then goes into removing the small factors to determine if the cardinality is acceptable or not.

Listing 6.3 Finding curves: small prime check

```
tmp = gcdii(f, num);  | if gcd(f, num)==1
if(isint1(tmp))       | then no small factors
{
  if(isprime(f))   ← if f is prime
    PARI_printf(" 2^%d * %Ps\n", pw2, f);  | save powers of 2 and prime
  else
    printf("\n");  | otherwise ignore this curve
}
else ← gcd(f, num) != 1
{
  pw[0] = itos(tmp); ← save first group small factors
```

```
    k = 1;
    f = gdivexact(f, tmp); ← remove first group small factors
    while((k < 4) && !isint1(tmp))
    {                                  limit num^3 max powers
      tmp = gcdii(f, num);   if gcd(f, num)==1
      if(isint1(tmp))        then no small factors
      {
        if(isprime(f))  ← if f is prime
        {
          printf(" 2^%d * %d ", pw2, pw[0]); ← powers of 2 and first group
          k--;
          while(k)                          output
          {                                 remaining
            printf("* %d  ", pw[k]);        group
            k--;                            powers
          }
          PARI_printf("* %Ps\n", f); ← large prime factor
        }
        else
          printf("\n"); otherwise ignore this curve
      }
      else ← gcd(f, num) != 1
      {
        pw[k] = itos(tmp);          save next group
        f = gdivexact(f, tmp);      and remove
        k++;                        from cardinality
      }
    }
    if(k >= 4)
      printf("\n");  ← hit limit so ignore curve
  }
  gerepileall( av, 1, &ell5); ← reduce stack size
  a6coef++;   ← go to next curve
}
printf("all done\n");
```

The variable `pw[]` counts the powers of each factor discovered from the `gcdii()` function. Suppose `f` has cofactor $3^2 \cdot 5^3$. When I take the gcd only one power of 3 and 5 will be common. The variable `pw[]` is four deep because more powers than four mean I should ignore this curve.

The line `f = gdivexact(f, tmp);` removes one set of small primes from the cardinality. When the `gcdii()` function goes to 1 there are no more small primes. At that point I can call the function `isprime()` to determine if the leftover value is a prime. If not, I can ignore this curve.

At the 160 bit level an overnight run on a 4 GHz processor found over 1000 curves with a large prime. Of those over 100 were prime cardinality. At the 512 bit level only eight worthwhile curves were found with three having prime cardinality. Table 6.2 lists the cardinality of the largest prime found for each program in hexadecimal notation. All

the parameters of each curve are in listings 4.8 through 4.11.

Table 6.2 Cardinality of best curves

160	0xac0000000000000000006543ba11adf8eb6345c77
256	0x2b0000000000000000000000000000002e7f521c85bba055a6e2161b956a47f69
384	0x2e0000000000000000000000000000000000000000000002275cc5f2f7fcc15352a2c9939 ➥00a851b3a75365a9ac54733
512	0xe2000000000000000000000000000000000000000000000000000000000007788830d ➥091dc57e3af7d7bbd15386ee9414602d88d1e6489cd056336922bbf4d

6.3 Bad curves

In this section the difference between good and bad cryptographic curves is described.

What a "good curve" is for cryptography will be a really "bad curve" for factoring large numbers. For cryptography, we require a large prime field for our points to float in. If there are many factors in the cardinality of the curve there are many combinations of groups each point can belong to. So the first aspect of a "good curve" for cryptography is that the cardinality have a large prime and small cofactor.

In chapter 13 I will get into the details on field extensions. A field extension takes a prime order field p to some power k, so the size of the field is p^k. The MOV attack on elliptic curve key sharing uses a field extension to map an elliptic curve to a small extension field which can be manipulated more easily. The attack changes an elliptic curve discrete log problem into an exponential discrete log problem. The former is exponential in the size of the key, the latter is subexponential. For descriptions of this and other methods of solving for the private key from the public key and base point see chapter 5 in (Blake *et al.*, 1999).

So even if we have a large prime in the cardinality, having a low field extension would turn it into a "bad curve". Finding the actual value of the field extension is challenging, but all we really care about is that it should be large.

The average field extension on random curves is approximately $\sqrt{p}$. Since $\sqrt{p}$ has more than 80 bits for even the smallest security level, that's classified as "big" for a field extension. All the curves found using the above program (listings 6.1 through 6.3) were found to have an extension greater than 256. They are immune to the MOV attack.

Other attacks on elliptic curves require a small cofactor. Since the curves chosen here have a cofactor of 1 those attacks can not be performed. Every point on the curve has the same order. While this means there are no shortcuts for computation, there are no hand holds for an adversary to attack with.

There are other parameters involved than just the curve. The application environment also is a factor. Is speed really important? If so, then the field prime chosen might create a side channel attack vector. The security of your system depends on thinking about issues other than just cryptography. So it might take a few overnight runs with many different

field primes to find the best curve for your situation. Since you only have to do it once, the time is well worth the effort.

> **Exercise 6.3**
>
> The cardinality of the curve in excercise 1 is 97 bits, so it has a possible security level of 46 bits. Is this a good or bad cryptographic curve?

6.4 Summary

- Ordinary elliptic curves over finite fields are defined by equation

$$E : y^2 = x^3 + a_4 x + a_6 \bmod p$$

  and $a_6 \neq 0$.

- The cardinality of an elliptic curve is

$$\#E = p + 1 - t$$

  where $|t| < 2\sqrt{p}$ and t is called the trace of Frobenius.

- PARI/gp is both a math calculator and library with an API for computing cardinality and factors of elliptic curves.
- For this book specific primes are chosen for several security levels. The form is $h \cdot 2^n + 1$.
- libpari requires specific initialization. Requesting gigabytes helps with very large primes.
- Finding good curves for cryptography demands a small cofactor and one large prime for cardinality. For ordinary curves demanding cofactor of 1 is possible and highly recommended for best security.
- Bad curves for cryptography have too low an embedding degree or too many small cofactors. For random ordinary curves both conditions are avoidable with careful searching.

Chapter Bibliography

Blake, I., Seroussi, G., & Smart, N. 1999. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press. 61, 70

Riesel, H. 2013. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser Boston. 65

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York. 63, 64

6.5 Answers to exercises

- 6.1) Using PARI/gp with F=ellinit([1, 97], 952898713027537551650783963 11), type F.group to find

```
%2 = [952898713027532801177859728 87, [952898713027532801177859728 87]...
```

  The cardinality of the curve is thus 952898713027532801177859728 87.

- 6.2) Rewriting equation 6.2 as $t = p + 1 - \#E$, PARI/gp gives

```
? 952898713027537551650783963 11 + 1 - 952898713027532801177859728 87
%10 = 475047292423425
```

- 6.3) This is a bad curve. Using the command factor(952898713027532801177859728 87) PARI/gp returns:

```
[          61 1]

[     5151169 1]

[ 9364883051 1]

[32382378793 1]
```

7 Description of finite field polynomial math

This chapter covers

- Essence of field extension is a polynomial
- Routines to create polynomials
- Addition routine for polynomials
- A debugging routine for polynomials

In this chapter I show a simple structure for polynomials and how to add them together. The code developed here is used through the rest of the book.

Understanding how pairings work on elliptic curves requires the use of field extensions. These are just polynomials with finite field coefficients. But they have all the properties of a finite field because there is a fixed number of elements and they can be added, multiplied, and inverted. The following six chapters are short, so they cover just one aspect of an operation over a field extension. The code to execute the mathematics takes up more text than the mathematical description. Hopefully it can be easily absorbed so that the code and math associated with elliptic curve pairing operations will then make more sense.

In this chapter I will cover a polynomial structure which is much simpler than those used in most mathematics packages. Typical use will be a fixed sized polynomial, so a general construction is not necessary. A few utility subroutines will be described along with the

addition and subtraction routines.

7.1 Field extension

In this section the general description of a field extension over a finite field is described. This concept is essential for the rest of the book.

A field extension means we extend a finite field by taking the prime defining the field to a power. So a field of p elements can become an extension of degree k when we have p^k elements. As an example suppose $p = 1187$ and $k = 3$. The base field has 1187 elements and the field extension has $1187^3 = 1672446203$ elements.

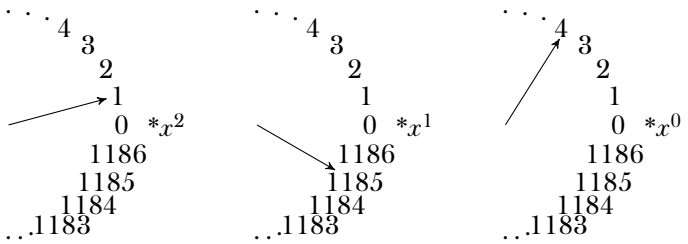

Figure 7.1 Finite field extension over prime numbers viewed as a clock ($p = 1187$, $k = 3$)

Figure 7.1 expands figure 2.2 to show the idea behind a field extension. For a general field extension, any value is allowed for the leading coefficient (modulo the field prime.) The polynomial from figure 7.1 is $x^2 - 2x + 4 \bmod 1187$. Counting backwards from 0 gives negative values, but we can always change them to be positive by adding p (in this case 1187).

To keep track of each element in the extension field we want an indexing method. To do that we choose an arbitrary symbol like t or x and construct a polynomial of powers of that variable. The general form looks like

$$a_{k-1}x^{k-1} + \cdots + a_1x + a_0 \tag{7.1}$$

In this form each coefficient is taken modulo p. In our numerical example each coefficient $\{a_0, a_1, a_2\}$ has 1187 possible values, so the total number of indexes is 1187^3.

We add two polynomials in a field extension by summing the coefficients of matching powers. Because the coefficients are modulo the field prime each coefficient stays in the range of the prime and there is no mixing between powers of the variable. The general form looks like

$$\begin{array}{r} a_{k-1}x^{k-1} \cdots + a_1x + a_0 \\ + \; b_{k-1}x^{k-1} \cdots + b_1x + b_0 \\ \hline c_{k-1}x^{k-1} \cdots + c_1x + c_0 \end{array}$$

Subtraction is identical. In fact, since everything is modulo a prime we end up with positive results in software.

This introduction is missing a lot of important details. I will save these details for chapter 8, so we can concentrate on how to use polynomials in code.

Exercise 7.1

What is the sum of $37x^3 + 96x^2 + 7x + 3$ mod 127 and $14x^2 + 83x + 124$ mod 127?

7.2 Polynomial setup

In this section polynomial support code is described. It is used in every subroutine through the rest of the book.

To use polynomials I create a simple structure as shown in listing 7.1. The value `MAXDEGREE` sets the maximum power of field extension I plan on dealing with. Because the structure is simple all polynomials have space for the same number of coefficients even if they are not used. For small embedded systems this is not efficient. The complexity involved with that efficiency would confuse the purpose of explaining how things work. A web search on "efficient representation of polynomials" will give ideas on sparse vectors and linked lists.

Listing 7.1 Polynomial structure

```
#define MAXDEGREE 32

typedef struct
{
  unsigned long deg;
  mpz_t coef[MAXDEGREE];
}POLY;
```

Because structure 7.1 is used in many places it is placed in a header file `poly.h`. All the routines in the file `poly.c` are also listed as prototypes in the header file. These routines include initialization, clearing, addition, subtraction and a few utilities which will be explained here. The header also includes prototypes for the routines which will be described in chapters 8 through 12.

The first routine in file `poly.c` initializes a structure. This is shown in listing 7.2. The index into the coefficient array matches the power of the variable for the polynomial. So `*.coef[4]` is the coefficient to x^4.

Listing 7.2 Polynomial initialization

```
void poly_init(POLY *p)
{
  int i;

  for(i=0; i<MAXDEGREE; i++)
    mpz_init(p->coef[i]);
  p->deg = 0;
}
```

create space for every coefficient

← constant term only

Creating a polynomial as a variable in a subroutine requires removing it before returning to avoid memory leaks. Listing 7.3 shows the simple routine that accomplishes this. I used the same syntax as GMP to be both consistent and lazy.

Listing 7.3 Polynomial clearing

```
void poly_clear(POLY *p)
{
  int i;

  for(i=0; i<MAXDEGREE; i++)
    mpz_clear(p->coef[i]);
}
```

remove every coefficient created

7.3 Polynomial addition

In this section I describe how two polynomials are added.

Adding two polynomials is simple. But even simple things get complicated when they are not quite the same. If we have two polynomials of different degree the larger one will have coefficients which are copied to the result. So the code in listing 7.4 first checks to see which input is larger. If they are equal it picks the first input to define the degree.

Listing 7.4 Polynomial add

```
void poly_add(POLY *c, POLY a, POLY b)
{
  int i, dc;
  POLY rslt;

  poly_init(&rslt);
  if(a.deg > b.deg)
  {
    rslt.deg = a.deg;
    for(dc=a.deg; dc>b.deg; dc--)
      mpz_set(rslt.coef[dc], a.coef[dc]);
  }
  else if(b.deg > a.deg)
  {
    rslt.deg = b.deg;
    for(dc=b.deg; dc>a.deg; dc--)
      mpz_set(rslt.coef[dc], b.coef[dc]);
  }
  else
  {
      dc = a.deg;
      rslt.deg = a.deg;
  }
  while(dc >= 0)
  {
```

a bigger than *b*
copy over higher *a* coefficients

b bigger than *a*
copy over higher *b* coefficients

same size
use *a* to define result

```
    madd(rslt.coef[dc], a.coef[dc], b.coef[dc]);   | add common
    dc--;                                          | powers
  }
  i = rslt.deg;
  while((i > 0) && (!mpz_cmp_ui(rslt.coef[i], 0)))
  {
    rslt.deg--;   | remove
    i--;          | leading zeros
  }
  for(i=0; i<=rslt.deg; i++)                       | copy result
    mpz_set(c->coef[i], rslt.coef[i]);             | to designated
  c->deg = rslt.deg;                               | storage
  poly_clear(&rslt);
}
```

All the coefficients with common powers are then summed modulo the field prime. If higher degree coefficients go to zero in this process, the degree of the resulting polynomial must be reduced. When I first tested this routine that check was not included, and I found some interesting bugs down the line. This is labeled "remove leading zeros" in listing 7.4.

The final step is to transfer the internal result to the specified place. At least I figured that one out ahead of time!

Once we have addition we can do subtraction by negation of the second argument and calling addition. This is shown in listing 7.5.

Listing 7.5 Polynomial subtract

```
void poly_sub(POLY *c, POLY a, POLY b)
{
  int i;
  POLY bneg;

  poly_init(&bneg);
  bneg.deg = b.deg;
  for(i=0; i<=b.deg; i++)
  {                                                | negate each
    mpz_init_set(bneg.coef[i], b.coef[i]);         | coefficient
    mneg(bneg.coef[i], bneg.coef[i]);              | one at a time
  }
  poly_add(c, a, bneg);   <- then add to get result
  poly_clear(&bneg);
}
```

7.4 Polynomial utilities

In this section useful low level common routines are shown. They are used in many subroutines in the rest of the book.

There are several simple routines that are useful for manipulating polynomials. These include copying, comparing, printing and creating random polynomials. Duplication of a

polynomial is common for routines that will manipulate inputs. The comparison routine can only check for equality. The concept of greater or lesser does not make sense when we are working modulo prime numbers in a cyclic field. Printing polynomials to the console is very useful for debugging as well as preserving work. Random polynomials are used for digital signatures and the square root algorithm.

The utility for copying polynomials is shown in listing 7.6. The assumption is that the place being copied to has already been initialized.

Listing 7.6 Polynomial copy

```
void poly_copy(POLY *a, POLY b)
{
  int i;

  a->deg = b.deg;     ← first copy degree
  for(i=0; i<=b.deg; i++)              | then each coefficient
    mpz_set(a->coef[i], b.coef[i]);    | one at a time
}
```

Comparing two polynomials is only useful to test if they are equal. The idea of "greater than" does not make a lot of sense because there are no negative numbers when we finish with the coefficients. The compare utility in listing 7.7 returns 1 if the inputs are equal and 0 if not.

Listing 7.7 Polynomial compare

```
int poly_cmp(POLY a, POLY b)
{
  int i;

  if(a.deg != b.deg) ← different degree then not equal
    return 0;
  for(i=a.deg; i>=0; i- -)                | if any coefficient different
    if(mpz_cmp(a.coef[i], b.coef[i]))     | then not equal
      return 0;
  return 1;
}
```

The third utility is useful for debugging. I modified this routine several times to either make it look pretty or be useful. This particular form is very useful for copying into PARI/gp. I found that comparing my calculation to PARI uncovered many problems. By writing out each coefficient using the `Mod()` form along with the power of x it corresponds with, I can use the result directly as input to `gp`.

Listing 7.8 shows the core print routine. For debugging multiple items this is useful.

Listing 7.8 Polynomial print routine

```
void poly_print(POLY a)
```

```
{
  int i;
  mpz_t prm;

  mget(prm);    ← modulus for PARI
  for(i=a.deg; i>0; i--)
  {                                    | don't print if
    if(mpz_cmp_ui(a.coef[i], 0))       | coefficient is zero
      gmp_printf("Mod(%Zd, %Zd)*x^%d + ", a.coef[i], prm, i);
  }
  gmp_printf("Mod(%Zd, %Zd) ", a.coef[0], prm); ← don't print x^0
  printf("\n");
  mpz_clear(prm);
}
```

Listing 7.9 adds a string input, so I can label the output and remind myself what I was trying to look at.

Listing 7.9 Polynomial print with string

```
void poly_printf(char *string, POLY a)
{
  printf("%s", string);
  poly_print(a);
  printf("\n");  ← add blank line for visibility
}
```

A final utility routine is the generation of random polynomials. This assumes an irreducible polynomial has been set, so the maximum degree is known, then it just creates random values for each coefficient. Listing 7.10 shows how easy this is to do.

Listing 7.10 Polynomial random value

```
void poly_rand(POLY *rnd)
{
  int i;

  rnd->deg = irrd.deg - 1;  ← max degree possible
  for(i=0; i<irrd.deg; i++)   | all coefficients
    mrand(rnd->coef[i]);      | are random
}
```

7.5 Summary

- A degree k extension of a finite field p has p^k elements.
- Polynomial structure includes integer degree and a fixed number of coefficients.
- Adding two polynomials of different degree results with the highest degree in output.
- Care must be taken with output degree if highest coefficient goes to zero.

- Random polynomials will be one degree less than the irreducible polynomial defining the field.

7.6 Answer to exercise

- 7.1) Modulo 127 we find:

$$\begin{array}{l} 37x^3 + 96x^2 + 7x + \quad 3 \\ \qquad\quad 14x^2 + 83x + 124 \\ \hline 37x^3 + 110x^2 + 90x \end{array}$$

8 Multiplication of polynomials explained

This chapter covers

- Irreducible polynomials
- How irreducible polynomials act like primes
- Multiplying two polynomials modulo an irreducible polynomial

In this chapter we'll learn what irreducible polynomials are and how they depend on the underlying prime number modulus. The fundamental takeaway for this chapter is the multiplication table derived from an irreducible polynomial which defines an extension field. This table will allow us to compute extension field algorithms efficiently. The code for this chapter is the core of all the routines in the rest of the book.

As I said in chapter 7 I left out some details about finite field extensions. The first detail is that the arbitrary symbol x or t is not actually arbitrary. It is the solution to an irreducible polynomial equation. In this chapter we dive into the detail of what makes a polynomial irreducible and how that is used like a prime number to create finite fields over polynomials. The coefficients are reduced modulo a field prime as shown in chapter 7, but the multiplication of two polynomials requires a modulus which is a polynomial. I will sometimes use the term prime polynomial instead of irreducible polynomial. The difference in terminology comes from the use of the polynomial as a modulus, where it is

like a prime versus its use as a factor where it is irreducible.

8.1 Defining irreducible polynomials

In this section I define what an irreducible polynomial is. Irreducible polynomials are fundamental to field extensions. They are also called prime polynomials because they act like prime numbers in a finite field. The concept is important for the rest of the book.

A reducible polynomial has multiple polynomial factors. In this section I want to explain what is meant by an irreducible polynomial. With a simple example using different prime numbers we'll see that the same equation can have different properties. The magic of field extensions is then shown by setting the irreducible polynomial equal to zero.

A general polynomial can have several factors. An irreducible polynomial has no reduction with smaller factors. A simple example of a factorable (or reducible) polynomial is

$$x^2 - 1 = (x + 1)(x - 1)$$

The polynomial $x^2 - 1$ has two factors. This formula is true no matter what the field prime is. Now let's take a look at a polynomial which has different factors with different field primes. The formula

$$x^2 + 13x + 1 \bmod 1187$$

has no factors modulo 1187. But the same formula

$$x^2 + 13x + 1 \bmod 43$$

factors into

$$(x + 25)(x + 31) \bmod 43$$

The formula $x^2+13x+1 \bmod 1187$ is irreducible but the same formula modulo 43 is reducible. So the choice of field prime also determines the choice of irreducible polynomial for creating a field extension.

In chapter 11, I describe finding good irreducible polynomials for efficient programming. For now, I will just assume we have an irreducible polynomial we can use with our chosen field prime, so we can proceed with using it as a modulus.

The usefulness of an irreducible polynomial comes from setting it equal to zero. In the above example we have

$$x^2 + 13x + 1 = 0 \bmod 1187$$

so

$$x^2 = -13x - 1.$$

Multiplying both sides by x we have

$$x^3 = -13x^2 - x.$$

Replacing x^2 with $-13x + 1$ gives

$$x^3 = -13(-13x - 1) - x = -170x - 13 = 1017x + 1174.$$

where the last step replaces a negative number with its positive modulo equivalent by adding the prime 1187. So no matter what power we raise x to we end up with $a \cdot x + b$ with a and b in the range {0..1186}. The total number of field elements will be 1187^2.

Technically, x is a root of the irreducible polynomial. But since it is over a prime field there is no integer value which satisfies the equation. The equation itself is the answer we want. The equation acts exactly like a prime number.

Exercise 8.1

$b = x^3 + x + 5$ mod 131 is irreducble. What is x^5 mod b?

8.2 Irreducible polynomial as modulus

In this section I show how an irreducible polynomial becomes a modulus for all other polynomials.

The general multiplication of polynomials will result in a highest power that is the sum of the highest powers of the factors. For an extension field of degree k, we only need $k - 1$ coefficients to create all p^k combinations as shown in chapter 7. In this section I show how we can use an irreducible polynomial as a modulus to maintain the p^k field size.

Assume we have a general irreducible polynomial we can write as

$$t^n + a_{n-1}t^{n-1} + a_{n-2}t^{n-2} + \cdots + a_1 t + a_0 \tag{8.1}$$

This is called a monic polynomial because the leading coefficient is 1. To use this as a modulus we would normally divide formula 8.1 into another polynomial to determine the remainder. The quotient would be thrown away similar to how we find the remainder of a number modulo a prime.

There are easier ways to find a modular result especially for multiplication. Setting equation 8.1 to zero gives

$$t^n = -a_{n-1}t^{n-1} - a_{n-2}t^{n-2} - \cdots - a_1 t - a_0 \tag{8.2}$$

Taking the quotient of all powers of t less than n with the irreducible polynomial will not change the remainder. But once we hit n and get larger we can use equation 8.2 to reduce the result back to a sum of powers less than n. If all our polynomials are reduced by the irreducible polynomial they will all have maximum degree $n - 1$. Multiplying two polynomials that are already modulo a prime polynomial results in polynomial with maximum degree of $2n - 2$ before reduction.

To see this take two polynomials $y_1 = a_k t^k + \cdots + a_0$ times $y_2 = b_k t^k + \cdots + b_0$. The largest possible term is $a_k b_k t^{2k}$. All the other terms will be of lower power in t. The process of multiplication only requires using a lookup table for all powers of t from 0 to $2n - 2$.

I am going to follow the brute force method described in section 3.1.2 of reference

(Cohen, 2000). Take the first arbitrary polynomial as

$$a = \sum_{i=0}^{r} a_i t^i \tag{8.3}$$

and the second arbitrary polynomial as

$$b = \sum_{j=0}^{s} b_j t^j. \tag{8.4}$$

The multiplication of 8.3 with 8.4 then gives

$$c = \sum_{k=0}^{r+s} c_k t^k \tag{8.5}$$

where each coefficient c_k is given by

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}. \tag{8.6}$$

The trick with formula 8.6 is to take $a_i = 0$ when $i > r$ and $b_j = 0$ when $j > s$.

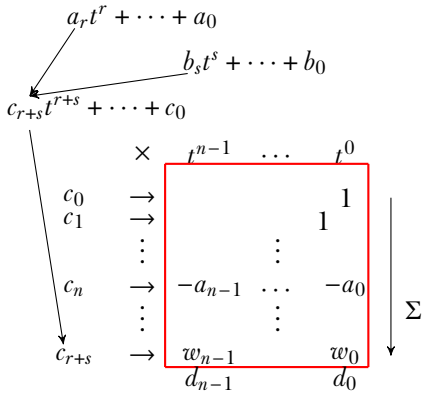

Figure 8.1 Multiplication modulo a prime polynomial. Polynomial a times polynomial b gives polynomial c - each coefficient of c is multiplied with the corresponding row in the expansion matrix, then summed to find the result modulo the prime polynomial.

The computation of equation 8.5 results in powers of t up to t^{r+s}. But we want to reduce this modulo our irreducible polynomial. The process is diagrammed in figure 8.1. We first compute each coefficient c_k with equation 8.6. This is schematically drawn at the top of figure 8.1. Those values then multiply the corresponding row k of a precomputed table of powers t^k depicted as the red box. All the columns are then summed with the result to find c modulo the irreducible polynomial (shown as d_k in figure 8.1).

8.3 Building the matrix

In this section I show how the irreducible polynomial is used to construct a matrix of coefficients that helps with multiplication. The subroutines based on these ideas are fundamental to the rest of the book.

Figure 8.1 shows the general idea of our lookup table. In this section I show how we compute each row, one at a time using the previous row along with the n^{th} row to find the full matrix.

The powers of t table is a matrix with each row being a power of t and each column a coefficient in the range of the field prime. Starting with row $t^0 = 1$ and multiplying each row by t, I create a matrix as shown in table 8.1.

The first column lists the power of t. At row n, I have equation 8.2. Row $n + 1$ shows the algebra of multiplying equation 8.2 by t which results in

$$t^{n+1} = -a_{n-1}t^n - a_{n-2}t^{n-1} \cdots - a_1t^2 - a_0t$$

Substitution of 8.2 in the first term gives the results in the table.

Table 8.1 Powers of t expanded modulo irreducible polynomial

	coefficient index				
power	$n-1$	$n-2$	$\cdots$	1	0
0	0	0	$\cdots$	0	1
1	0	0	$\cdots$	1	0
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
n	$-a_{n-1}$	$-a_{n-2}$	$\cdots$	$-a_1$	$-a_0$
$n+1$	$a_{n-1}^2 - a_{n-2}$	$a_{n-1}a_{n-2} - a_{n-3}$	$\cdots$	$a_{n-1}a_1 - a_0$	$a_{n-1}a_0$
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$2n-2$	c_{n-1}	c_{n-2}	$\cdots$	c_1	c_0

Each subsequent row is then the same process. Multiplying the previous row by t shifts all the coefficients over and the highest power coefficient is multiplied with row n. Then the two results are added together.

Exercise 8.2

Create the multiplication table for $x^4 + x^2 + 19 \bmod 131$

8.4 Multiplication code

In this section I describe the subroutines used to allow multiplication modulo a prime polynomial.

As described in section 8.2 there are two main steps to perform a multiply. Since we usually pick an irreducible polynomial as a fixed parameter the matrix of coefficients only has to be computed once. Once created, the table is used every time we multiply two polynomials modulo that prime polynomial. In this section I show the code for creating the table of coefficients and then the routine that uses the table to complete a multiply.

8.4.1 Creating the multiplication table

This section shows how the multiplication table is created.

The file `poly.c` contains the global variables `table` and `poly_degree` as shown here:

```
static mpz_t *table = NULL;
static long poly_degree;
```

← two-dimensional array

The indexing into the array `table` will be row = power of t with variable `i` and column = coefficient index with variable `j`. I called the routine `poly_mulprep` because it prepares the multiplication table.

Listing 8.1 shows the initialization portion of the routine. The input is an irreducible polynomial. There is no check here that the input is a prime polynomial (remember that prime and irreducible mean the same thing for our situation with polynomials). I will show how to do that in later chapters. The input polynomial can be anything, so I will force it to be monic (meaning the leading coefficient is 1) using a normalization routine. I will explain normalization in the miscellaneous section later.

Listing 8.1 Multiply table initialization

```
void poly_mulprep(POLY f)
{
  int i, j, tst;
  mpz_t tmp;
  POLY fnrml;

  poly_init(&fnrml);
  poly_copy(&fnrml, f);
  poly_degree = f.deg;
  if(table)
    free(table);
  table = (mpz_t*)malloc(sizeof(mpz_t)*poly_degree*poly_degree*2);
  for(i=0; i<2*poly_degree; i++)
    for(j=0; j<poly_degree; j++)
      mpz_init(table[poly_degree*i + j]);
  mpz_init(tmp);
```

multiple calls require clear before reuse

2n rows

n columns

$2n^2$ coefficients

coefficient automatically zero

I first check to see if the `table` was previously used and `free()` it if so. It is then set up to be a 2D array of `mpz_t` values of size $2n^2$. Clearly this is too big according to table 8.1 but not by too much.

The next step is easy, just filling in the first n rows of the matrix. Listing 8.2 shows that operation. Note that each diagonal element has a coefficient set to 1 with the same row and

column index.

Listing 8.2 Multiply table low powers of t

```
/*  set lowest degree terms to x^j  */

  for(i=0; i<poly_degree; i++)
    mpz_set_ui(table[poly_degree*i + i], 1); ← diagonal coefficent = 1
```

The last step of initial setup is to copy equation 8.2 to row n. This is shown in listing 8.3. If the input polynomial is monic the `poly_normal()` routine does nothing.

Listing 8.3 Multiply table row n

```
  poly_normal(&fnrml); ← force monic
  for(j=0; j<poly_degree; j++)
  {
    mpz_neg(tmp, fnrml.coef[j]);
    mpz_set(table[poly_degree*poly_degree + j], tmp); | negative of each coefficient in n^th row
  }
```

The meat of the preparation is filling in the bottom half of the table. The entry at row n (=`poly_degree`) is used along with the previous table entry. The outer loop does each row and the inner loop does each column of the matrix as shown in listing 8.4.

Listing 8.4 Multiply table bottom half

```
  for(i=poly_degree+1; i<2*poly_degree; i++) ← start at row n+1
  {
    for(j=1; j<poly_degree; j++)
    {
      mmul(tmp, table[poly_degree*poly_degree + j], table[i*poly_degree - 1]);
      madd(table[i*poly_degree + j], table[(i-1)*poly_degree + j - 1], tmp);
    }
    mmul(tmp, table[poly_degree*poly_degree], table[i*poly_degree - 1]);
    mpz_set(table[i*poly_degree], tmp);
  }
  mpz_clear(tmp);
  poly_clear(&fnrml);
```

add x^n entry to rotated coefficients

n^{th} row this column

highest coefficient previous row

t^0 coefficient special case

As seen in table 8.1 the last column of each row does not have the final term of all previous columns. That is why the `j = 0` term is done separately.

8.4.2 Polynomial multiply

This section shows how the multiplication table is used to compute the product of two polynomials modulo an irreducible polynomial.

Once the table has been created computing the multiplication of two polynomials mod-

ulo the prime polynomial is a matter of bookkeeping. An easy way to ensure the rule of equation 8.6 is to create a list of coefficients with twice the maximum possible length and zero out the coefficients beyond the size of the polynomial. Since initialization does this automatically I just create coefficient vector arrays which are sized to be the sum of each input degree. The initialization is shown in listing 8.5.

Listing 8.5 Multiply initialization

```
void poly_mul(POLY *rslt, POLY a, POLY b)
{
  int i, j, m, n;
  mpz_t coef[2*MAXDEGREE], acf[2*MAXDEGREE], bcf[2*MAXDEGREE];
  mpz_t tmp;

  m = a.deg;
  n = b.deg;
  for(i=0; i<=n+m; i++)
    mpz_inits(coef[i], acf[i], bcf[i], NULL);
  for(i=0; i<=m; i++)
    mpz_set(acf[i], a.coef[i]);
  for(i=0; i<=n; i++)
    mpz_set(bcf[i], b.coef[i]);
  mpz_init(tmp);
```

space for maximum possible degree

initialize space

actual space for maximum degree

copy over coefficients for each polynomial

The next step is to compute the coefficients of equation 8.6. This is shown in listing 8.6. The inner loop only goes from 0 to i, so we don't hit negative indexing. A lot of multiplies are zeros, so this is a point where more code can optimize fewer operations.

Listing 8.6 Multiply initial coefficients

```
  for(i=0; i<=n+m; i++)
  {
    for(j=0; j<=i; j++)
    {
      mmul(tmp, acf[j], bcf[i - j]);
      madd(coef[i], coef[i], tmp);
    }
  }
```

for each possible coefficient

compute equation 8.6

Once all the double length coefficients are computed I use the lookup table to reduce them down to the size of $n - 1$. If the sum of the degrees of the two input polynomials is less than the degree of the prime polynomial the result is simply transferred to the output. Otherwise, the result degree is set to $n - 1$ and each result coefficient less than n is copied to the matching output coefficient. The higher level coefficients are multiplied times the row in the mulprep table of each power above n, and this is added to the output. The code is shown in listing 8.7.

Listing 8.7 Multiply table reduction

```
combine upper powers
with lower using table
    if(n+m < poly_degree)
    {
      rslt->deg = n+m;                          final degree
      for(i=0; i<=n+m; i++)                     less than
        mpz_set(rslt->coef[i], coef[i]);        prime polynomial
    }
    else
    {
      rslt->deg = poly_degree - 1;  <- maximum possible degree
      for(i=0; i<poly_degree; i++)              lower coefficients
        mpz_set(rslt->coef[i], coef[i]);        do not change
      for(i=poly_degree; i<=n+m; i++) <- for each degree n and above
      {
        for(j=0; j<poly_degree; j++) <- for each coefficient
        {
          mmul(tmp, coef[i], table[i*poly_degree + j]);   multiply row by coefficient
          madd(rslt->coef[j], rslt->coef[j], tmp);        and add to result
        }
      }
    }
```

The final step is to check that none of the highest level coefficients went to zero. If they do, the degree of the polynomial result must be reduced. This chunk of code was discovered the hard way when really weird bugs showed up having high degree polynomials with zero coefficients. The code is shown in listing 8.8.

Listing 8.8 Multiply check maximum degree

```
  while((mpz_cmp_ui(rslt->coef[rslt->deg], 0) <= 0) && (rslt->deg > 0))
    rslt->deg--;                          most significant
  for(i=0; i<=n+m; i++)                   coefficient is zero?
    mpz_clears(coef[i], acf[i], bcf[i], NULL);   and degree is still positive
  mpz_clear(tmp);      clean up stack
}
```

8.5 Miscellaneous multiply routines

This section describes routines that are rarely used in the rest of the book. The normalization routine is used in chapter 12. The debug routine is presented as an example of how to find problems in coding.

While the use of a monic polynomial to set up the irreducible polynomial means normalization is not required, the ability to normalize polynomials will come in handy later. I also want to include a debug routine which I found exceptionally useful. The multiply table has a specific form which is easy to check when printed out. By using small numbers I can verify using a hand calculator or PARI/gp that the code is behaving properly.

The normalization routine is shown in listing 8.9. The variable `tst` compares the indexed coefficient to 1. A result less than zero implies the indexed coefficient is zero, so the next coefficient is checked. When `tst == 0` the indexed coefficient is 1 which implies the polynomial is already monic. When `tst > 0` these two tests fail and the inverse of the indexed coefficient is computed, and the loop is terminated.

Listing 8.9 Normalization routine

```
void poly_normal(POLY *a)
{
  int i, tst;
  mpz_t c;

  mpz_init(c);
  for(i=a->deg; i>=0; i--)
  {
    tst = mpz_cmp_ui(a->coef[i], 1);  ← is leading coefficient == 1?
    if(tst < 0)
      continue;  ← assumes coefficient must be zero
    if(!tst)     ← leading coefficient = 1 nothing to do
      return;
    minv(c, a->coef[i]);  inverse leading coefficient
    break;                use on all other coefficients
  }
  if(i < 0) return;     ← all zeros!
  while(i >= 0)
  {
    mmul(a->coef[i], a->coef[i], c); ← inverse times all coefficients
    i--;
  }
}
```

The inverse of the leading coefficient is then multiplied by all the coefficients. This forces the leading coefficient to 1 (this is not efficient, obviously) and adjusts all the other coefficients accordingly.

The final routine is a simple printing function to look at the table as it was generated. Since the table is not in polynomial format but just a matrix of coefficients all with the same degree, this is a special routine. Listing 8.10 shows the listing. Once debugged it was commented out using `#if`.

Listing 8.10 Debug table routine

```
#ifdef DEBUG
void poly_debug(int n)
{
  int i, j;

  for(i=0; i<2*n; i++)
  {
```

```
    for(j=n-1; j>=0; j--)                          | each row has
      gmp_printf("%Zd  ", table[i*n + j]);         | all coefficients
    printf("\n");
  }
}
#endif
```

Included in the repository is a program called test_mod.c which includes a degree 4 polynomial. The initialization of the polynomial is shown in listing 8.11.

Listing 8.11 Test of polynomial routines

```
C.deg = 4;
mpz_set_ui(C.coef[4], 1);
mpz_set_ui(C.coef[3], 2);          | irreducible polynomial is
mpz_set_ui(C.coef[2], 1);          | $x^4 + 2x^3 + x^2 + 3x + 5$
mpz_set_ui(C.coef[1], 3);          | mod 7
mpz_set_ui(C.coef[0], 5);
poly_mulprep(C);
```

The debug routine outputs the table shown in listing 8.12. This is actually the same data marked off in the box of figure 8.1 and the columns to the right of "power" in table 8.1.

Listing 8.12 Mulprep debug output

```
 0   0   0   1
 0   0   1   0
 0   1   0   0
 1   0   0   0
-2  -1  -3  -5      | $t^4$ row
 3   6   1   3
 0   5   1   6
 5   1   6   0
```

As you can see the first four lines are the 1's on the diagonal, then we have the negative of each coefficient on the t^4 row followed by the shifted and added rows for the last three row entries. The middle row is negative because the modulus was not applied. In the last 3 rows, all calculations are done modulo 7, so they are always positive.

8.6 Summary

- An irreducible polynomial over a finite field is defined as a polynomial having no other polynomial factors.
- The variable defining an irreducible polynomial is a root of the polynomial when set equal to zero.
- A matrix of coefficients with k columns and 2k rows allows the multiplication of any

two polynomials modulo an irreducible polynomial. This lookup table is only computed once for an irreducible polynomial.

- A degree n polynomial multiplied by a degree m polynomial has intermediate result of degree $n + m$. The degrees higher than k are reduced by multiplying the coefficient at that power by all the coefficients in the corresponding row of the matrix and summed to the final result which has at most k coefficients.
- A monic polynomial has a leading coefficient of 1. All irreducible polynomials used for field extensions are monic.
- Normalization of a polynomial inverts the leading coefficient and multiplies that with all coefficients leaving a monic polynomial.
- The term prime polynomial means the same as the term irreducible polynomial because working with a polynomial as a modulus is the same concept as working with a prime number as a modulus for integers.

Chapter Bibliography

Cohen, Henri. 2000. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag. 86

8.7 Answers to exercises

- 8.1) $x^3 = -x - 5$ so $x^4 = -x^2 - 5x$ and $x^5 = -x^3 - 5x^2$. Putting x^3 back in we get $x^5 = -5x^2 + x + 5 = 126x^2 + x + 5$.
- 8.2) The first four rows are simple: $1 \rightarrow x \rightarrow x^2 \rightarrow x^3$. Row 5 is $x^4 = -x^2 - 19$ and row 6 is just a left shift $x^5 = -x^3 - 19x$. The last row requires the use of row 5 because $x^6 = -x^4 - 19x^2$. This becomes $x^6 = x^2 + 19 - 19x^2 = -18x^2 + 19$. So the full table of coefficients is

```
    3  2  1  0
0   0  0  0  1
1   0  0  1  0
2   0  1  0  0
3   1  0  0  0
4   0 -1  0 -19
5  -1  0 -19 0
6   0 -18 0  19
```

9 Computing powers of polynomials

This chapter covers

- Exponentiation by expansion of an integer
- A square and multiply algorithm to compute powers of polynomials
- Examples for arbitrary powers of polynomials
- Examples for field prime powers of polynomials

In this chapter we use the code from chapters 7 and 8 to compute exponentials of polynomials modulo a prime polynomial. These routines are important for computing elliptic curve point pairings that underlie the routines shown in chapters 18 and 19.

Now that we know how to multiply polynomials modulo a prime polynomial, we can compute powers of polynomials. We need this ability to find irreducible polynomials and to find pairing friendly curves. In chapter 8 we found that powers of a variable modulo the irreducible polynomial is limited to one less than the degree of the irreducible polynomial. Similar to how we used the double and add method to compute multiplication of a point on an elliptic curve, we are going to use the square and multiply method to compute powers of a polynomial modulo a prime polynomial. This is exponentially faster than the method used in chapter 8.

The next interesting step after computing a general power is to take x^p modulo the

irreducible polynomial where p is the field prime. This will eventually connect back to the field extension when we compute x^{p^j}.

9.1 Using square and multiply to rapidly compute powers

In this section I present an algorithm to rapidly compute high powers of polynomials modulo a prime polynomial.

Generating large powers of x using the method of chapter 8 going one power at a time is exceptionally time-consuming. Especially if the power is a 160 bit number. We can more easily get there by expanding the exponent in powers of two. Each power of two is a squaring operation. We write this as

$$x^k = x^{k_0+2(k_1+2(k_2+\cdots+2k_j))}$$

where k_j is the most significant bit of k and k_0 is the least significant bit.

Figure 9.1 shows the full x^k power algorithm. The inputs are polynomial x and power k. The result is $r = x^k$ modulo the irreducible polynomial.

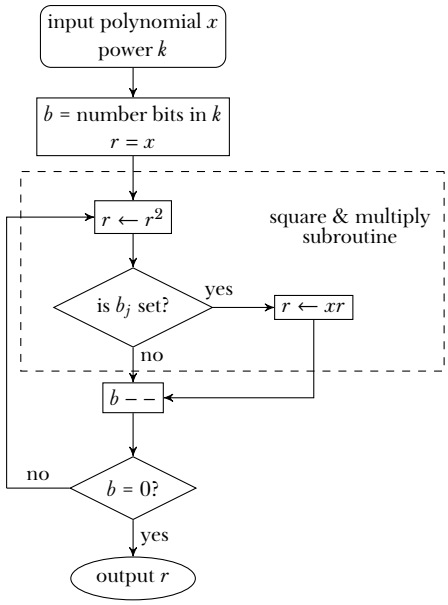

Figure 9.1 Polynomial square and multiply algorithm which will be used in chapter 12 to compute square roots

The most significant bit is always set, so we start with $x^1 = x^{k_j}$ and then square to get x^{2k_j}. Multiplying by $x^{k_{j-1}}$ gives $x^{2k_j+k_{j-1}}$. If bit k_{j-1} is 0, then that is simply multiplying by 1 which means we skip that step.

Proceeding this way through the entire integer k requires j squarings and Hamming weight multiplies (minus 1). Even if the variable x is a polynomial in t the same process

applies. Each step is done modulo the irreducible polynomial using the multiply routine from chapter 8.

In chapter 13 and beyond I will use a tiny example to illustrate some of the elliptic curve pairing properties. The irreducible polynomial in that example is x^2+x+3 mod 43. Let's see what the square and multiply routine gives when we take x^5 modulo x^2+x+3.

The exponent is 5 which in binary is 101. The most significant bit always being set (= 1), we start with x. We then square this to get x^2. Now we have to replace x^2 with $-x-3$. The bit in the exponent is clear (= 0), so we do not multiply by x this round.

$-x-3$ squared is the same as $(x+3)^2 = x^2+6x+9 = -x-3+6x+9 = 5x+6$. The last bit is set, so we multiply by x to get $5x^2+6x = 5(-x-3)+6x = x-15 = x+28$. Thus, we have x^5 modulo x^2+x+3 is $x+28$ (whose coefficients are modulo 43). The lookup table makes this process far more efficient, especially when we get to very large coefficients.

Exercise 9.1

Using the irreducible polynomial $c = x^4 + x^2 + 19$ mod 131 and the table found from exercise 8.2, find x^{17} mod c. Hint: be careful when squaring.

9.2 Polynomial powers code for general exponents

In this section I implement the code for computing powers of polynomials.

As shown in figure 9.1, there is a single stage square and multiply subroutine within dotted lines. This will be useful in a later routine for field prime exponents. In this section I explain how to compute powers of polynomials modulo an irreducible polynomial with an arbitrary exponent.

I break up the power routine into two simple subroutines

- single stage square and multiply, and
- general polynomial power function.

Shown in listing 9.1 is the inner squaring routine. This takes an input called `flag` which is the next bit in the integer power. If the flag is clear the output is simply x^2. If the flag is set the input a is used to compute ax^2 for the result.

Listing 9.1 Square and multiply

```
void poly_sqm(POLY *x2, POLY x, POLY a, int flag)
{
  POLY tmp;

  poly_init(&tmp);
  poly_mul(&tmp, x, x);   ← compute x^2
  if(flag)
    poly_mul(x2, tmp, a);   ← output is ax^2 with flag set
  else
    poly_copy(x2, tmp); ← output is x^2 with flag clear
  poly_clear(&tmp);
```

```
}
```

The routine `poly_mul()` assumes the irreducible polynomial table has already been initialized. So the operations in listing 9.1 are automatically done modulo the chosen prime polynomial.

The next routine implements the full exponentiation process. The input polynomial `g` is taken to the power `k`. Listing 9.2 shows how the variable `bitcnt` acquires the number of bits in the exponent. As with the double and add routine this is decremented by two for the same reason: the most significant bit is already taken care of and the base of counting is 1.

Listing 9.2 Polynomial exponentiation

```
void poly_pow(POLY *h, POLY g, mpz_t k)
{
  int bitcnt, bit;
  POLY a;

  poly_init(&a);
  poly_copy(&a, g); ← start with g^1
  bitcnt = mpz_sizeinbase(k, 2) - 2;  ← number of bits left to do
  while(bitcnt >= 0)
  {
    bit = mpz_tstbit(k, bitcnt);
    poly_sqm(&a, a, g, bit);          square at this position
    bitcnt--;                         and multiply if
  }                                   bit set
  poly_copy(h, a);  ← allow operation in place
  poly_clear(&a);
}
```

For each bit in the field prime the routine calls the square and multiply routine with the result and initial input, overwriting the result. The end of the loop happens when `bitcount == 0` and the last bit determines if there is a final multiply or not.

9.3 Explicit polynomial example

In this section I give an example of what to expect from powers of polynomials modulo a prime polynomial.

The whole point of cryptographic security is to work with very large numbers. However, to see what is going on it is a lot easier to work with very small numbers. The number 43 is a nice small prime and the irreducible polynomial $x^2 + x + 3$ mod 43 has $43^2 = 1849$ elements. In this section I use this irreducible polynomial to examine what happens when values are taken to a power modulo $x^2 + x + 3$. This tiny example is used in many places throughout the book.

Listing 9.3 shows a simple test program to exercise routine `poly_pow()`. The random number selection is always the same because the state of the program is the same every

time, but for this it doesn't matter.

Listing 9.3 Example program for `poly_pow()`

```
#include "poly.h"

#define PRIME 43

int main(int argc, char *argv[])
{
  POLY r, tst, pow;
  mpz_t n, pk2;
  int ck;

  if(argc < 2)                    when I forget how to use program
  {
    printf("Use: ./poly_exp_test <exponent>\n");
    exit(-1);
  }
  mpz_init_set_ui(n, PRIME);      setup field prime
  minit(n);
  mpz_init_set_str(pk2, argv[1], 10);
  gmp_printf("pk2= %Zd\n", pk2);
  poly_init(&r);
  r.deg = 2;                      irreducible
  mpz_set_ui(r.coef[0], 3);       polynomial
  mpz_set_ui(r.coef[1], 1);       x^2 + x + 3
  mpz_set_ui(r.coef[2], 1);
  poly_mulprep(r);  ← setup polynomial multiply table
  poly_printf("r = ", r);
  poly_init(&tst);
  poly_init(&pow);
  tst.deg = 1;
  mrand(tst.coef[1]);             random
  mrand(tst.coef[0]);             coefficients
  poly_pow(&pow, tst, pk2);
  poly_printf("taking ", tst);                use PARI/gp
  printf("to power %s\n", argv[1]);          to check code
  poly_printf("gives ", pow);
}
```

Given the input power 25 the output is

```
pk2= 25
r = Mod(1, 43)*x^2 + Mod(1, 43)*x^1 + Mod(3, 43)

taking Mod(11, 43)*x^1 + Mod(3, 43)

to power 25
gives Mod(3, 43)*x^1 + Mod(26, 43)
```

Now let's see what happens when the input power is 1848:

```
pk2= 1848
r = Mod(1, 43)*x^2 + Mod(1, 43)*x^1 + Mod(3, 43)

taking Mod(11, 43)*x^1 + Mod(3, 43)

to power 1848
gives Mod(1, 43)
```

The result is 1! This is exactly what we expect from Fermat's Little Theorem $a^{p^2-1} = 1$. In this case the total number of elements is $43^2 = 1849$ so $p^2 - 1 = 1848$. In a field extension we operate with powers of the field prime.

9.4 Powers of field prime

In this section I give a routine to compute polynomials to powers of the field prime. This is used in chapter 11 to find irreducible polynomials.

Taking a polynomial to powers of the field prime means we have multiple levels of exponents. The formula is

$$x^{p^j} = x^{\overbrace{p \cdot p \cdots p}^{j}}.$$

This is a very useful function all by itself which is shown in listing 9.4. The only difference between the general form in listing 9.2 and the special form in listing 9.4 is no exponent for input because the field prime is the exponent.

Listing 9.4 Polynomial to field prime power

```
void poly_xp(POLY *xp, POLY x)
{
  int i, bitcnt, bit;
  mpz_t prm;

  mget(prm);                          <- power is field prime
  bitcnt = mpz_sizeinbase(prm, 2) - 2; <- number of bits left to do
  poly_copy(xp, x);                   <- start with x^1
  while(bitcnt >= 0)
  {
    bit = mpz_tstbit(prm, bitcnt);    | square and
    poly_sqm(xp, *xp, x, bit);        | multiply each
    bitcnt--;                         | bit position
  }
  mpz_clear(prm);
}
```

By sending the previous output back into the routine again we obtain a way to compute

$$(x^{p^m})^p = x^{p^{m+1}}.$$

This will be useful in later routines where we are looking for irreducible polynomials.

There is one more useful routine shown in listing 9.5. This takes a polynomial to the power of $(p-1)/2$. This is a miscellaneous routine that will be useful when hunting down

elliptic curves for pairings. Since the field prime is always odd, $p - 1$ is always even so the call to `mpz_divexact()` always works.

Listing 9.5 Polynomial to half field prime power

```
void gpow_p2(POLY *h, POLY g)
{
  mpz_t prm;

  mget(prm);
  mpz_sub_ui(prm, prm, 1);
  mpz_divexact_ui(prm, prm, 2);
  poly_pow(h, g, prm);
  mpz_clear(prm);
}
```

compute $(p - 1)/2$

return $g^{(p-1)/2}$

9.5 Summary

- The square and multiply algorithm computes powers of x modulo a prime polynomial. The variable x can also be a polynomial and the same algorithm applies. The square and multiply algorithm is exponentially faster than a straightforward multiplication algorithm.
- Exponentiation uses the square and multiply routine for every bit in a power. Every bit forces a squaring operation, only set bits force a multiply operation.
- Fermat's Little Theorem applies to field extensions. For prime polynomial f of degree k

$$x^{p^k - 1} \cong 1 \bmod f.$$

  This is used to find square roots and irreducible polynomials.
- Computing x^{p^i} is an important subroutine for finding irreducible polynomials.

9.6 Answer to exercise

- 9.1) 17 in binary is 10001. Starting with x^1 we square to get x^2. The second bit (from the left) is clear, so we just square again to get $x^4 = -x^2 - 19$. The third bit is also clear so there is no multiply by x. Now life gets interesting because we square again to get $x^8 = x^4 + 38x^2 + 99$. Substitution for x^4 leaves $37x^2 + 80$. We square again to find $x^{16} = 59x^4 + 25x^2 + 112$. Multiplying x^4 by 59 amounts to subtraction of $59x^2 + 73$ from $25x^2 + 112$. The result is $-34x^2 + 39$. The last bit is set so we multiply by x to get the final result: $x^{17} = -34x^3 + 39x = 97x^3 + 39x \bmod 131$.

10 Description of polynomial division using Euclid's algorithm

This chapter covers

- Quotient and remainder from Euclid's algorithm
- Greatest common divisor of polynomials
- Inversion modulo an irreducible polynomial

In this chapter I cover polynomial division which is required to compute the point addition algorithm for points on a field extension elliptic curve.

To compute elliptic curve formulas over finite fields consisting of polynomials, we started in chapters 7 through 9 working with polynomials performing addition and multiplication. In this chapter we are going to dive into division so we can compute slopes of lines in an extension field. Working with an irreducible polynomial is similar to working with a prime number in terms of fields. Inversion modulo an irreducible polynomial is similar to inversion modulo a prime number. In this chapter I'll cover Euclid's division algorithm applied to polynomials. Using that algorithm I'll then look at the greatest common divisor function. The greatest common divisor (gcd) of polynomials will be used to help us find irreducible polynomials in chapter 11.

The last algorithm of this chapter covers inversion modulo a prime polynomial. Inver-

sion is really the extended Euclidean algorithm, but we don't need the quotient portion. I then show that division of two polynomials modulo an irreducible polynomial is inversion followed by multiplication.

10.1 Euclid's algorithm and gcd

In this section I describe Euclid's algorithm applied to polynomials. This implementation of Euclid's algorithm gives a quotient and remainder for a general division application. To find slopes on an elliptic curve over an extension field requires an inverse modulo an irreducible polynomial. In computing inverses modulo a prime polynomial the denominator is an irreducible polynomial, so there will always be a remainder. For the greatest common divisor (gcd) function, only the remainder will be useful. The gcd function will be used to find irreducible polynomials. I give an example of Euclid's division algorithm using very simple polynomials so each step can be seen.

The basic idea behind division using Euclid's algorithm is to find the quotient q and remainder r from the division of a/b. I'm going to use the method 3.1.1 in reference (Cohen, 2000). In the following description $lc()$ means leading coefficient and $deg()$ means degree of.

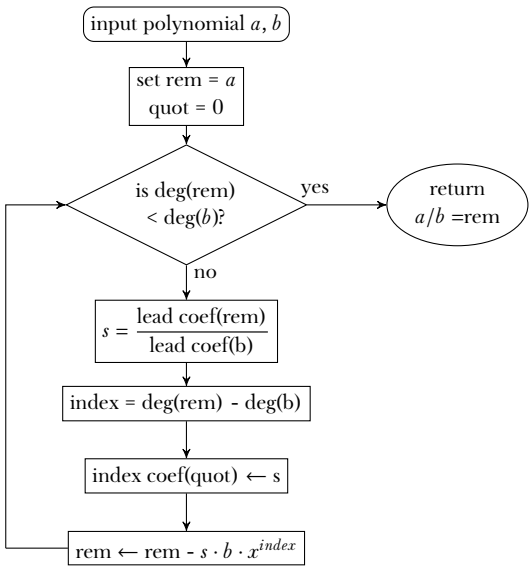

Figure 10.1 Polynomial Euclid's algorithm which is the guts of inversion in elliptic curve pairing algorithms

Figure 10.1 describes Euclid's algorithm for polynomials. The remainder is set to the numerator (a) and the degree of the remainder is compared with the degree of the denominator (b). If the remainder is lower degree than b the routine is done because the remainder is what is left after division. The process then goes one coefficient at a time which is similar to how division of polynomials is done by hand. After computing the di-

vision of the leading coefficients the correct index position of the quotient is set and the remainder is reduced by s times the denominator b.

In more mathematical detail we start by initializing the remainder and quotient:

$$r \longleftarrow a, q \longleftarrow 0 \tag{10.1}$$

Then loop while $deg(r) >= deg(b)$

$$\begin{aligned} s &\longleftarrow \frac{lc(r)}{lc(b)} \\ q[deg(r) - deg(b)] &\longleftarrow s \\ r &\longleftarrow r - s \cdot b \cdot x^{deg(r)-deg(b)} \end{aligned} \tag{10.2}$$

Line 10.1 initializes the remainder to the numerator and the quotient to zero. The first two lines in 10.2 compute the leading coefficient of the quotient for the leading degree in the quotient result. The last line in 10.2 reduces the degree of r. This loop continues until the degree of r is less than the degree of b.

The second line in 10.2 places the value of s in the variable q at index deg(r) - deg(b). This makes sense because the leading coefficient of r is at index deg(r) and the leading coefficient of b is at index deg(b) so the result of their division must be at the difference of the degrees.

As a concrete example of Euclid's algorithm, let's look at input polynomials

$$a = x^3 + 2x^2 + 3x + 1$$

and

$$b = 5x^3 + x + 6$$

with all coefficients modulo 7. Initialize

$$r = a = x^3 + 2x^2 + 3x + 1$$

and

$$q = 0$$

Then we have the value of s is

$$s = \frac{1}{5} \bmod 7 = 3$$

(because $5 \times 3 = 15 = 1 \bmod 7$). Since the $deg(r) = deg(b)$ The value of q is

$$q = s = 3$$

We then compute

$$r \leftarrow r - s \cdot b = x^3 + 2x^2 + 3x + 1 - 3(5x^3 + x + 6) = 2x^2 + 4$$

At this point the algorithm halts because the degree of r is less than the degree of b. This example of

$$\frac{x^3 + 2x^2 + 3x + 1}{5x^3 + x + 6} = 3 + \frac{2x^2 + 4}{5x^3 + x + 6}$$

is from the program test_mod.c used to test the subroutines.

Exercise 10.1

Find quotent and remainder from $\frac{13x^7+14x^3+19x+6}{x^5-x^2+17}$ mod 43.

The greatest common divisor (gcd) is useful for finding factors of polynomials. Using Euclid's division algorithm finding the gcd between two polynomials is very easy. I modified the routine 3.2.1 of (Cohen, 2000) to include the following initialization.

$$\begin{aligned} &\text{if } a = 0 \text{ return } b \\ &\text{if } b = 0 \text{ return } a \\ a_w \longleftarrow\ &\text{highest degree polynomial } (a, b) \\ b_w \longleftarrow\ &\text{smallest degree polynomial } (a, b) \end{aligned} \tag{10.3}$$

If one of the arguments is zero, the greatest common divisor of both arguments is the other argument. If both arguments are zero, the routine returns zero by definition. The reason for this is that everything divides zero. That is

$$\frac{0}{b} = 0.$$

My first reaction to this was a headache. But really it's "obvious" because 0 is smaller than every possible number and any number not 0 can divide 0, so the greatest divisor is the non-zero number.

The main loop of the gcd routine is then

$$\begin{aligned} &\text{while } deg(b_w) > 0 \\ &\qquad r \longleftarrow a_w / b_w \\ &\qquad a_w \longleftarrow b_w \\ &\qquad b_w \longleftarrow r \end{aligned} \tag{10.4}$$

The first line in 10.4 is the remainder from Euclidean division. The assumption is that a_w has larger degree than b_w otherwise the remainder degree is negative which would terminate the loop on the first step. The following two steps keep the reduction process going.

Exercise 10.2

What is the GCD of $(x+14)(x^2+22x+8)$ mod 23 and $(x+4)(x+19)(x^2+22x+8)$ mod 23?

10.2 Inversion and division

In this section I describe Euclid's extended algorithm applied to polynomials. These algorithms are used to compute point addition on field extension elliptic curves.

As shown in chapter 3, the formulas for adding points over an elliptic curve requires inversion modulo a field prime. The same formula applies to extension fields. There are more complex algorithms involved with pairings of points on extension field elliptic curves which also use inversion and division of polynomials modulo a prime polynomial.

Inversion modulo an irreducible polynomial uses the same steps as the extended Euclidean algorithm. In this section I go over these steps and then use a very simple example to show how each step appears for a real computation. Elliptic curve algorithms use division, so I also describe how a division subroutine is created from the inversion routine.

Inversion of polynomials is done with respect to a given irreducible polynomial. To compute a polynomial inverse I use most of the extended Euclidean algorithm described in section 3.2 of reference (von zur Gathen & Gerhard, 1999). The initialization step is

$$\begin{aligned} v &\longleftarrow irrd \\ u &\longleftarrow b \\ r &\longleftarrow 0 \\ w &\longleftarrow 0 \\ q &\longleftarrow 0 \\ t &\longleftarrow 0 \\ y_0 &\longleftarrow lc(b) \end{aligned} \tag{10.5}$$

where $irrd$ is the irreducible polynomial of the field, y_0 means coefficient of t^0 in polynomial y (the constant coefficient) and $lc(b)$ means leading coefficient of input polynomial b.

We then perform the loop

$$\begin{aligned} (q, r) &= v/u \\ \rho &= 1/lc(r) \\ r &\longleftarrow \text{normal}(r) \\ t &\longleftarrow \rho(w - qy) \\ w &\longleftarrow y \\ y &\longleftarrow t \\ v &\longleftarrow u \\ u &\longleftarrow r \\ &\text{until } r = 1 \end{aligned} \tag{10.6}$$

This loop uses both the quotient and remainder from Euclid's algorithm. The third line uses the `poly_normal()` routine from chapter 8. At the termination of the loop we will have $r = 1$. At that point the variable t is the inverse. The reason this works is that the extended Euclidean algorithm maintains $t \cdot b = r$. On each step t is increased and r is decreased until $r = 1$ which is the $\gcd(irrd, b)$. Then we have

$$t \cdot b = 1 \bmod irrd.$$

Thus t is the inverse of b modulo the irreducible polynomial.

When doing general polynomial equations modulo an irreducible polynomial the formula will include division. This is easy to write as a formula, but not so easy to execute directly. A simple subroutine is all it takes to convert

$$\frac{a}{b} = a \cdot \frac{1}{b} \tag{10.7}$$

It just makes life easier to have division modulo a prime polynomial as a single call.

For an example I will use the irreducible polynomial

$$x^2 + x + 3 \bmod 43$$

to find the inverse of

$$x + 17$$

Following the initialization step in 10.5 we have

$$v = x^2 + x + 3$$
$$u = x + 17$$
$$r = w = q = t = 0$$
$$y = 1$$

The steps in 10.6 give us

- compute quotient q and remainder r from

$$(q, r) = v/u = x - 16, 17$$

- compute ρ from $\rho = 1/lc(r)$

$$\rho = 1/17 = 38$$

- compute r from

$$r = \text{normal}(17) = 1$$

- compute $t = \rho(w - qv)$

$$t = 38(0 - (x - 16)1) = -38x + 6$$

- transfer w, y, v, and u

$$w = 1$$
$$y = -38x + 6$$
$$v = x + 17$$
$$u = 1$$

Since $r = 1$ the loop ends with t as our answer. Adding 43 to the leading coefficient in t to make it positive, we find

$$\frac{1}{x + 17} = 5x + 6 \bmod x^2 + x + 3$$

10.3 Euclid's algorithm code

In this section the subroutine to compute Euclid's algorithm for polynomials is described.

Euclid's algorithm is a low-level division routine required to create a polynomial inversion routine. There are a few subroutines where we require both the quotient and remainder of a division. The subroutine to accomplish this is shown in listing 10.1.

Listing 10.1 Polynomial Euclid algorithm

```
void poly_euclid(POLY *q, POLY *r, POLY a, POLY b)
{
  int i, j;
  mpz_t s;
  POLY tmp;
  int k;

  poly_copy(r, a);
  q->deg = 0;                          r = a
  mpz_set_ui(q->coef[0], 0);           q = 0
  if(b.deg > a.deg) return;            ← if b > a then all done

  mpz_init(s);                         perform division of each coefficient
  poly_init(&tmp);
  q->deg = a.deg - b.deg;              ← quotient degree
  while((r->deg >= b.deg) && r->deg)   ← degree of r > b and not zero
  {
    j = r->deg - b.deg;                ← index of coefficient
    mdiv(s, r->coef[r->deg], b.coef[b.deg]);   ← s = lc(r)/lc(b)
    mpz_set(q->coef[j], s);
    for(i=0; i<=b.deg; i++)            compute s · b
      mmul(tmp.coef[i + j], s, b.coef[i]);
    tmp.deg = r->deg;
    poly_sub(r, *r, tmp);              ← reduce r by s · b
  }
  mpz_clear(s);
  poly_clear(&tmp);
}
```

The initialization assumes q may have a previous value, so I force q to zero. If the degree of a is less than the degree of b then the remainder is a as it should be. The degree of the quotient is the difference in degrees of numerator and denominator. The loop then executes the formulas described in equation 10.2. When the degree of r is less than the degree of b the loop terminates. The extra check for degree of r not being zero comes from an edge case when b is a degree zero polynomial (a constant).

10.4 Gcd code

The code to compute the greatest common divisor between two polynomials is shown in this section..

The gcd algorithm uses the remainder from Euclidean division to find the common factors between two polynomials. When there are no common factors, we have found an irreducible polynomial. The initial checks take up more lines of code than the main loop. Listing 10.2 shows how equations 10.3 are implemented. The subscript w was used to indicate "working" so the working value of a_w is always the larger degree polynomial.

Listing 10.2 Gcd initialization

```
void poly_gcd(POLY *d, POLY a, POLY b)
{
  POLY aw, bw, q, r;

  poly_init(&aw);   ← set aw = 0
  if(poly_cmp(a, aw))
  {
    poly_copy(d, b);   | input a = 0
    poly_clear(&aw);   | output d = b
    return;
  }
  poly_init(&bw);   ← set bw = 0
  if(poly_cmp(b, bw))
  {
    poly_copy(d, a);
    poly_clear(&aw);   | input b = 0
    poly_clear(&bw);   | output d = a
    return;
  }
  if(a.deg >= b.deg)
  {
    poly_copy(&aw, a);   | a > b
    poly_copy(&bw, b);   | good order
  }
  else
  {
    poly_copy(&aw, b);   | b > a
    poly_copy(&bw, a);   | swap order
  }
```

After initializing the quotient and remainder variables the main loop and termination of the gcd routine is shown in listing 10.3.

Listing 10.3 Gcd main loop

```
  poly_init(&q);
  poly_init(&r);
  while(bw.deg > 0)   ← loop until b_w == 0
  {
    poly_euclid(&q, &r, aw, bw);   ← quotient and remainder from a_w / b_w
    poly_copy(&aw, bw);   ← a_w = b_w
```

```
    poly_copy(&bw, r);     ← b_w = r
  }
  if(!mpz_cmp_ui(bw.coef[0], 0))     | if b_w == 0
    poly_copy(d, aw);                | a_w holds result
  else                               | otherwise
    poly_copy(d, bw);                | b_w is result
  poly_clear(&r);
  poly_clear(&q);
  poly_clear(&bw);
  poly_clear(&aw);
}
```

The loop continues until the degree of b_w goes to zero. At that point the value of b_w coefficient to t^0 is checked to see if it is zero. If it is, then all of a_w is the gcd result. Otherwise, the result is b_w which will be 1 if there are no common factors, or it could be a constant value.

10.5 Inversion modulo a prime polynomial

In this section the code which computes the inverse of a polynomial is described.

To compute the slope between two points on a field extension curve we need a denominator of the sum of the y values of the two points (equation 3.2). As with a field prime, we first compute the inverse modulo the prime polynomial.

The inversion routine uses both the quotient and remainder from Euclid's algorithm. The initialization follows equations 10.5 and the main loop follows equations 10.6. The variable `t` is the answer we seek and all the other variables keep track of previous quotients and remainders to allow the reduction process to proceed.

Listing 10.4 shows the initialization process for inversion. Every variable in equation group 10.5 is created and automatically set to zero. The nonzero variables are each set to their initial condition. Note that the value of `one` is a polynomial of degree 0 with the coefficient of t^0 set to 1.

Listing 10.4 Inversion initialization

```
void poly_invert(POLY *a, POLY b)
{
  mpz_t rho;
  POLY r, u, q, one, t, w, tmp, y, v;
  int done, i;

  mpz_init(rho);     ← ρ = 0
  poly_init(&v);                 | v = irrd
  poly_copy(&v, irrd);           |
  poly_init(&r);     ← r = 0
  poly_init(&u);                 | u = b
  poly_copy(&u, b);              |
  poly_normal(&u);               |
  poly_init(&w);     ← w = 0
```

```
  poly_init(&q);   ← q = 0
  poly_init(&one);
  mpz_set_ui(one.coef[0], 1);   | one = 1 · t^0
  poly_init(&t);   ← t = 0
  poly_init(&y);
  minv(y.coef[0], b.coef[b.deg]);   | y = 1/lc(b) · t^0
```

In listing 10.5 the variable `done` is used to check if the remainder goes to 1. The inverse of the leading coefficient of `r` is kept as a separate value in `rho` and then `r` is normalized using routine 8.9. To compute the 4th line in equation 10.6 the variable `tmp` holds $q \cdot y$, so it can be subtracted from `w`. Every coefficient in `t` is then multiplied by `rho`. The rest of the loop is copying new values to old ones and then checking to see if the loop is finished.

Listing 10.5 Inversion main loop

```
  poly_init(&tmp);
  done = 0;                    | finished when
  while(!done)                 | r == 1
  {                                                  | quotient and remainder
    poly_euclid(&q, &r, v, u);  ← from v / u
    minv(rho, r.coef[r.deg]);   ← ρ = 1/lc(r)
    poly_normal(&r);            ← r *= ρ
    poly_mul(&tmp, q, y);       | t = w - q · y
    poly_sub(&t, w, tmp);       |
    for(i=0; i<=t.deg; i++)                          | t *= ρ
      mmul(t.coef[i], t.coef[i], rho);               |
    poly_copy(&w, y);           | w = y
    poly_copy(&y, t);           | y = t
    poly_copy(&v, u);           | v = u
    poly_copy(&u, r);           | u = r
    done = poly_cmp(r, one);    ← is r == 1?
  }
  poly_copy(a, t);   ← output t as result
  mpz_clear(rho);
  poly_clear(&r);
  poly_clear(&u);
  poly_clear(&q);
  poly_clear(&one);   | clean up stack
  poly_clear(&t);     |
  poly_clear(&w);     |
  poly_clear(&tmp);   |
  poly_clear(&y);
  poly_clear(&v);
}
```

There are a lot of variables in this routine, so the end is simply saving the final result to the expected output location and clearing out all the variables to prevent memory leaks.

10.6 Division modulo a prime polynomial

In this section inversion code is extended to act as division of two polynomials.

The last routine in this chapter is very simple. The division routine is shown in listing 10.6. The variable `q` holds the inverse of input `c`. The input `b` is multiplied by `q` and we are done. This is exactly the same process as the `mod_div()` and `mdiv()` routines from chapter 2. The main difference here is the lack of check for division by zero. Eventually `mod_div()` will be called in `poly_euclid()` and that error will be exposed. The program will halt and the long process of debugging will begin.

Listing 10.6 Division modulo irreducible polynomial

```
void poly_div(POLY *a, POLY b, POLY c)
{
  POLY q;
  int i;

  poly_init(&q);
  poly_invert(&q, c);
  poly_mul(a, b, q);
  poly_clear(&q);
}
```

$\mathbf{a} = \frac{1}{c} \cdot b$

10.7 Summary

- Euclid's division algorithm applies to polynomials over a finite field as well as to integers.
- The greatest common divisor (gcd) function between two polynomials returns the other argument if one input is zero.
- The gcd of two polynomials uses the remainder from Euclidean division.
- The gcd code is used to find irreducible polynomials.
- Inversion modulo a prime polynomial uses the Extended Euclidean algorithm to solve the equation

$$A \cdot t \cong 1 \bmod f.$$

  The polynomial $t = 1/A$.
- Division of polynomials is interpreted as inversion then multiplication.
- All the routines in this chapter are essential for computing pairing algorithms in part 3.

Chapter Bibliography

Cohen, Henri. 2000. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag. 106, 108

von zur Gathen, Joachim, & Gerhard, Jürgen. 1999. *Modern Computer Algebra*. 1 edn. Cambridge University Press. 109

10.8 Answers to exercises

- 10.1) Initialize with rem = $13x^7 + 14x^3 + 19x + 6$ and quot = 0.
  Then $s = 13$, index = 2 and quot = $13x^2$
  $s \cdot b \cdot x^2 = 13x^7 - 13x^4 + 6$. Subtracting from rem gives
  rem = $13x^4 + 14x^3 + 19x$
  Final result is quotient = $13x^2$ and remainder = $13x^4 + 14x^3 + 19x \bmod 43$
- 10.2) $x^2 + 22x + 8$. When multiplied out it would be $\gcd(x^3 + 13x^2 + 4x + 12, x^4 + 22x^3 + 2x^2 + 16x + 11)$. This becomes tedious without a computer.

11 Creating irreducible polynomials

This chapter covers

- Finding irreducible polynomials
- Code for finding irreducible polynomials

Computing pairings of points on elliptic curves over field extensions requires an irreducible polynomial. The irreducible polynomial defines the specific values of the field extension. In this chapter, the details of how to create an irreducible polynomial are described.

In chapter 8 I defined an irreducible polynomial over a finite field as a polynomial with coefficients taken modulo a prime number which has no other factors. A field extension of degree k will have a defining irreducible polynomial of degree k. There are a great many irreducible polynomials one can choose to define a field extension. The simplest polynomial with the highest probability of existence is the trinomial for any field prime. I first describe the theory for finding irreducible trinomials and then explain the code for finding irreducible trinomials.

11.1 Basic theory of irreducible polynomials

In this section I cover the theory behind construction of irreducible polynomials. It's actually more like discovering because the process involves trial and error. I discuss a few theorems that we can take advantage of and then explain an algorithm which will find

irreducible polynomials we can use.

I have found reference (Lidl & Niederreiter, 1997) to be very useful for understanding finite fields over polynomials and finite field extensions. Reference (von zur Gathen & Gerhard, 1999) is very useful for algorithms associated with field extensions and polynomials. Both of these books prove the statement that the product of all monic irreducible polynomials whose degrees divide n is $x^{q^n} - x$. That is a very powerful thing to know.

Theorem 3.84 in (Lidl & Niederreiter, 1997) says: for a prime n, the trinomial x^n+x+a will be irreducible under certain conditions. Rather than compute those conditions I'll use Ben-Or's algorithm explained in section 14.9 of (von zur Gathen & Gerhard, 1999) to test for irreducible polynomials.

For purposes of security I am going to choose prime field extensions for values of n. Alternatives for efficiency using small factors for n are described as towers in reference (Koblitz & Menezes, 2005). An example would be $n = 15$ where you can have a polynomial of 3 terms be coefficients for a polynomial with 5 terms:

$$(s_{52}t^2 + s_{51}t + s_{50})x^5 + (s_{42}t^2 + s_{41}t + s_{40})x^4 + \cdots \tag{11.1}$$

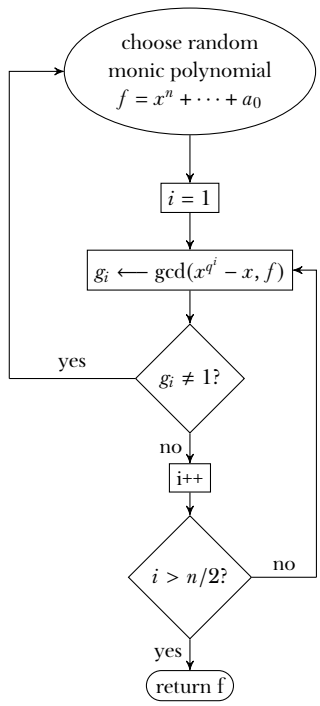

Figure 11.1 Ben-Or's algorithm for finding irreducible polynomials using trial and error

Over a finite field $q = p^n$ Ben-Or's algorithm is shown in figure 11.1. This uses the above statement that all monic irreducible polynomials whose degree divides i are contained in

$x^{q^i} - x$. We search for polynomials of degree 1 through $n/2$ which could be factors in a chosen polynomial.

As an example of how the algorithm works, take the irreducible polynomial from chapter 8: $f = x^2 + 13x + 1 \bmod 1187$. We first set $i = 1$ and compute the $\gcd(x^{1187} - x, f)$. Pari/gp gives the constant 849, which can be normalized to 1. Since $n = 2$ we are actually done. But to double-check, I used PARI/gp to find $\gcd(x^{1187^2} - x, f)$. The result was f, which is to be expected since $x^2 + 13x + 1$ must be one of the factors of $x^{1187^2} - x$ by the theorems mentioned at the start of this section.

Ben Or's algorithm starts by choosing a random polynomial. The idea here is that irreducible polynomials are uniformly distributed over all possible coefficients. The power index is initialized with $i = 1$. The main step is to check if there are any common factors between polynomial f and $x^{q^i} - x$. If there are, then f has factors so it can not be irreducible. We only need to search i up to $n/2$ because any factor larger was found by its smaller companion.

By combining the theorem for prime degree with Ben-Or's algorithm the random choice can be replaced with a counter on the coefficient to the constant term (x^0) of polynomial f.

According to Theorem 3.86 in (Lidl & Niederreiter, 1997) the density of irreducible polynomials goes as q/n where $q = p^n$ is a field extension and n is the degree of the irreducible polynomial we seek. For our case, p is over 160 bits and n is four to six bits. We expect to find an irreducible polynomial very quickly!

Exercise 11.1

Find the first irreducible trinomial for $x^7 + x + b \bmod 29$. Hint: use PARI/gp to factor

```
Mod(1, 29)*x^7 + Mod(1, 29)*x + Mod(b, 29)
```

11.2 Code for finding irreducible polynomials

In this section I describe one way to deal with using and finding irreducible polynomials. I first introduce storage and methods for setting and retrieving values. Then I dive into the execution of Ben Or's algorithm to find an irreducible trinomial.

Because a field extension requires a fixed irreducible polynomial as a reference, I create two variables as globals in the `poly.c` file. The variable `irrd` has been seen before. The variable `ptok` is $q = p^k$ in Ben-Or's algorithm. The variables are shown in listing 11.1.

Listing 11.1 Irreducible static variables

```
static POLY irrd;        | irreducible basis polynomial (degree k)
static mpz_t ptok;       |  p^k
```

Listing 11.2 shows how the irreducible polynomial is set into the variables in listing 11.1. The input polynomial `i` is placed into the static variable `irrd`. The degree of poly-

nomial `i` is used to compute $q = p^k$.

Listing 11.2 Irreducible set function

```
void poly_irrd_set(POLY i)
{
  poly_init(&irrd);          | polynomial irrd
  poly_copy(&irrd, i);       | initialized
  mget(ptok);          ← get p
  mpz_pow_ui(ptok, ptok, i.deg); | save field size p^k
}
```

Listing 11.3 shows two simple routines that return either the irreducible polynomial or the value of q.

Listing 11.3 Irreducible get functions

```
void poly_irrd_get(POLY *i)
{
  poly_copy(i, irrd);   ← return irreducible polynomial
}

void poly_q_get(mpz_t pk)
{
  mpz_init(pk);
  mpz_set(pk, ptok);   ← copy extension field cardinality
}
```

Listing 11.4 shows the start of the routine which finds an irreducible polynomial of a specified degree. If the requested degree is larger than the maximum sized polynomials allowed, the routine returns 0 which means it did not find an irreducible polynomial. The calling program should trap the error at that point because none of the mathematics will work without one.

Listing 11.4 Irreducible polynomial startup

```
int poly_irreducible(POLY *f, long n)
{
  POLY q, r, x, xp[MAXDEGREE/2], xpm1;
  mpz_t j, prime;
  long i, mlimt, done;

  if(n > MAXDEGREE)          | safety check
    return 0;                | change MAXDEGREE to fix
```

Listing 11.5 shows the variable initialization. The polynomial `r` is the trial polynomial. If this passes Ben-Or's test it is returned as the result. It starts out as $x^n + x + 2$. The array of polynomials `xp[]` hold the powers of x^{p^i} modulo r. The variable `xpm1` holds $x^{p^i} - x$ for each i through the loop.

Listing 11.5 Irreducible polynomial variable initialization

```
poly_init(&r);
r.deg = n;                          | test polynomial
mpz_set_ui(r.coef[n], 1);           | r = x^n + x
mpz_set_ui(r.coef[1], 1);           |
poly_init(&x);
x.deg = 1;                          | variable x
mpz_set_ui(x.coef[1], 1);           | is 1 · x^1
poly_init(&q);   ← gcd result
poly_init(&xpm1);   ← x^(p^m) − x
mlimt = n/2;                        | space for
for(i=0; i<mlimt; i++)              | x^(p^m)
  poly_init(&xp[i]);                | for all m < n/2
mget(prime);
mpz_init_set_ui(j, 2); ← start at a_0 = 2
```

Listing 11.6 is the main rejection loop. For initial testing the value of `prime` is small, so I check that the variable `j`, which is the coefficient of x^0, does not exceed the prime value. For large primes this would take the age of the universe, so it is pointless.

Listing 11.6 Irreducible polynomial rejection loop

```
done = 0;
while((mpz_cmp(j, prime) < 0) && !done)   ← for small primes, check j < p
{
  mpz_set(r.coef[0], j); ← convert counter to coefficient
  mpz_add_ui(j, j, 1);   ← increment counter
  poly_mulprep(r);   ← create multiplication table
  i = 0;
  q.deg = 0;                        | keep going while
  while((i<mlimt) && !q.deg)  ←     | gcd result is 1
  {
    if(!i)
      poly_xp(&xp[0], x);           | index i
    else                            | is 1 less than
      poly_xp(&xp[i], xp[i-1]);     | exponent power
    poly_sub(&xpm1, xp[i], x);      | q = gcd(x^(p^i) − x, r)
    poly_gcd(&q, xpm1, r);
    i++;
  }
  if(q.deg)   ← has factors so not irreducible
    continue;
```

Once the polynomial `r` is set, it is used to create the multiplication matrix described in chapter 8. The variable `i` is used as an index for the array of powers of x modulo r. But as we see in Ben-Or's algorithm the first power is 1. So the index is off by 1 for each power which means `xp[i]` $= x^{p^{i+1}}$.

The inner while loop checks for `q.deg` being zero. If it is not, then `r` has a factor in $x^{p^{i+1}}$

so it can not be irreducible. If the inner loop bails out with `q.deg > 0` the next `j` value will be tested.

If `i` is incremented to `mlimit` and `q.deg` has remained zero for every test then we have found an irreducible polynomial. Listing 11.7 shows the end of the routine.

Listing 11.7 Irreducible polynomial finish

```
      if we get this far
      it is irreducible
    done = 1;              finished with loop
    poly_copy(f, r);       and output result
  }
  poly_clear(&r);
  poly_clear(&x);
  poly_clear(&q);
  for(i=0; i<mlimt; i++)
    poly_clear(&xp[i]);    clean up stack
  poly_clear(&xpm1);
  mpz_clears(j, prime, NULL);
  if(done)
    return 1;
  return 0;    ← will never be executed!
}
```

The main task is cleaning up all the variables. The expectation is that `done` will always be set and the last line of code will never be executed. While reference (Lidl & Niederreiter, 1997) omits the proof, theorem 3.86 of that text ensures us that trinomials used in the above routine will be irreducible with high probability.

11.3 Summary

- Trinomials are the simplest polynomials with high probability of being irreducible over a finite field. Trinomials reduce the amount of work required in computing polynomial products and remainders.
- A k degree polynomial f is irreducible when

$$\gcd(x^{p^i} - x, f) = 1$$

  for all $1 \leq i \leq k/2$.
- We can find irreducible trinomials incrementing the constant term a_0 until one passes Ben Or's algorithm. Mathematicians have proven the density of irreducible polynomials is high, so this is guaranteed to work in only a few steps.

Chapter Bibliography

Koblitz, Neal, & Menezes, Alfred. 2005. Pairing-Based Cryptography at High Security Levels. *Pages 13–36 of:* Smart, Nigel P. (ed), *Cryptography and Coding*. Berlin, Heidelberg: Springer Berlin Heidelberg. 118

Lidl, Rudolf, & Niederreiter, Harald. 1997. *Finite Fields*. 2 edn. Encyclopedia of Mathematics and its Applications. Cambridge University Press. 118, 119, 122

von zur Gathen, Joachim, & Gerhard, Jürgen. 1999. *Modern Computer Algebra*. 1 edn. Cambridge University Press. 118

11.4 Answer to exercise

- 11.1) $b = 7$ is the first irreducible trinomial. PARI/gp gives

```
factor(Mod(1,29)*x^7+Mod(1,29)*x+Mod(7,29))
%29 =
[Mod(1, 29)*x^7 + Mod(1, 29)*x + Mod(7, 29) 1]
```

12 Taking square roots of polynomials

This chapter covers

- Polynomial pseudo-division
- Resultant function of two polynomials
- Quadratic residue for a polynomial
- Square root of a polynomial modulo a prime polynomial

In this chapter we dive into the details of computing square roots modulo a prime polynomial so we can find solutions to the elliptic curve equation. At the end of chapter 2 I showed how to compute square roots over a prime number field. The same process is used over a field created by an irreducible polynomial.

The equation of an elliptic curve has the form $y^2 = f(x)$. The example given in chapter 3 is $y^2 = x^3 - 5x + 5$, where $f(x) = x^3 - 5x + 5$. To find the y coordinate requires taking a square root. When the field we are using is an extension of a prime field the coordinates are polynomials.

Computing square roots modulo an irreducible polynomial is similar to computing square roots modulo a prime number. The main difference is that we are looking for two identical polynomial factors. This is similar to factoring a polynomial.

In the article by (Doliskani & Schost, 2011), there are descriptions of algorithms for

exceptionally high polynomial degrees. They show that the algorithm of Tonelli-Shanks is perfectly adequate for our purposes since our embedding degrees are small. This is fortunate because we already know how to perform the Tonelli-Shanks algorithm with prime numbers.

The methods proposed in reference (Doliskani & Schost, 2011) include using a function called a resultant of two polynomials. A resultant of two polynomials is zero if they have a common factor. We will use the resultant of a polynomial with the irreducible polynomial of the field extension to determine if a polynomial is a quadratic residue. As we saw in chapter 2, knowing if something is a quadratic residue tells us we can in fact compute a square root. To compute a resultant I will follow algorithm 3.3.7 in (Cohen, 2000).

The computation of the resultant uses in turn an algorithm called pseudo-division. So in the following description of how to compute a square root, I describe how the resultant computes a quadratic residue, and how pseudo-division helps compute the resultant.

After the theory is described, routines to compute pseudo-division, resultant, quadratic residue and the square root itself will be listed.

12.1 Mathematics for square root modulo a prime polynomial

In this section I cover the algorithms used to find a square root over a field extension so we can ensure a point is on an elliptic curve.

Factoring polynomials is a complicated process. Factoring polynomials modulo an irreducible polynomial over a prime field can get fairly deep. Fortunately we don't need most of that machinery to search for a square root. From the formula for an elliptic curve ($y^2 = x^3 + a_4x + a_6$) we only need the square root of a polynomial modulo the prime polynomial to find the y coordinate. Similar to how a square root was done in chapter 2 modulo a prime number we first want to know if we can even take a square root by determining if the polynomial is a quadratic residue.

Figure 12.1 shows the same algorithm as described in chapter 2 with primes replaced by polynomials. Instead of checking the last two bits of field prime p, we check the last two bits of p^k. If both are not set we perform the Tonelli-Shanks algorithm.

The most significant change is the blue box in figure 12.1 which requires finding a polynomial quadratic nonresidue. Solving this problem took a round about path which involved learning about factoring high degree polynomials.

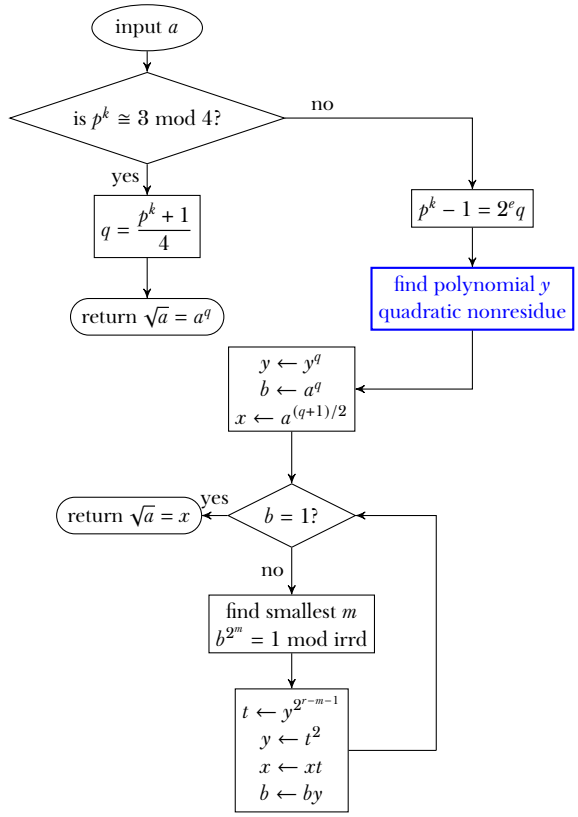

Figure 12.1 **Polynomial square root algorithm for finding y value of point on elliptic curve**

Reference (Doliskani & Schost, 2011) is a general factorization process for finite field elements modulo an irreducible polynomial f. In section 3.2 the authors first state that testing if an element a has a t-th root is the same as testing if $a^{(q-1)/t} = 1$ where $q = p^n$. They then go on to say "In the particular case when t divides $p-1$, we can actually do better: we have $a^{(q-1)/t} = \text{res}(f, a)^{(p-1)/t}$, where $\text{res}(\cdot, \cdot)$ is the resultant function." For us what that means is finding out if a polynomial a is a quadratic residue or not requires checking

$$\text{res}(f, a)^{(p-1)/2} \stackrel{?}{=} 1.$$

If the result is 1 we can take a square root, otherwise we cannot. Figure 12.2 lays out the quadratic residue algorithm as it will be written in code.

A resultant of two polynomials is a single value modulo the field prime. Section 3.3.2 of (Cohen, 2000) describes a couple of equivalent definitions. One is a determinant of the coefficients of both polynomials f and a. Another is the product of the difference in every root of both polynomials. The resultant algorithm 3.3.7 in (Cohen, 2000) uses neither of those definitions. (Cohen, 2000) proves his algorithm gives the correct answer and that algorithm is what I use.

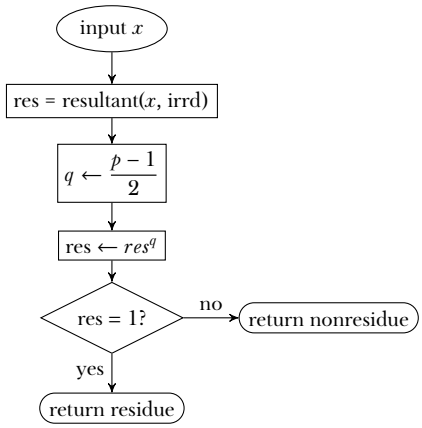

Figure 12.2 Polynomial quadratic residue algorithm to determine if polynomial has square root or not

The resultant algorithm is shown in figure 12.3. If either input polynomial is zero, the resultant is zero. The next thing we see in the algorithm of computing the resultant is $a \leftarrow \text{cont}(A)$ which is the content of a polynomial. The content of a polynomial is found by taking the gcd of all the coefficients. For monic polynomials this is always 1.

As an example look at

$$a = 4x^4 + 24x^3 + 12x^2 + 4x + 32.$$

The $\text{cont}(a) = 4$. The first step in finding a resultant is removing the content of each polynomial by dividing each coefficient by the common factor.

Along with removing the content of each polynomial, the result variables are initialized. The higher degree polynomial is forced to be A. If both polynomials have odd degree the sign variable s is set to -1.

The main loop starts by setting δ to the power that would come from dividing A/B. If both polynomials have odd degree the sign of s is toggled. The quotient and remainder from pseudo-division are then computed. The reduction step modifies A, B, g and h. The loop terminates when $\deg(B)$ goes to zero.

A degree zero B means it is a constant so the $lc(B)$ in the last step is the only coefficient of B. The final output is the product of sign, initial content values and variable h.

It is pretty amazing this algorithm computes the determinant of a matrix created by the coefficients of two polynomials. It definitely works to determine quadratic residues.

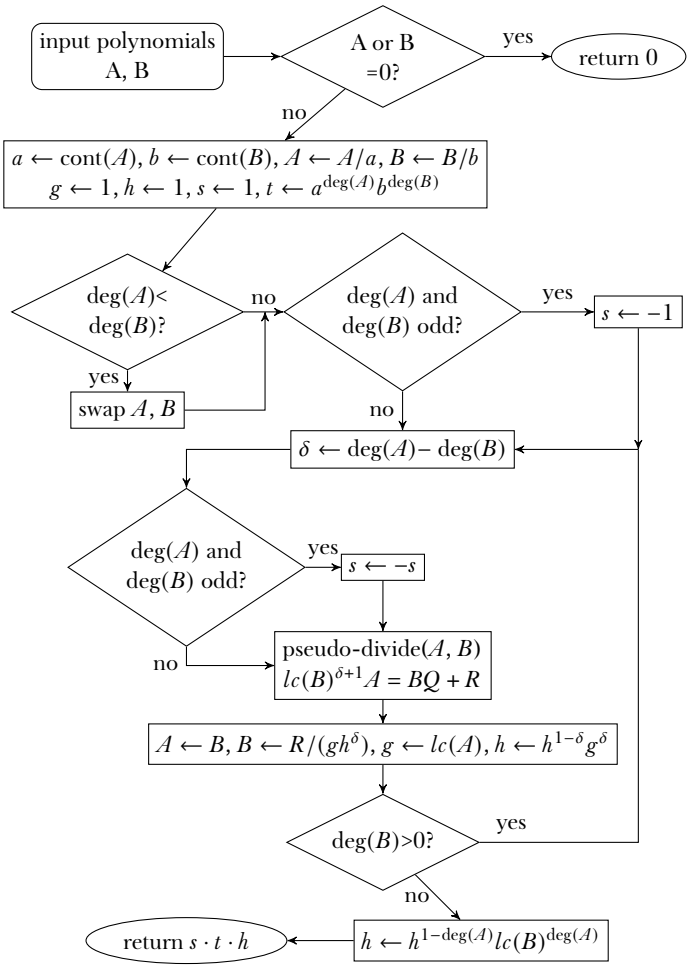

Figure 12.3 Resultant algorithm

An important step within the resultant algorithm is pseudo-division. This works with the leading coefficients (symbolized by $lc(\cdot)$) of the divisor polynomial. Given two polynomials A and B with $lc(B) = d$ then

$$d^{\deg(A)-\deg(B)+1} A = BQ + R$$

where Q is the quotient and R the remainder with $\deg(R) < \deg(B)$.

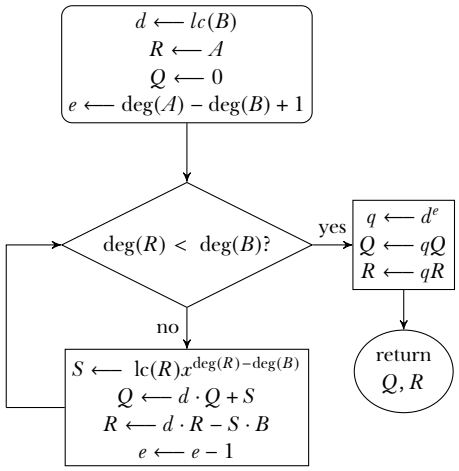

Figure 12.4 Pseudo-divide algorithm

Figure 12.4 shows the pseudo-division algorithm taken from (Cohen, 2000) algorithm 3.1.2. The first step initializes the quotient Q and remainder R as well as the starting exponent e and leading coefficient d. The reason the second step checks if the algorithm is done is in case $\deg(A) < \deg(B)$, at which point there is nothing more to do.

The compute step puts the leading coefficient of R into the coefficient of temporary variable S which would be the leading coefficient of a division of R/B. The variable S is then used to modify Q and R and the exponent is decremented.

There is a lot of similarity between the algorithm in figure 12.4 and equation 10.2. There is no actual division computation so calling this pseudo-division makes sense.

Exercise 12.1

In the computation of polynomial square roots, the resultant has a similar task to what function in the calculation of square roots modulo a prime number?

12.2 Code for square roots modulo a prime polynomial

In this section, the code for computing square roots of polynomials is described. We need these routines to find random points on field extension curves. Those points are needed to compute the pairing calculations of chapters 18 and 19.

There are five subroutines involved with computing a square root modulo an irreducible polynomial. These routines are

- content of polynomial,
- pseudo-division of polynomials,
- resultant of two polynomials,

- quadratic residue of a polynomial,
- computing the square root of a polynomial.

The simplest are content and checking if a polynomial is a quadratic residue. The more complicated routines are pseudo-division, resultant and taking the square root. The order of calling is content, pseudo-division, resultant, residue check, and finally square root. The following sections are in order of requirement.

12.2.1 Content routine

Computing the content of a polynomial finds the greatest common divisor of all the coefficients. If the gcd between any two coefficients is 1, there is no reason to check the remaining coefficients. So I start with the leading coefficient hoping the polynomial is monic, and the routine can exit early. This is shown in listing 12.1.

Listing 12.1 Content of polynomial

```
void poly_cont(mpz_t cont, POLY A)
{
  int i;
  mpz_t rslt;

  mpz_init_set(rslt, A.coef[A.deg]);   ← set result to lc(A)
  for(i=A.deg-1; i>=0; i--)            | from leading coefficient
  {                                    | down to lowest
    mpz_gcd(rslt, rslt, A.coef[i]);
    if(!mpz_cmp_ui(rslt, 1))           | when gcd of result
      break;                           | with coefficient
                                       | == 1 then done
  }
  mpz_set(cont, rslt);   ← copy result to return value
  mpz_clear(rslt);
}
```

The comparison `mpz_cmp(rslt, 1)` will give 0 when the variable `rslt` is 1. If there is a common factor, `rslt` will contain it on every `mpz_gcd()` call. So the returned value is the common factor which defines the content.

12.2.2 Pseudo-division routine

The pseudo-division routine has three phases. The initialization is easy, the main loop is messy, and the cleanup is the last phase. Listing 12.2 shows the initialization phase.

Listing 12.2 Pseudo-division initialize

```
void poly_pseudo_div(POLY *Q, POLY *R, POLY A, POLY B)
{
  long e, i, k;
  mpz_t d;
  POLY S, T;
```

```
poly_init(&S);
poly_init(&T);
poly_copy(R, A);          | set S = 0, T = 0 and R = A
Q->deg = 0;
mpz_set_ui(Q->coef[0], 0); | set Q = 0
e = A.deg - B.deg + 1;
mpz_init_set(d, B.coef[B.deg]); | set e to degree difference and d to lc(B)
```

The return values are quotient Q and remainder R. As with the division routine I assume variable Q might be something random, and so I force it to zero. Variables `e` and `d` are set according to the first block in figure 12.4.

The main loop is shown in listing 12.3. The loop continues while the $\deg(R) \geq \deg(B)$. The variable `T` holds $S \cdot B$ which is really just a shift of B by the $\deg(S)$ multiplied by the leading coefficient of R. Since only one coefficient of S is set, that coefficient is cleared at the end of the loop.

Listing 12.3 Pseudo-division main loop

```
while(R->deg >= B.deg) ← while remainder greater than divisor
{
  S.deg = R->deg - B.deg;    ← degree of S if division happened
  mpz_set(S.coef[S.deg], R->coef[R->deg]); ← lc(S) = lc(R)
  for(i=0; i<=Q->deg; i++)              | scale Q by d
    mmul(Q->coef[i], Q->coef[i], d);
  poly_add(Q, *Q, S);     ← Q ← Q + S
  for(i=0; i<=R->deg; i++)              | scale R by d
    mmul(R->coef[i], R->coef[i], d);
  k = S.deg;
  for(i=0; i<=B.deg; i++)                                          | T = B · S
    mmul(T.coef[i + k], B.coef[i], S.coef[k]);
  T.deg = B.deg + k;
  poly_sub(R, *R, T);  ← reduce R by T
  e--;                                  | decrement exponent
  mpz_set_ui(S.coef[k], 0); | clear S to 0
}
```

The change of variable from `S.deg` to `k` was simply to make the multiply indexing easier to read.

When the loop finishes we have a remainder which is smaller than the denominator. Listing 12.4 shows the cleanup phase.

Listing 12.4 Pseudo-division clean up

```
if(e >= 1) ← only change Q and R for e > 0
{
  mpowi(d, d, e);    ← d ← d^e
  for(i=0; i<=Q->deg; i++)              | scale Q by d
    mmul(Q->coef[i], Q->coef[i], d);
```

```
    for(i=0; i<=R->deg; i++)                    | scale R
      mmul(R->coef[i], R->coef[i], d);          | by d
  }
  mpz_clear(d);
  poly_clear(&S);                               | clean
  poly_clear(&T);                               | up
}                                               | stack
```

If variable `e` is zero then $d^e = 1$ and there is no change to the quotient or remainder. Otherwise, the value of d^e is multiplied with every coefficient of both quotient and remainder. The internal variables are cleared out to prevent memory leaks and the routine is done.

12.2.3 Resultant subroutine

In this section I describe the code which computes the resultant of two polynomials.

The resultant subroutine breaks up into four phases. The description in figure 12.3 has a main loop which I'll show as one phase. The first phase determines if either input is zero, then the routine just returns zero. This is shown in listing 12.5.

Listing 12.5 Resultant zero check

```
void poly_resltnt(mpz_t rsltnt, POLY A, POLY B)
{
  POLY Aa, Bb, Q, R;
  mpz_t g, h, ta, tb, a, b;
  long dlta, s, i;

                                                | if either A or B is zero
  if(!A.deg && !A.coef[0])                      | resultant is zero
  {
    mpz_set_ui(rsltnt, 0);
    return;                                     | both degree
  }                                             | and coefficient
  if(!B.deg && !B.coef[0])                      | must be zero
  {                                             | to exit
    mpz_set_ui(rsltnt, 0);
    return;
  }
```

The initialization portion uses variables `ta` and `tb` to hold the value of $\text{cont}(A)^{\deg(B)}$ and $\text{cont}(B)^{\deg(A)}$ respectively. The variables `Aa` and `Bb` hold $A/cont(A)$ and $B/cont(B)$. The remaining variables are the same as in figure 12.3.

Listing 12.6 Resultant initialization

```
  mpz_inits(g, h, ta, tb, a, b, NULL);          | initialize local variables
  poly_cont(a, A);                              | get content
  poly_cont(b, B);                              | of input polynomials
```

```
poly_init(&Aa);
if(mpz_cmp_ui(a, 1))   ← content != 1
{
  Aa.deg = A.deg;
  for(i=0; i<=A.deg; i++)                    | Aa← A/a
    mdiv(Aa.coef[i], A.coef[i], a);          |
  mpowi(ta, a, B.deg);    ← ta← a^deg(B)
}
else
{
  poly_copy(&Aa, A);      | if content == 1
  mpz_set_ui(ta, 1);      | just copy A to Aa
}
poly_init(&Bb);
if(mpz_cmp_ui(b, 1))    ← content != 1
{
  Bb.deg = B.deg;
  for(i=0; i<=B.deg; i++)                    | Bb← B/b
    mdiv(Bb.coef[i], B.coef[i], b);          |
  mpowi(tb, b, A.deg);     ← tb← b^deg(A)
}
else
{
  poly_copy(&Bb, B);      | if content == 1
  mpz_set_ui(tb, 1);      | just copy B to Bb
}
mpz_set_ui(g, 1);
mpz_set_ui(h, 1);         | initialize
poly_init(&Q);            | g, h and s
s = 1;                    |
if(A.deg < B.deg)     ← make A larger than B
{
  poly_copy(&Q, Aa);      |
  poly_copy(&Aa, Bb);     | swap A, B
  poly_copy(&Bb, Q);      | to ensure
}                         | A > B
poly_init(&R);
```

The main loop from figure 12.3 is shown in listing 12.7. The loop continues as long as deg(B) is not zero.

Listing 12.7 Resultant main loop

```
while(Bb.deg > 0)   ← terminate when B turns into constant
{
  dlta = Aa.deg - Bb.deg;    ← dlta is degree after division
  if((Aa.deg & 1) && (Bb.deg & 1))   | both polynomials odd
    s = -s;                          | change sign
  poly_pseudo_div(&Q, &R, Aa, Bb);   ← quotient and remainder from pseudo-division
```

```
    poly_copy(&Aa, Bb);
    mpowi(a, h, dlta);
    mmul(b, a, g);
    Bb.deg = R.deg;
    for(i=0; i<=R.deg; i++)
      mdiv(Bb.coef[i], R.coef[i], b);

    mpz_set(g, Aa.coef[Aa.deg]);
    i = 1 - dlta;
    mpowi(a, h, i);
    mpowi(b, g, dlta);
    mmul(h, a, b);
  }
```

A ←B
B ← $R/(gh^{\delta})$

save new h, g values

g← $lc(A)$
h← $h^{1-\delta}g^{\delta}$

On exit from the main loop the last line in figure 12.3 becomes listing 12.8. This is a straightforward computation.

Listing 12.8 Resultant final output

finished
compute final resultant

```
    i = 1 - Aa.deg;
    mpowi(a, h, i);
    mpowi(b, Bb.coef[0], Aa.deg);
    mmul(h, a, b);
    mmul(rsltnt, h, ta);
    mmul(rsltnt, rsltnt, tb);
    if(s < 0)
      mneg(rsltnt, rsltnt);
    poly_clear(&Aa);
    poly_clear(&Bb);
    poly_clear(&Q);
    poly_clear(&R);
    mpz_clears(g, h, ta, tb, a, b, NULL);
}
```

h← $h^{1-\mathbf{deg}(A)}lc(B)^{\mathbf{deg}(A)}$

output = $s \cdot h \cdot t$

clean up stack

Rather than create the variable t as in the algorithm I just left `ta` and `tb` as separate values. The final value is always a positive number modulo the field prime.

12.2.4 Quadratic residue

In this section I give the code which determines if a polynomial has a square root or not.

Now that we have the resultant function we can find out if a polynomial is a quadratic residue. If so we can take a square root. Listing 12.9 shows how to compute a polynomial quadratic residue.

Listing 12.9 Quadratic residue routine

```
int poly_sqr(POLY x)
{
  mpz_t res, p, q;
```

```
  int k;

  mpz_inits(res, q, NULL);
  poly_resltnt(res, x, irrd);
  mget(p);
  mpz_sub_ui(q, p, 1);
  mpz_div_ui(q, q, 2);
  mpz_powm(res, res, q, p);
  k = mpz_cmp_ui(res, 1);
  mpz_clears(res, p, q, NULL);
  if(!k)
    return 1;
  return 0;
}
```

compute resultant between input and prime polynomial

$(p-1)/2$

res$(x, f)^{(p-1)/2}$

k is 0 if res is 1

The trick here is that we have to save the result, so we can clear temporary variables off the stack and internal heap of GMP. The compare function returns zero on exact match and ± 1 otherwise. If $res(x, f)^{(p-1)/2}$ equals 1 the polynomial x is a quadratic residue.

12.2.5 Polynomial square root routine

In this section the code to implement square roots of a polynomial is described. This is used to find points on field extension elliptic curves.

The assumption I make for the square root routine is that the check for quadratic residue happens first. This is normally the case for embedding because checking for square root takes less effort than computing the square root and finding out there isn't one.

As with square roots modulo a prime if the value of $p^k \cong 3 \bmod 4$ then we can power our way to a square root. This is shown in listing 12.10.

Listing 12.10 Polynomial square root $p^k = 3 \bmod 4$

```
void poly_sqrt(POLY *sqt, POLY a)
{
  long r, i, ck, m, m2;
  mpz_t pk, q;
  POLY x, b, y, one, bpw, t;

  poly_q_get(pk);
  mpz_init(q);
  if(mpz_tstbit(pk, 0) && mpz_tstbit(pk, 1))
  {
    poly_init(&y);
    mpz_add_ui(q, pk, 1);
    mpz_divexact_ui(q, q, 4);
    poly_pow(&y, a, q);
    poly_copy(sqt, y);
    poly_clear(&y);
    mpz_clears(q, pk, NULL);
    return;
```

see if $p^k = 3 \bmod 4$

last two bits set in p^k use easy method

q = $\frac{p^k + 1}{4}$

allow in place square root = a^q

```
}
```

As with the description of Tonelli-Shanks in chapter 2 there is an initialization section as shown in listing 12.11. The main difference is that most of the variables are now polynomials instead of numbers. The coefficients of the polynomials are modulo the same field prime.

Listing 12.11 Polynomial square root initialize

```
                  | p^k = 1 mod 4
                  | so do Tonelli-Shanks
mpz_sub_ui(q, pk, 1);
r = 0;
while(!mpz_tstbit(q, 0))                  | find p^k - 1
{                                         | = 2^r q
  mpz_divexact_ui(q, q, 2);               |
  r++;                                    |
}
poly_init(&y);
ck = 1;
while(ck)
{                                         | choose random
  poly_rand(&y);                          | non-residue polynomial
  ck = poly_sqr(y);                       |
}
poly_pow(&y, y, q);   <- y <- y^q
poly_init(&b);
poly_pow(&b, a, q);   <- b = a^q
poly_init(&x);
mpz_add_ui(pk, q, 1);                     | x=a^((q+1)/2)
mpz_divexact_ui(pk, pk, 2);               |
poly_pow(&x, a, pk);                      |
poly_init(&one);
one.deg = 0;                              | polynomial constant one
mpz_set_ui(one.coef[0], 1);               |
poly_init(&bpw);
poly_init(&t);
```

The initialization is slightly different here than in the field prime code. The formula in equations 2.2 have a value for x and b. Multiplying b out we have

$$b = ax^2 = a(a^{(q-1)/2})^2 = a^q.$$

The final value for x is then

$$x = a(a^{(q-1)/2}) = a^{(q+1)/2}.$$

While the code looks different it is actually doing the same job.

The main loop is also similar, but instead of checking if a value is 1 I check if a polynomial is 1. While technically it is the same thing the structures being worked with are different. Listing 12.12 shows this main loop.

Listing 12.12 Polynomial square root main loop

```
while(!poly_cmp(b, one)) ← done when b equals 1
{
  m = 0;
  while(!poly_cmp(bpw, one))
  {
    m++;                                  find smallest m
    m2 = 1 « m;                           such that b^(2^m)
    mpz_set_ui(pk, m2);                   equals 1
    poly_pow(&bpw, b, pk); ←              modulo prime polynomial
    if(m == r)
    {
      printf("square root failed\n");                  should never happen
      return;
    }
  }
  mpz_set_ui(bpw.coef[0], 0); ← clear test polynomial
  i = r - m - 1;                          t = y^(2^(r-m-1))
  m2 = 1 « i;
  mpz_set_ui(pk, m2);
  poly_pow(&t, y, pk); ←
  poly_mul(&y, t, t); ←                   y ← t^2
  r = m;
  poly_mul(&x, x, t);                     x ← x t
  poly_mul(&b, b, y);                     b ← b y
}
```

The test for `m == r` should never succeed if the polynomial is a quadratic residue.

When the loop is finished the value for the square root is copied to the desired result storage location and all variables are cleaned out from the stack. This is shown in listing 12.13.

Listing 12.13 Polynomial square root clean up

```
  poly_copy(sqt, x); ← copy result to output
  poly_clear(&x);
  poly_clear(&b);
  poly_clear(&y);
  poly_clear(&one);                       clean up stack
  poly_clear(&bpw);
  poly_clear(&t);
  mpz_clears(pk, q, NULL);
}
```

With the code complete for taking a square root of a polynomial modulo an irreducible polynomial we can now embed random polynomial x values on an elliptic curve defined over the same irreducible polynomial. Visualizing an x and y axis which are finite field polynomials of dimension k with an elliptic curve running through the space is challenging.

The best we can do is revisit the curve from chapter 3 and pretend we know what is going on because the equations are the same.

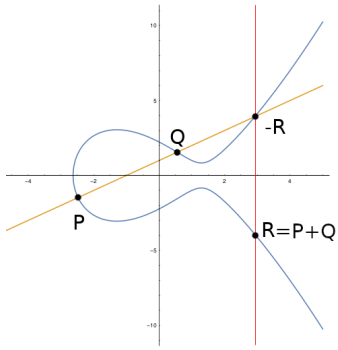

Figure 12.5 Elliptic Curve $y^2 = x^3 - 5x + 5$

As pointed out in chapter 3, finite field elliptic curves have a one to one correspondence with curves over the complex plane. The symmetry we see in figure 12.5 applies to elliptic curves over field extensions. Using images like figure 12.5 we can see how picking some random x polynomial would give rise to two points on an elliptic curve.

12.3 Summary

- The resultant of two polynomials is a single value modulo the field prime of the coefficients. The resultant is used to determine if a polynomial has square root.
- A polynomial A modulo a prime polynomial f has a square root when

$$resultant(A, f)^{(p-1)/2} = 1.$$

  The polynomial A is called a quadratic residue when this is true. When true, we have a point on a field extension elliptic curve.
- The content of a polynomial is the gcd of all its coefficients. Removing the content of a polynomial is an essential step in computing a square root.
- The resultant is computed using pseudo-division and a reduction step. The resultant is used to determine if a polynomial has a square root.
- A polynomial A mod f which is a quadratic residue over an extension field that is congruent to 3 mod 4 has a square root computed using

$$A^{(p^k+1)/4}.$$

  Otherwise the Tonelli-Shanks algorithm is used. This is the same process used in chapter 2 and has the same purpose: to find a point on an elliptic curve.

Chapter Bibliography

Cohen, Henri. 2000. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag. 125, 126, 129

Doliskani, Javad, & Schost, Éric. 2011. Taking roots over high extensions of finite fields. *Math. Comput.*, **83**, 435–446.

12.4 Answer to exercise

- 12.1) The resultant is similar to the Legendre symbol. Both functions determine if it is possible to compute a square root.

13 Finite field extension curves described

This chapter covers

- Cardinality of field extension curves
- Structures for polynomial points and curves
- Embedding polynomial points on curves
- Addition and multiplication of field extension points
- Tiny example showing points on field extension curve

In this chapter curves over finite field extensions are described. These curves are required to compute pairings.

The cryptographic protocols enabled by pairings of points over elliptic curves have a lot of advantages. To compute pairings of points over elliptic curves, we need to use extension fields. But before we can compute pairings, we first need a point addition algorithm over a field extension of an elliptic curve.

Elliptic curve subroutines using polynomials are the subject for this chapter. I'll first cover some of the rules required to create a field extension and then assume we already have found a useful curve. In chapter 14 I'll go into how that actually happens. As with chapter 3 I will cover routines that manipulate point and curve structures, embed polyno-

mials on a curve, create random points, add points and multiply points. While most places replace an `m*` routine with a `poly*` routine there are a few differences because polynomials are more complicated than numbers.

To help visualize the ideas here and in the next few chapters I will use a very tiny curve as an illustration. This curve was definitely cherry-picked to contain as many useful examples as possible.

13.1 Field extension properties

Recall from chapter 2 that a finite field has elements that can be added, multiplied and inverted. In this section I will expand that same concept using irreducible polynomials in place of prime numbers. This is another reason irreducible polynomial is synonymous with prime polynomial.

In chapter 8 I showed how irreducible polynomials act as a modulus for polynomial multiplication and in chapter 10 how irreducible polynomials act as the modulus for inversion. Since we also have addition, it seems pretty clear that an irreducible polynomial creates a finite field. In this section we are going to make use of an irreducible polynomial to create an extension field of a precise size. With the polynomial routines of part 2 as a base, creating elliptic curve routines similar to chapter 3 is then easy.

Theorem 1.87 of reference (Lidl & Niederreiter, 1997) states that an irreducible polynomial f creates a field extension with the root of f as the defining element. Setting $f(t) = 0$ defines those roots. We did that in chapter 8 to create the polynomial multiplication table. Everything we did in part 2 allows us to do mathematics over a field extension.

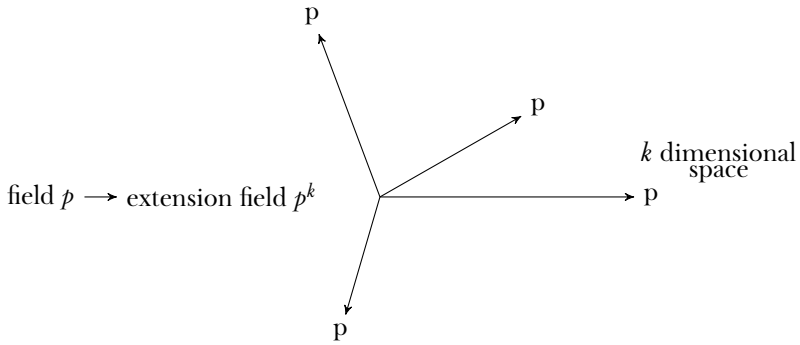

Figure 13.1 Expanding a prime base field p to an extension field p^k. Extension fields are required to compute pairings.

Figure 13.1 is a conceptual illustration of a field extension. A prime field p is extended to a k dimensional vector space. Each dimension acts like the prime field p, but there are k of these fields. A polynomial is used to keep track of where we are in each dimension.

An elliptic curve can be defined over any finite field. However, we can not just pick any irreducible polynomial for a field extension of degree k. Extending a curve from a number field p to a field extension p^k has special rules.

Suppose we have a specific elliptic curve over a prime finite field p with cardinality $\#E$ as in equation 6.2 ($\#E = p + 1 - t$). Let's assume there is a large prime r as a factor of $\#E$. For pairings to work the extension field p^k must also have a factor of r in the extension curve cardinality ($\#E_k$). The smallest value of k for which there is a factor of r in $\#E_k$ is called the embedding degree.

From reference (Freeman *et al.*, 2006) the rule is written as

$$p^k \cong 1 \bmod r. \tag{13.1}$$

Equation 6.2 is true for extension curves but is rewritten as

$$\#E_k = p^k + 1 - t_k. \tag{13.2}$$

Homework problem 5.13 in reference (Silverman, 2013) gives a recurrence relation for t_k. Starting with $t_0 = 2$ and $t_1 = p + 1 - \#E$ (equation 6.2) there is the formula

$$t_{n+2} = t_1 t_{n+1} - p t_n \tag{13.3}$$

which takes the prime p and trace of Frobenius of the curve t_1 to find the cardinality of the field extension $\#E_k$.

According to (Freeman *et al.*, 2006) the average embedding degree for arbitrary curves over a prime field p with points of order r is on the order of r. For a prime r with 160 bits that means the number of coefficients is around 2^{160} and each coefficient is 160 bits. Even imagining the level of computational impossibility is difficult.

Curves over a finite field p are called group 1 curves or G_1. Curves over a field extension p^k are called group 2 curves or G_2. The same coefficients in equation 3.1 ($y^2 = x^3 + a_4 x + a_6$) on a G_1 curve are used to create the points on a G_2 curve. The coefficients a_4 and a_6 are still modulo the same field prime but turn into coefficients of t^0 modulo an irreducible polynomial. So the x and y values become polynomials for the G_2 curve.

After listing all the polynomial elliptic curve code I will give a really tiny example so the G_1 and G_2 points can be listed. This will make the meaning more visible.

Exercise 13.1

Suppose we have an elliptic curve over field prime of 41. There is a "large prime factor" in the cardinality of 29. What is the embedding degree?

13.2 Elliptic curve routines

This section repeats the code from chapter 3 but changes it to work with polynomials in a field extension.

The routines that implement elliptic curves over a field extension are similar to the routines of chapter 3. In addition to operating modulo a field prime they also operate modulo an irreducible polynomial. Similar to chapter 3 a structure for points and curves

is set up but using polynomials. Initializing and clearing these structures is used in all the following routines:

- poly_test_point()
- poly_fofx()
- FF_bump()
- poly_elptic_embed()
- poly_point_rand()
- poly_elptic_sum()
- poly_elptic_mul()

Code for copying points as well as printing points and curves for field extensions will then be shown. A routine to test for the point at infinity (poly_test_point()) followed by a routine to compute the right-hand side of equation 3.1 ($x^3 + a_4x + a_6$) (poly_fofx()) over a field extension is presented.

For embedding a point a special routine is created to increment a polynomial (FF_bump()). It is overly complicated, so it will work with the tiny example curve. With that special routine, the embedding code itself is then presented (poly_elptic_embed()). Creating a random point on a field extension then calls the embedding routine (poly_point_rand()).

The routines to compute addition (poy_elptic_sum()) and multiplication (poly_elptic_mul()) of finite field extension points are at the end of this section.

13.2.1 Polynomial curve setup

This section defines structures used with field extension points and curves. These will be used in all the remaining subroutines in this book..

I start off with a header which defines a point and curve structure using polynomials as shown in listing 13.1. I put this in a file called `poly_eliptic.h` (and I still can't spell!)

Listing 13.1 Polynomial elliptic curve structures

```
#include "poly.h"

typedef struct
{
  POLY   x;        | polynomials for
  POLY   y;        | x and y values
}POLY_POINT;

typedef struct
{
  POLY  a4;        | polynomials for
  POLY  a6;        | curve parameters
}POLY_CURVE;
```

As with structures for group 1 curves the structures in 13.1 require initialization and clearing. These routines are shown in listing 13.2.

Listing 13.2 Polynomial elliptic curve structure manipulation

```
void poly_point_init(POLY_POINT *P)
{
  int i;

  P->x.deg = 0;
  P->y.deg = 0;
  for(i=0; i<MAXDEGREE; i++)
    mpz_inits(P->x.coef[i], P->y.coef[i], NULL);
}

void poly_point_clear(POLY_POINT *P)
{
  int i;

  for(i=0; i<MAXDEGREE; i++)
    mpz_clears(P->x.coef[i], P->y.coef[i], NULL);
}

void poly_curve_init(POLY_CURVE *E)
{
  int i;

  E->a4.deg = 0;
  E->a6.deg = 0;
  for(i=0; i<MAXDEGREE; i++)
    mpz_inits(E->a4.coef[i], E->a6.coef[i], NULL);
}

void poly_curve_clear(POLY_CURVE *E)
{
  int i;

  for(i=0; i<MAXDEGREE; i++)
    mpz_clears(E->a4.coef[i], E->a6.coef[i], NULL);
}
```

zero out both x and y components

remove all components from GMP heap

put polynomials a_4 and a_6 on stack

remove a_4 and a_6 from stack

The polynomial code is not space efficient. It is just a lot easier to see what is going on by simply allowing the possibility of a maximum degree polynomial.

13.2.2 Polynomial curve utilities

In this section a few basic routines are described for manipulating polynomial points.

Utility routines are shown in listing 13.3. Copying a point and printing points which are polynomials are simple `poly*` calls for the x and y components.

Listing 13.3 Polynomial elliptic curve utility functions

```
void poly_point_copy(POLY_POINT *R, POLY_POINT P)
{
  int i;
```

```
  poly_copy(&R->x, P.x);        copy x component
  poly_copy(&R->y, P.y);        copy y component
}

void poly_point_printf(char *str, POLY_POINT P)
{
    printf("%s", str);
    poly_printf("x: ", P.x);     output x and y components on separate lines
    poly_printf("y: ", P.y);
}

void poly_curve_printf(char *str, POLY_CURVE E)
{
    printf("%s", str);
    poly_printf("a4: ", E.a4);   output a4 and a6 components on separate lines
    poly_printf("a6: ", E.a6);
}
```

As with prime field curves the point at infinity is the same test over polynomial curves. Listing 13.4 shows the routine for this test.

Listing 13.4 Polynomial elliptic curve test point at infinity

```
int poly_test_point(POLY_POINT P)
{
  int i;

  if(P.x.deg || P.y.deg)   ← neither polynomial constant then not 0
    return 0;
  if(!mpz_cmp_ui(P.x.coef[0], 0) && !mpz_cmp_ui(P.y.coef[0], 0))
    return 1;
  return 0;          ↑ both x and y zero for point at infinity
}
```

This is pretty easy to bail on if either the x or y are actually polynomials, they can't be zero. If both x and y are constants then I can check if both are zero.

13.2.3 Polynomial curve point embedding

In this section the code for embedding a polynomial onto a field extension curve is explained.

There are several subroutines used to embed polynomial points on a curve. The `fofx()` routine is the same as before, but there is a new routine to increment an x value. Listing 13.5 shows the polynomial version of `fofx()` which is the right-hand side of equation 3.1 ($x^3 + a_4x + a_6$).

Listing 13.5 Polynomial elliptic curve right-hand side

```
void poly_fofx(POLY *f, POLY x, POLY_CURVE E)
{
  POLY t1, t2;
```

```
  poly_init(&t1);
  poly_init(&t2);
  poly_mul(&t1, x, x);          | t1 = x^3
  poly_mul(&t1, t1, x);         |
  poly_mul(&t2, E.a4, x);       <- t2 = x a4
  poly_add(f, t1, t2);          | f= x^3 + x a4 + a6
  poly_add(f, *f, E.a6);        |
  poly_clear(&t1);
  poly_clear(&t2);
}
```

Listing 13.6 shows the new routine to increment an x value. The problem I found is that the coefficients will just cycle because they are modulo a prime. When a coefficient rolls over I want to then increment the next coefficient. This goes all the way up the chain to the maximum possible degree. The only time this really happens is for very small prime fields.

To know the limit of how far I can increment, the irreducible polynomial is retrieved. The input polynomial is assumed to be of small enough degree that its size can be increased. If a coefficient at index `i` rolls over then `i` is incremented. If index `i` is larger than the input degree then the degree of x is incremented as well but only if the degree is less than the prime polynomial.

Listing 13.6 Polynomial elliptic curve finite field increment

```
void FF_bump(POLY *x)
{
  int i;
  mpz_t one;
  POLY ird;

  mpz_init_set_ui(one, 1);        | create constant 1
  poly_init(&ird);                |
  poly_irrd_get(&ird);            | get irreducible polynomial
  i = 0;
  while(i < ird.deg)
  {
    madd(x->coef[i], x->coef[i], one);   <- increment i^th coefficient
    if(mpz_cmp_ui(x->coef[i], 0))        | no rollover
      return;                            | then all done
    i++;                                 | next coefficient
    if((i > x->deg) && (x->deg < ird.deg))  | and increase
      x->deg++;                          | degree of x
  }
  mpz_clear(one);
  poly_clear(&ird);
}
```

To embed a polynomial x on a curve I first compute the right-hand side of equation 3.1 using function `fofx()` as shown in listing 13.5. I then check to see if this is a quadratic

residue using `poly_sqr()` (listing 12.9). The variable x is incremented using `FF_bump()` from listing 13.6 until a quadratic residue is found. This first part of the embedding routine is shown in listing 13.7.

Listing 13.7 Polynomial elliptic curve embedding

```
void poly_elptic_embed(POLY_POINT *P1, POLY_POINT *P2, POLY x, POLY_CURVE E)
{
  POLY f;
  int done, i;
  mpz_t tmp;

  poly_init(&f);
  poly_copy(&P1->x, x);   ← work with copy of input
  done = 0;
  while(!done)
  {
    poly_fofx(&f, P1->x, E);   | look for f(x)
    if(poly_sqr(f) > 0)        | which is quadratic residue
      done = 1;
    else
      FF_bump(&(P1->x));   ← it was not so try next value
  }
  poly_copy(&(P2->x), P1->x);   ← two y values at same x
  poly_sqrt(&(P1->y), f);                    y1 = √f
  for(i=0; i<=P1->y.deg; i++)                y2 = −y1
    mneg(P2->y.coef[i], P1->y.coef[i]);   ←
  P2->y.deg = P1->y.deg;
  done = mpz_cmp(P2->y.coef[P2->y.deg], P1->y.coef[P1->y.deg]);
  if(done < 0)
  {
    mpz_init(tmp);
    for(i=0; i<=P1->y.deg; i++)
    {
      mpz_set(tmp, P1->y.coef[i]);
      mpz_set(P1->y.coef[i], P2->y.coef[i]);
      mpz_set(P2->y.coef[i], tmp);
    }
    mpz_clear(tmp);
  }
  poly_clear(&f);
}
```

smallest leading y coefficient becomes y_1

The second part of the routine is not required. The idea is to set the first point with "smaller" y value which is determined by the leading coefficient. This really is arbitrary since a smaller positive value means a larger negative one over a prime field. Doing this helped keep track of things while debugging.

13.2.4 Polynomial curve random point

In this section I show how a random polynomial is converted into a point. This will be used in chapter 18 for several protocols.

Choosing random points is straightforward once we have an embedding routine. Listing 13.8 shows the routine. Since there are two points to choose from the last bit from the lowest coefficient of the returned random polynomial is used to determine which of the two is actually used.

Listing 13.8 Polynomial elliptic curve random point

```
void poly_point_rand(POLY_POINT *P, POLY_CURVE E)
{
  POLY r;
  POLY_POINT mP;

  poly_init(&r);
  poly_rand(&r);      ← create random polynomial modulo prime polynomial
  poly_point_init(&mP);  ← dummy point
  if(mpz_tstbit(r.coef[0], 0))          | last bit set
    poly_elptic_embed(P, &mP, r, E);    | return first point
  else                                  | otherwise
    poly_elptic_embed(&mP, P, r, E);    | second point
  poly_clear(&r);
  poly_point_clear(&mP);
}
```

13.2.5 Polynomial elliptic curve addition

In this section I show how point addition is computed on a field extension curve.

Adding two points over a field extension curve uses the same formulas 3.2 through 3.4 with each value becoming a polynomial modulo the irreducible polynomial. Here are the formulas

$$\begin{aligned} \lambda &= \frac{x_1^2 + x_1 x_2 + x_2^2 + a_4}{y_1 + y_2} \\ x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda (x_1 - x_3) - y_1 \end{aligned} \tag{13.4}$$

Compared with the routines from chapter 3 the code expands a bit because each polynomial variable has to be initialized and removed from the stack separately.

Listing 13.9 shows the head of the point addition routine. The first thing done is to check if either input point is the point at infinity and to return the other point if so.

Listing 13.9 Polynomial elliptic curve addition zero check

```
void poly_elptic_sum(POLY_POINT *R, POLY_POINT P, POLY_POINT Q, POLY_CURVE E)
{
  POLY lmbda, ltp, lbt, t1, t2, t3;
  POLY_POINT rslt;
```

```
                                  see if either input
if(poly_test_point(P))            is point at infinity
{
  poly_point_copy(R, Q);          first point at infinity
  return;                         return second point
}
if(poly_test_point(Q))
{
  poly_point_copy(R, P);          second point at infinity
  return;                         return first point
}
```

There is still a check if the slope λ is infinite to determine if the result really is the point at infinity or a different form for λ should be used. This portion is shown in listing 13.10. If $y_1 + y_2$ is zero we need to check if $x_1 - x_2$ is zero. Only under those two conditions being true do we return the point at infinity. Then of course clean up the stack.

Listing 13.10 Polynomial elliptic curve addition slope calculation

```
poly_init(&t1);
poly_init(&t2);                   initialize all
poly_init(&t3);                   temporary variables
poly_init(&ltp);
poly_init(&lbt);
poly_init(&lmbda);
poly_mul(&t1, P.x, P.x);          compute lambda
poly_mul(&t2, P.x, Q.x);          using general form
poly_mul(&t3, Q.x, Q.x);
poly_add(&ltp, t1, t2);           numerator =
poly_add(&ltp, ltp, t3);          x1^2 + x1*x2 + x2^2 + a4
poly_add(&ltp, ltp, E.a4);
poly_add(&lbt, P.y, Q.y);   <- denominator = y1 + y2

if(!lbt.deg && !mpz_cmp_ui(lbt.coef[0], 0))      <- if(y1 = -y2)
{                                                   denominator is zero
  poly_sub(&lbt, Q.x, P.x);
  if(!lbt.deg && !mpz_cmp_ui(lbt.coef[0], 0)) <- Really P == -Q?
  {
    R->x.deg = 0;                 <- x1 == x2
    mpz_set_ui(R->x.coef[0], 0);
    R->y.deg = 0;                 return point at
    mpz_set_ui(R->y.coef[0], 0);  infinity
    poly_clear(&t1);
    poly_clear(&t2);
    poly_clear(&t3);              clean up
    poly_clear(&ltp);             stack
    poly_clear(&lbt);
    poly_clear(&lmbda);
    return;
  }
```

```
    poly_sub(&ltp, Q.y, P.y);
  }
```

The last chunk of the routine computes formulas 13.4 for x_3 and y_3 then cleans up the temporary variables as shown in listing 13.11.

Listing 13.11 Polynomial elliptic curve addition result point

```
                finally, compute
                resulting point
  poly_div(&lmbda, ltp, lbt); ← λ = correct version
  poly_point_init(&rslt);
  poly_mul(&t1, lmbda, lmbda);
  poly_add(&t2, P.x, Q.x);          x3 = λ^2 − (x1 + x2)
  poly_sub(&rslt.x, t1, t2);
  poly_sub(&t1, P.x, rslt.x);
  poly_mul(&t2, t1, lmbda);
  poly_sub(&rslt.y, t2, P.y);       y3 = (x1 − x3)λ − y1
  poly_point_copy(R, rslt);  ← copy to output

  poly_clear(&t1);
  poly_clear(&t2);
  poly_clear(&t3);
  poly_clear(&ltp);
  poly_clear(&lbt);          clean up stack
  poly_clear(&lmbda);
  poly_point_clear(&rslt);
}
```

13.2.6 Polynomial elliptic curve point multiplication

In this section code to compute point multiplication over field extension curves is described.

The last routine in the chapter does look the same as its counterpart in chapter 3. The only difference are the calls to polynomial routines instead of modular math routines. The double and add method still applies, and a local result is copied to the destination at the end.

Listing 13.12 Polynomial elliptic curve multiplication

```
void poly_elptic_mul(POLY_POINT *Q, POLY_POINT P, mpz_t k, POLY_CURVE E)
{
  int bit, j;
  POLY_POINT R;

  poly_point_init(&R);
  poly_point_copy(&R, P);   ← allow in place operation
  j = mpz_sizeinbase(k, 2) - 2;  ← number of bits to go
  while(j >= 0)
  {
```

```
    poly_elptic_sum(&R, R, R, E);   ← double at every bit position
    bit = mpz_tstbit(k, j);
    if(bit)                          | if this bit set
      poly_elptic_sum(&R, R, P, E);  | add in original
    j--;                             | point
  }
  poly_point_copy(Q, R);   | copy to output
  poly_point_clear(&R);    | clean up stack
}
```

13.3 Tiny example

In this section example code for a curve with small field prime value is shown.

In the next few chapters I will cover how to find the cardinality of a field extension as well as how to find the order of a point on a curve over a finite field. I chose the following example using a field prime $p = 43$ with a specific curve that had cardinality of 55 with an embedding degree of 2. This was a very rare curve. This example will be used in chapters 15 and 16 on pairings, so there are a few subroutines used in the following listings which will be covered later. The program name is weil_6_bit_pairing.c.

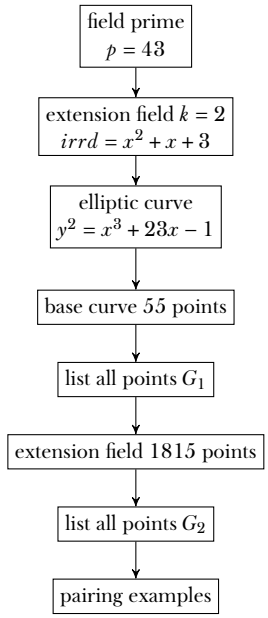

Figure 13.2 Tiny example overall description

Figure 13.2 shows the procedure followed in the tiny example. The field prime determines what irreducible trinomial will work for the tiny extension field. Both the field prime and irreducible polynomial determine the order of the elliptic curve. The "large prime" for the tiny example base curve is 11. The group structure of the extension field is 11×165. The pairing examples appear in chapter 15.

13.3.1 Tiny example variables

This section describes the initialization code for the tiny example.

The start of the tiny example lists the includes for points and curves of both G_1 and G_2 types. It also creates space for all the variables on the stack, many will be used in later chapters. Listing 13.13 is the start of the program.

Listing 13.13 Tiny example startup

```
#include "poly_eliptic.h"
#include "eliptic.h"
#include "pairing.h"

#define M 43

int main(int argc, char *argv[])
{
  FILE *pnts;
  POLY_CURVE Ex;          ← extension
  CURVE E;                ← base
  POLY_POINT Px1, Px2, Qx, Tx;   ← extension
  POINT P1, P2;                  ← base
  mpz_t prm, x, ordr, factors[8], tor;
  POLY irrd, xtnd, t1, t2, t3, t4;
  int i, j, k, m, which, skip;

  mpz_init_set_ui(prm, M);   | initialize
  minit(prm);                | field prime
  poly_init(&irrd);
  if(poly_irreducible(&irrd, 2))     | find irreducible polynomial
    poly_printf("Found irreducible polynomial:\n", irrd);
  else
    printf("no irreducible polynomial found...\n");
  poly_irrd_set(irrd);   | set up prime polynomial
  poly_mulprep(irrd);    | and multiplication table
```

`POINT`s `P1, P2` and `POLY_POINT`s `Px1, Px2` along with `CURVE E` and `POLY_CURVE Ex` will be used to find all the points on the base curve and the field extension curve. The prime field is initialized to 43 using the variable `prm` and then an irreducible polynomial is found. If that failed the program should really just exit. So far in all my tests `poly_irreducible()` has never failed to find an irreducible polynomial. Once the irreducible polynomial is found the multiplication table is created for all the polynomial multiplication routines.

Since none of this is random the output is always the same as shown here:

```
Found irreducible polynomial:
Mod(1, 43)*x^2 + Mod(1, 43)*x^1 + Mod(3, 43)
```

Which is x^2+x+3. This means all polynomials will be degree 1 (or zero) for the x and y values.

13.3.2 Tiny example base curve

This section describes the base curve of the tiny example in detail.

Computing the points on the G_1 group comes from the curve I found using a PARI search. The curve is

$$y^2 = x^3 + 23x + 42 \bmod 43.$$

This curve is plotted in figure 3.6. This curve was chosen because it has a very small number of points and a field extension that is also very small. A duplicate of figure 3.6 is shown in figure 13.3.

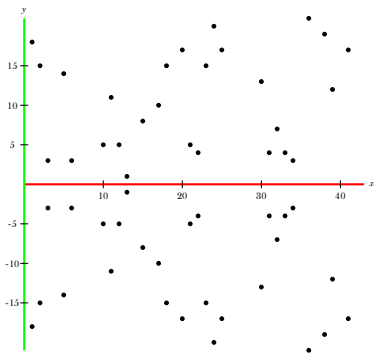

Figure 13.3 All points on tiny elliptic curve $y^2 = x^3 + 23x - 1$ mod 43

The initialization for that curve is listed here:

```
pnts = fopen("all_points_prm_43.dat", "w"); ← save points to file
curve_init(&E);
mpz_set_ui(E.a4, 23);
mpz_set_ui(E.a6, 42);
```

initialize curve to $y^2 = x^3 + 23x + 42$

A file to hold all the points is also created.

The search for points is in listing 13.14. I start with two counters. Variable `i` is used to keep track of the x value. Since `eliptic_embed()` increments x until it finds a value on the curve, I have to update `i` at the end of the loop to match the new x value.

Listing 13.14 Tiny example base points

```
i = 1;      variable i becomes x
j = 0;      variable j is point index
while(i<M-1)
{
  mpz_set_ui(x, i);
  elptic_embed(&P1, &P2, x, E);     find next available x value
  get_order(ordr, P1, E, factors, 3);  ← find order of point
  gmp_fprintf(pnts, "%2d: (%Zd, %Zd) order: %Zd\n", j, P1.x, P1.y, ordr);
  j++;
  gmp_fprintf(pnts, "%2d: (%Zd, %Zd)\n", j, P2.x, P2.y);
                                        save both points to disk
```

```
    j++;
    while(mpz_cmp_ui(P1.x, i) >= 0) i++; ← increment i until 1 past x
  }
  fprintf(pnts, "\n");
```

The variable `j` counts each point. The two points are printed to the file along with the order of the point. The subroutine `get_order()` is a simple brute force operation which I'll explain in chapter 16.

The reason I use variable `i` instead of `P1.x` directly is because the while loop requires an integer.

Sample output from the file looks like this:

```
0: (1, 18) order: 5
1: (1, 25)
2: (2, 15) order: 55
3: (2, 28)
4: (3, 3) order: 11
5: (3, 40)
6: (5, 14) order: 55
7: (5, 29)
8: (6, 3) order: 55
9: (6, 40)
10: (10, 5) order: 55
11: (10, 38)
12: (11, 11) order: 11
13: (11, 32)
14: (12, 5) order: 55
15: (12, 38)
16: (13, 1) order: 55
17: (13, 42)
```

There are 54 points in G_1 listed in the file. The order is 55 because the point at infinity is included in the size of the group. This is true of each group. There are 4 points of order 5, 10 points of order 11 and 40 points of order 55 in the list. All the points of order 55 eventually become points of order 5 or 11. That is one reason to always choose a base point of prime order. For these examples I will choose points of order 11 because they are prime order and will not mix with other points of other orders.

13.3.3 Tiny example field extension curve

This section describes the field extension points for the tiny example.

The field extension curve uses the same coefficients as the base curve, but they are polynomials. Here is the code that sets up the curve for the group 2 points:

```
poly_curve_init(&Ex);
mpz_set_ui(Ex.a4.coef[0], 23);
mpz_set_ui(Ex.a6.coef[0], 42);
```

We see from listing 13.2 that the degree of the coefficients is set to zero during initialization. So these values are polynomial constants.

As a first step, let's see how many points we expect on the field extension curve. From formula 13.3 we have $t_0 = 2$, $t_1 = -11$, so

$$t_2 = t_1 \times t_1 - 43 \times t_0 = -11 \times (-11) - 43 \times 2 = 35.$$

This gives us (from equation 13.2)

$$\#E_2 = 43^2 + 1 - 35 = 1815.$$

From listing 13.7 two points are returned from the `poly_embed()` routine. The input x value is now a polynomial which I call `xtnd`. Here is the initialization:

```
poly_point_init(&Px1);
poly_point_init(&Px2);
poly_init(&xtnd);     ← start at x = 0
j = 0;
```

The variable `j` is again tracking the number of points. As we just computed, the cardinality of the curve is 1815. The j counter limit is one less than that because the point at infinity is part of the cardinality, but not on the curve. The variable j is incremented after each point found is saved to the file.

Listing 13.15 has one line that embeds the next x on the extension curve. As shown in section 13.2.3 this skips past gaps as in figure 13.3, but in two dimensions. The next line finds the order of the point. Routine `poly_get_order()`is described in chapter 16. The next block of lines prints the points in a nice readable format. The output is specific to this tiny example and not general. The last two lines update the `xtnd` variable to one past the x value of the point saved using routine `FF_bump()` described in listing 13.6.

Listing 13.15 Tiny example field extension points

```
while(j < 1814)   ← known number of points
{
  poly_elptic_embed(&Px1, &Px2, xtnd, Ex);   ← embed next value of x onto curve
  poly_get_order(ordr, Px1, Ex, factors, 8);   ← find order of point
  gmp_fprintf(pnts, "%2d: x = ", j);    ← save index to file
  if(Px1.x.deg)
    gmp_fprintf(pnts, "%Zd*x + ", Px1.x.coef[1]);
  gmp_fprintf(pnts, "%Zd  y = ", Px1.x.coef[0]);      save x and y
  if(Px1.y.deg)                                       values to file
    gmp_fprintf(pnts, "%Zd*x + ", Px1.y.coef[1]);
  gmp_fprintf(pnts, "%Zd  order: %Zd | %ld\n", Px1.y.coef[0], ordr, g1g2(Px1));
  j++;
  gmp_fprintf(pnts, "%2d: x = ", j);
  if(Px2.x.deg)
    gmp_fprintf(pnts, "%Zd*x + ", Px2.x.coef[1]);     repeat
  gmp_fprintf(pnts, "%Zd  y = ", Px2.x.coef[0]);      for
  if(Px2.y.deg)                                       second
    gmp_fprintf(pnts, "%Zd*x + ", Px2.y.coef[1]);     point
  gmp_fprintf(pnts, "%Zd | %ld\n", Px2.y.coef[0], g1g2(Px2));
```

```
    j++;
    poly_copy(&xtnd, Px1.x);
    FF_bump(&xtnd);      ← increment polynomial x
  }
  fclose(pnts);
```

The routine `g1g2()` looks at the degree of the x and y polynomials. If both are zero it returns 1, if degree of x is 1 and y is zero it returns 2, if x degree is 0 and y is 1 it returns 3 and if both degrees are 1 it returns 4. This was more for curiosity than useful.

The output from this search for points has many G_1 points to start with:

```
 0: x = 0  y = 4*x + 2  order: 11 | 3
 1: x = 0  y = 39*x + 41 | 3
 2: x = 1  y = 18  order: 5 | 1
 3: x = 1  y = 25 | 1
 4: x = 2  y = 15  order: 55 | 1
 5: x = 2  y = 28 | 1
 6: x = 3  y = 3  order: 11 | 1
 7: x = 3  y = 40 | 1
 8: x = 4  y = 10*x + 5  order: 33 | 3
 9: x = 4  y = 33*x + 38 | 3
10: x = 5  y = 14  order: 55 | 1
11: x = 5  y = 29 | 1
12: x = 6  y = 3  order: 55 | 1
13: x = 6  y = 40 | 1
14: x = 7  y = 6*x + 3  order: 33 | 3
15: x = 7  y = 37*x + 40 | 3
```

But in addition there are many G_2 points as well. Every constant x value (i.e. degree zero value) has a point on the curve. When x becomes a degree 1 polynomial then gaps begin to appear:

```
276: x = 6*x + 3  y = 2*x + 16  order: 11 | 4
277: x = 6*x + 3  y = 41*x + 27 | 4
278: x = 6*x + 6  y = 19*x + 21  order: 165 | 4
279: x = 6*x + 6  y = 24*x + 22 | 4
280: x = 6*x + 8  y = 16*x + 32  order: 55 | 4
281: x = 6*x + 8  y = 27*x + 11 | 4
282: x = 6*x + 10  y = 20*x + 40  order: 55 | 4
283: x = 6*x + 10  y = 23*x + 3 | 4
284: x = 6*x + 12  y = 17*x + 19  order: 165 | 4
285: x = 6*x + 12  y = 26*x + 24 | 4
286: x = 6*x + 14  y = 6*x + 39  order: 165 | 4
287: x = 6*x + 14  y = 37*x + 4 | 4
288: x = 6*x + 17  y = 2*x + 38  order: 165 | 4
289: x = 6*x + 17  y = 41*x + 5 | 4
```

The purpose of this example is to show that a field extension of a curve has all the points of the base field plus a lot more. This becomes very useful as we get into pairing calculations.

Unfortunately attempting to graph this tiny example on paper is challenging. There are two dimensions for the x value and two dimensions for the y value, so the graph sits on a four dimensional plane. Field extensions for cryptographic purposes require 11 to 31 dimensions. Using graphs like figure 13.3 for the base curve or graphs like figure 13.4 (both copied from chapter 3) allow us to imagine what is happening with the mathematics. The formulas are the same, so this abstract connection is perfectly valid.

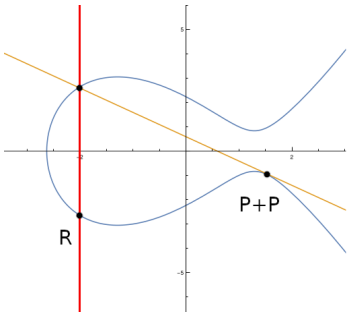

Figure 13.4 General shape of elliptic curve to help with field extension abstraction

13.4 Summary

- An elliptic curve over a finite field has cardinality $\#E = h \cdot r$ with r a large prime. That curve has field extension k when

$$p^k \cong 1 \bmod r.$$

  For elliptic curve pairings we want k to be small.

- The cardinality of a field extension curve is

$$\#E_k = p^k + 1 - t_k$$

  with t_k found from the recurrence relation

$$t_{n+2} = t_1 t_{n+1} - p t_n.$$

  This is used to find cofactors of field extension curves.

- Points on a field prime curve are labeled G_1. Points on an extension curve are labeled G_2. A field extension curve has the same coefficients as a field prime curve, but they are constructed as zero degree polynomials.

- Testing for a polynomial point at infinity uses the degree of the polynomials as well as lowest coefficient to determine if both x and y are zero. This is used to determine the order of a point on a field extension curve.

- Embedding a polynomial point on a polynomial curve requires a "bump" routine that increments the next coefficient. Only used for small test cases since large coefficients will never roll over.

- The polynomial random point function embeds a random polynomial on a polynomial curve. This is used to find reference points for pairing calculations.
- Polynomial elliptic curve addition uses the same formulas as field prime elliptic curve point addition. It just calls polynomial functions to process the math. These routines are used in aggregated signature and zero knowledge algorithms.
- Polynomial elliptic curve multiplication is also the same as field prime elliptic curve multiplication. The double and add formula is exactly identical. These routines are used in aggregated signature and zero knowledge algorithms.
- A tiny example shows how G_1 and G_2 points are on the same curve. A field extension has both G_1 and G_2 points. A prime field curve can only see G_1 points.

Chapter Bibliography

Freeman, David, Scott, Michael, & Teske, Edlyn. 2006. *A taxonomy of pairing-friendly elliptic curves*. Cryptology ePrint Archive, Paper 2006/372. https://eprint.iacr.org/2006/372.

Lidl, Rudolf, & Niederreiter, Harald. 1997. *Finite Fields*. 2 edn. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York.

13.5 Answer to exercise

- 13.1) Embedding degree = 4 because $41^4 = 1 \bmod 29$.

14 Finding low embedding degree elliptic curves

This chapter covers

- Algorithms for finding low embedding degree field extensions.
- The j-invariant and Hilbert class polynomials.
- The method of complex multiplication, used to find elliptic curves.
- Routines to find secure curves with low embedding degree.

In this chapter I explain the reasons for choosing specific size extension fields to create a secure pairing friendly curve. The optimal size depends on the security level desired. Too small and it might be broken, too large and it may not be efficient for practical use.

I then discuss algorithms which can find curves of low embedding degree. This involves searching for primes using mathematical functions. The parameters found during this process will allow us to use another mathematical function to find curves which have the prime number of points found for pairings.

The method used to find the curves of interest is called complex multiplication. One of the parameters from the search process will tell us which function we use to find the curve coefficients. The factoring of that function is the last step in theory before diving

into code.

14.1 Security of field extensions for elliptic curve pairing

In this section I cover the security requirements of pairings over elliptic curve field extensions. Knowing the security level you want drives the field sizes and embedding degrees.

Finding the private key from the public key and base point is called the elliptic curve discrete log problem. For curves over a prime field the difficulty comes from the size of the prime and goes as the square root of the order of the base point. So an equivalent symmetric key is half the size of an elliptic curve key. The elliptic curve is still an exponential security level.

Field extensions can be attacked with algorithms similar to factoring. These are subexponential, so the number of bits required for security grows more rapidly. This becomes a problem for pairing applications because both the base curve and the field extension must be secure at the same level.

Table 14.1 is taken directly from reference (Freeman *et al.*, 2006). The first column is the number of bits for a symmetric key system such as AES. The second column is the number of bits required for an equivalent security level for key exchange using elliptic curves over a prime field as in chapter 3. The third column is the number of bits required for the same level of security for an elliptic curve over a field extension as in chapter 13. The last column is the embedding degree required to go from the field in column two to the extension field in column three. It is derived by dividing column three by column two.

Table 14.1 Elliptic curve field sizes for pairings

Security level	Subgroup size	Extension field	Embedding degree
80	160	960 - 1280	6 - 8
112	224	2200 - 3600	10 - 16
128	256	3000 - 5000	12 - 20
192	384	8000 - 10000	20 - 26
256	512	14000 - 18000	28 - 36

What is not shown in table 14.1 is the size of the prime field. The column headed subgroup size indicates the number of bits in a large prime factor within the cardinality usually called r. The ratio of the number of bits in the field prime (p) to the number of bits in the subgroup size is called ρ. In formula form

$$\rho = \frac{log_2(p)}{log_2(r)}.$$

One of the early descriptions of pairing friendly curves which has gained wide use is called BLS curves (Barreto *et al.*, 2003). These curves have a ρ value near 1.5 while the curves from (Freeman *et al.*, 2006) are less than 1.1 for embedding degrees above 19. For security levels above 128 bits this is quite a large savings in resources, thus lower cost for the same level of security.

In the preface and about this book, I describe how one of the primary goals of this book is to ensure the best level of security rather than high speed or efficiency. I choose embedding degrees of prime value for that reason. The choice of embedding degree is then 7, 11, 13, 17, 19, 23, 29 or 31 for the security levels shown in table 14.1. Reference (Freeman *et al.*, 2006) gives many algorithms for any embedding degree, but I will use only two of them applicable to these primes.

The methods of reference (Freeman *et al.*, 2006) determine a field prime and a large prime order for a given embedding degree along with the cardinality of the base curve. They then call on the method of complex multiplication (CM) to find a curve that fits the parameters. It is at this point that the mathematical rabbit hole gets very, very deep (see chapter II in (Silverman, 2013)).

In the following sections I describe the mathematics at a functional level. Two formulas from reference (Freeman *et al.*, 2006) each apply to some of the primes in table 14.1. Then I give a very simplified description of how the CM method works. References for deeper understanding include (Blake *et al.*, 1999), (Cohen, 2000) and (Silverman, 2013).

As part of the trip down the rabbit hole we are going to run across two terms that you may have run across before. The first is the discriminant of a quadratic equation. This discriminant really is similar to what you ran across in high school solving quadratic equations. The quadratic equation is

$$Dy^2 = 4q - t^2 \tag{14.1}$$

where q is the field prime, t is the trace of Frobenius, y is being solved for and D is the discriminant. For details on where this formula comes from see (Freeman *et al.*, 2006).

The second term is called the j-invariant. This is the last value in the list from chapter 6 from PARI's elliptic curve parameters. Elliptic curves which have the same j-invariant are isomorphic which means they have the same number of points. We can find curve coefficients a_4 and a_6 once we know the value of a j-invariant.

The method of complex multiplication is a search process seeking prime values of q and r with known values of D. Using q and D we can find the roots of a polynomial which will give us the j-invariant of an elliptic curve with those values of q, r and t. As part of the trip down the rabbit hole keep track of the relationships $\#E = q + 1 - t = h \cdot r$.

Exercise 14.1

Define the meaning of the variables $\#E$, q, t, h, and r in the previous sentence.

14.2 Low embedding degree

In this section I describe how low embedding degree field extensions are found.

In chapter 13 I pointed out that the typical embedding degree for average curves with large prime r as a factor in the cardinality normally have embedding degree the same size as r. This is many orders of magnitude too large for pairing where we only need an embedding degree in the six bit range. Mathematicians have spent a lot of time and effort developing methods which solve this problem. In this section I discuss just two of the many

methods which have been developed.

Figure 14.1 shows the algorithm which will be described for finding pairing friendly curves. Choosing from the list of primes determines which set of formulas we pick from reference (Freeman *et al.*, 2006). A sweep over the inputs to those functions will eventually find a prime value of r. If q is not also prime the sweep continues. The red line is a dividing point in my software implementation. Above the line is a sweep program, below the line is the curve finding program.

The curve finding program uses the α value and q to factor a polynomial. Those factors are the j-invariant for the curve we seek. Once the j-invariant values are known the curve parameters a_4 and a_6 can be computed.

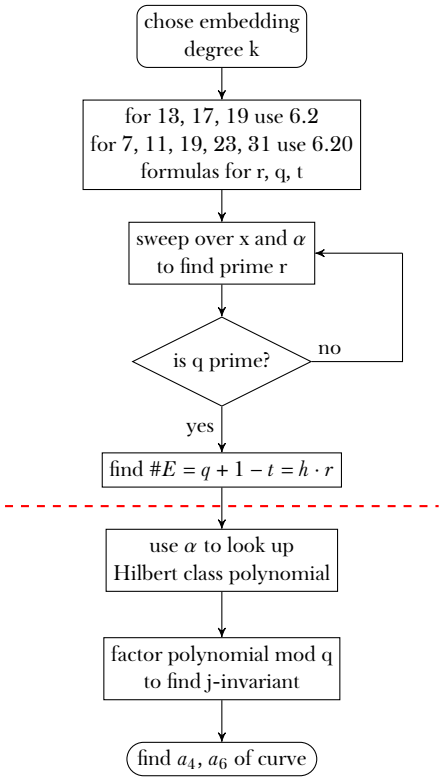

Figure 14.1 Pairing friendly curves algorithm. Above the dashed line is one program which searches for good parameters. Below the dashed line is a program that finds one curve for one set of parameters.

Reference (Freeman *et al.*, 2006) goes through many other methods listing a table for each embedding degree value and which algorithm gives the lowest value of ρ. I am actually going to combine two construction algorithms with Theorem 6.19 from (Freeman *et al.*, 2006) for a simplified description of the algorithms.

The algorithms to find curves with low embedding degree use functions for a large prime factor common to both G_1 and G_2 called $r(x)$, the trace of Frobenius $t(x)$ and the

field prime $q(x)$ for a specific embedding degree k. Theorem 6.19 in (Freeman *et al.*, 2006) says we can replace x^2 in all these functions by $z = \alpha x^2$ where α is a square free value. It turns out this value is the discriminant in equation 14.1. For the two algorithms of interest here, there are specific values of α which will create useful curves.

Table 14.2 Low embedding algorithms

construction	6.2	6.20
k	13, 17, 29	7, 11, 19, 23, 31
$r(z)$	$z^{k-1} - z^{k-2} + \cdots - z + 1$	$z^{k-1} - z^{k-2} + \cdots - z + 1$
$t(z)$	$1 - z$	$1 + z^{(k+1)/2}$
$q(z)$	$(z^{k+2} + 2z^{k+1} + z^k + z^2 - 2z + 1)/4$	$(z^{k+1} + z^k + 4z^{(k+1)/2} + z + 1)/4$

The first line of table 14.2 shows the k values allowed for the embedding degree formulas which I will use. The allowed values for α in $z = \alpha x^2$ are

$$\alpha = 7, 11, 15, 19, 23 + a \times 20 \text{ for } a \in \{0, 1, \cdots, 7\}$$

and x is any value.

These formulas require a great deal of testing. Sweeps in both α and x are required to determine if both $r(z)$ and $q(z)$ are primes because most are not. Once a triplet of (r, t, q) is found the next step is to find a curve which fits that triplet.

14.3 Complex multiplication

In this section I show how finding a curve which meets the low embedding degree parameters is found. After choosing security parameters, these methods give a curve which satisfies those parameters.

This is the place where the theory of elliptic curves gets really deep. It is fun to dive into the details of the mathematics, but it is not essential to creating pairing friendly curves. In this section I lay down the rules of mathematics used to solve the problem without explaining where those rules come from.

Taking the value of the discriminant, we look up a formula whose roots are the j-invariant. Once we know a j-invariant value we can write down the equation of the curve as

$$y^2 = x^3 + 3cx + 2c \tag{14.2}$$

where

$$c = \frac{j}{1728 - j}. \tag{14.3}$$

See (Blake *et al.*, 1999) section VIII.2 for more details.

The j-invariant can be computed from roots of the Hilbert-Class polynomial modulo the prime q. Rather than attempt to explain what that means, we are going to just use PARI to give us the polynomials for each of the α values listed above. The connection between the complex plane, the j-invariant modular function and integer results is mind-bending,

and I urge you to read as much as you can deal with. For example, the largest α in the list is 163. In section 7.2.3 of (Cohen, 2000) he shows that $e^{\pi\sqrt{163}}$ is almost an integer within 12 decimal places.

Appendix B lists the code which prints out the Hilbert polynomials along with the output file itself. Since the roots of these polynomials find a j-invariant and the formulas above give curves based on the j-invariant value, we should be done right? No, we have one more problem called the twist curve. This comes from the value of t in table 14.2 having the possibility of being positive or negative in the cardinality formula 6.2 ($\#E = p + 1 - t$).

For polynomials with several roots we can keep trying curves until we find the one with cardinality factors we desire. For Hilbert polynomials with one root we only have a 50% chance of getting the correct curve. When we don't get the right curve we use the method described in (Blake *et al.*, 1999) section VIII.2 which picks a quadratic non-residue f and changes equation 14.2 to

$$y^2 = x^3 + 3cf^2x + 2cf^3 \tag{14.4}$$

The method of determining the correct curve is to choose a random point on the curve and then multiply by the desired cardinality. If we get the point at infinity it is the curve we want. The twist curve will not result in the point at infinity.

14.3.1 Factoring Hilbert class polynomial

In this section I explain what a Hilbert class polynomial looks like and how it can be factored. Each factor gives a j-invariant which can be used to find the curve which has the prime field and large prime factor used to compute pairings. This completes the method of complex multiplication as shown below the dashed line in figure 14.1.

The Hilbert class polynomial has unique factors in the form

$$H(x) = \prod_i (x - j_i) \tag{14.5}$$

with j_i unique for each i. While the formula looks the same for every discriminant the results are different for each modulus. The method to solve for the factors comes from algorithm 1.6.1 in reference (Cohen, 2000) which is diagrammed in figure 14.2. The flow chart is actually part of a recursive algorithm with input $P(x)$ coming from one of the two possible outputs at the end of the algorithm.

Our input is actually a polynomial of the form

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots a_1 x + a_0.$$

The process of factoring breaks this down into two factors which each can be processed again to find smaller factors. At some point we get down to degree 1 or degree 2 polynomials which we can directly solve for the roots.

The first step in figure 14.2 computes the greatest common divisor by first defining $h(x) = x^p \bmod P(x)$ and then computing $\gcd(h(x) - x, P(x))$. The same process occurs in the computation of $B(x)$. If $A(0) = 0$ then there is a root equal to zero and a new $A(x)$ is found by division with x. Done means finished with that root, and we can proceed to check any others.

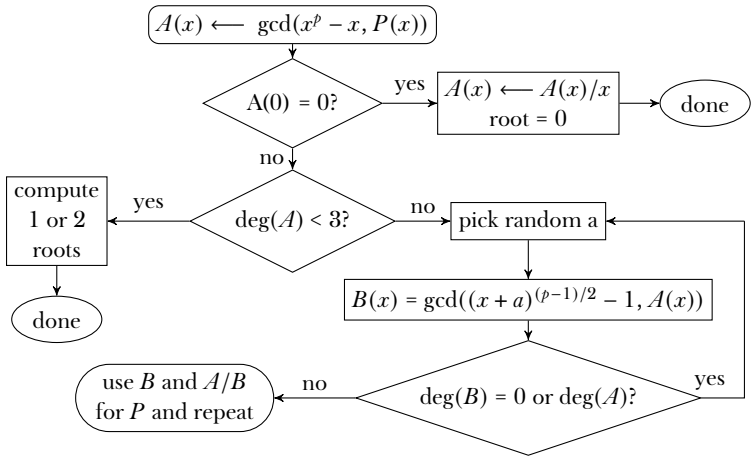

Figure 14.2 Roots mod p of P(x)

If the degree of $A(x)$ is less than three we can directly compute one or two roots. If the degree is 1 then we have

$$x = -\frac{a_0}{a_1}. \tag{14.6}$$

If the degree is two then we can use the quadratic solution using the method described in algorithm 1.6.1 from reference (Cohen, 2000)

$$\begin{aligned} d &\leftarrow a_1^2 - 4a_0a_2 \\ e &\leftarrow \sqrt{d} \\ x_1 &= \frac{-a_1 + e}{2a_2} \\ x_2 &= \frac{-a_1 - e}{2a_2} \end{aligned} \tag{14.7}$$

For the degree of $A(x)$ larger than two the algorithm picks a random value a and computes $h(x) = (x + a)^{(p-1)/2} \bmod A(x)$ to find $B(x) = \gcd(h(x) - 1, A(x))$. If the $\deg(B)$ is zero or the same as $\deg(A)$ we try a different random a. If the value for $\deg(B)$ is acceptable we use $B(x)$ and $A(x)/B(x)$ for the next round.

The actual implementation pushes each polynomial $B(x)$ and $A(x)/B(x)$ on a stack and adds roots to an array. The initial degree of the Hilbert class polynomial determines the number of roots. Every time we hit a done portion we have popped a polynomial off the stack and found roots. In this way all roots are found and the process for finding curves can begin.

14.4 Code for finding pairing friendly curves

In this section I present two programs which help find pairing friendly curves.

- The first program inputs the embedding degree and number of bits in r and sweeps over α and x searching for prime q and prime r.

- The second program takes the first program's output α, q, and t as input to find a curve by solving for the roots of the correct Hilbert class polynomial.

14.4.1 Pairing sweep

In this section I get into the details of sweeping over many possible inputs to equations $r(z)$, $t(z)$, and $q(z)$ from table 14.2. After describing the subroutines for those functions the main program will explain how the inputs are chosen to perform the sweep. Figure 14.3 shows the flow chart for the main program. Remember that $z = \alpha x^2$. The sweep changes α independently from x.

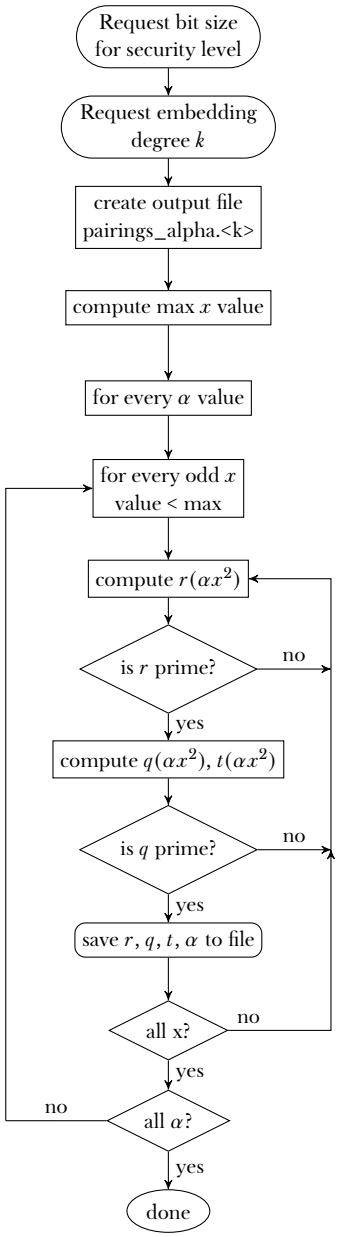

Figure 14.3 Program which sweeps over α to find pairing friendly curves

The sweep over α and x in formulas shown in table 14.2 leads to a maximum size in r of x^{2k-2}. The program input asks for maximum bit size because that is how we think about security. Calling the input `lg2r`, I take the maximum sweep over x to be $2^{\lg 2r/(2k-2)}$. This is a little too big because it does not account for α, but it is close enough to be reasonable.

The opening of program `pairing_sweep_alpha.c` brings in the modulo and polynomial function headers along with `math.h`. I then put in several subroutines to compute $r(z)$, $q(z)$ and $t(z)$. Listing 14.1 shows the top of the program along with the routine that computes $r(z)$.

Listing 14.1 Pairing sweep $r(z)$ routine

```
#include "modulo.h"
#include "poly.h"
#include <math.h>

void phi4k(mpz_t r, long k, mpz_t alpha, mpz_t x)   <- input α and x as separate values; k is embedding degree
{
  int i;
  mpz_t z[36], ck;   <- max embedding degree assumed < 36

  mpz_init_set_ui(ck, 1);
  mpz_init(z[0]);
  mpz_mul(z[0], x, x);
  mpz_mul(z[0], alpha, z[0]);          z[0] = αx^2
  mpz_sub(ck, ck, z[0]);    <- start at 1 − z
  for(i=1; i<k-1; i++)
  {
    mpz_init(z[i]);
    mpz_mul(z[i], z[i - 1], z[0]);     z[i] = z^(i−1)
    if(i & 1)
      mpz_add(ck, ck, z[i]);
    else                               alternate sign between terms
      mpz_sub(ck, ck, z[i]);
  }
  mpz_set(r, ck);   <- output result
  mpz_clear(ck);
  for(i=0; i<k-1; i++)                 clean up stack
    mpz_clear(z[i]);
}
```

The name of the routine comes from the cyclotomic polynomial which defines r. The inputs include the embedding degree k, the discriminant α and the value of x. I found that even x values never result in a prime, so the calling loops include only odd x values.

The purpose of subroutine `phi4k()` is to compute the formula $r(x)$ in table 14.2. The routine starts by computing $z = \alpha x^2$ and then begins with $1 - z$. The array `z[]` holds each power of z, so it is easy to add or subtract the correct power and then compute the next power on each loop. The index zero holds z^1, so the loop ends at $k - 2$ which is z^{k-1}. As seen in the $r(z)$ line of table 14.2 that is the correct final power.

The $q(z)$ routines are similar to each other but as seen in table 14.2 they are different enough to require their own routines. Listing 14.2 shows the code for algorithm 6.20 and listing 14.3 has the code for algorithm 6.2.

Listing 14.2 Pairing sweep $q(z)$ algorithm 6.20

```
void qofz_20(mpz_t q, long k, mpz_t alpha, mpz_t x)
{
  long k1, k2;
  mpz_t  t1, t2, t3, t4, z;

  mpz_inits(z, t1, t2, t3, t4, NULL);
  k1 = k + 1;
  k2 = k1/2;
  mpz_mul(z, x, x);
  mpz_mul(z, z, alpha);
  mpz_pow_ui(t1, z, k1);
  mpz_pow_ui(t2, z, k);
  mpz_pow_ui(t3, z, k2);
  mpz_mul_ui(t3, t3, 4);
  mpz_add_ui(t4, z, 1);
  mpz_add(t1, t1, t2);
  mpz_add(t1, t1, t3);
  mpz_add(t1, t1, t4);
  mpz_fdiv_q_ui(q, t1, 4);
  mpz_clears(z, t1, t2, t3, t4, NULL);
}
```

input α and x
as separate values
k is embedding degree

offsets to
embedding degree

z = αx^2

z^{k+1}

z^k

$4z^{(k+1)/2}$

$$\frac{1+z+4z^{(k+1)/2}+z^k+z^{k+1}}{4}$$

These routines are just straight calculation of the formulas. The exponents are slightly different in each routine. It's interesting that the structure of the formulas are quite similar otherwise.

Listing 14.3 Pairing sweep $q(z)$ algorithm 6.2

```
void qofz_2(mpz_t q, long k, mpz_t alpha, mpz_t x)
{
  long k1, k2;
   mpz_t  t1, t2, t3, t4, z;

  mpz_inits(z, t1, t2, t3, t4, NULL);
  k1 = k + 1;
  k2 = k + 2;
  mpz_mul(z, x, x);
  mpz_mul(z, z, alpha);
  mpz_pow_ui(t1, z, k2);
  mpz_pow_ui(t2, z, k1);
  mpz_mul_ui(t2, t2, 2);
  mpz_pow_ui(t3, z, k);
  mpz_sub_ui(t4, z, 1);
  mpz_mul(t4, t4, t4);
  mpz_add(t1, t1, t2);
  mpz_add(t1, t1, t3);
```

input α and x
as separate values
k is embedding degree

offsets to
embedding degree

z = αx^2

z^{k+2}

$2z^{k+1}$

z^k

$(z-1)^2$

```
  mpz_add(t1, t1, t4);
  mpz_fdiv_q_ui(q, t1, 4);
  mpz_clears(z, t1, t2, t3, t4, NULL);
}
```

$$\frac{z^{k+2} + 2z^{k+1} + z^k + (z-1)^2}{4}$$

Listing 14.4 shows the code for $t(z)$ algorithm 6.20 and listing 14.5 shows $t(z)$ for algorithm 6.2.

Listing 14.4 Pairing sweep $t(z)$ algorithm 6.20

```
                                                          input α and x
                                                          as separate values
                                                          k is embedding degree
void tofz_20(mpz_t t, long k, mpz_t alpha, mpz_t x)   <-
{
  mpz_t z, t1;
  long k1;

  mpz_inits(z, t1, NULL);
  mpz_mul(z, x, x);
  mpz_mul(z, z, alpha);          z = αx^2
  k1 = (k + 1)/2;
  mpz_pow_ui(t1, z, k1);
  mpz_add_ui(t, t1, 1);          t = 1 + z^((k+1)/2)
  mpz_clears(z, t1, NULL);
}
```

These are very simple. As with the other routines α and x are inputs.

Listing 14.5 Pairing sweep $t(z)$ algorithm 6.2

```
                                                          input α and x
                                                          as separate values
                                                          k is embedding degree
void tofz_2(mpz_t t, long k, mpz_t alpha, mpz_t x)    <-
{
  mpz_t z, one;

  mpz_init(z);
  mpz_mul(z, x, x);
  mpz_mul(z, z, alpha);          z = αx^2
  mpz_init_set_ui(one, 1);
  mpz_sub(t, one, z);            t = 1 - z
  mpz_clears(z, one, NULL);
}
```

Listing 14.6 is the beginning of the main routine for sweeping a specific embedding degree. The variables `atab[]` and `ktab[]` are arrays for the α and k values in table 14.2. The array `algt[]` tells which algorithm to use (6.2 or 6.20) for each value in the `ktab[]` array.

The program expects one input. This is the largest number of bits in r desired. The sweep will find many values smaller than this, so over estimating by five or six bits is useful.

Listing 14.6 Pairing sweep main

```
int main(int argc, char *argv[])
{
  FILE *pair;
  mpz_t r, alpha, x, q, t;
  long k, u, lg2r, alphabase, rpm, qpm, rsz, qsz;
  long max, atab[5] = 7, 11, 15, 19, 23;
  double rho;
  int j, m, nmrpm, a, kdex;
  int ktab[9] = 5, 7, 11, 13, 17, 19, 23, 29, 31;
  int algt[9] = 1, 0,  0,  1,  1,  0,  0,  1,  0; // 1 == 6.2  0 == 6.20
  double xsz, asz;
  char filename[128];

  if(argc < 2)
  {
    printf("Use: ./pairing_sweep_alpha  <log2(r) max range>\n");
    exit(-1);
  }
  lg2r = atol(argv[1]);
  if(lg2r < 3)
  {
    printf("need more bits to work with\n");
    exit(-2);
  }
  printf("choose embedding degree k from 5, 7, 11, 13, 17, 19, 23, 29, 31: ");
  fflush(stdout);
  scanf("%ld", &k);
  for(kdex=0; kdex<9; kdex++)
    if(k == ktab[kdex])
      break;
  if(kdex > 8)
  {
    printf("k must be from list.\n");
    exit(-1);
  }
  sprintf(filename, "pairings_alpha.%02ld", k);
  pair = fopen(filename, "w");
  nmrpm = 0;
  mpz_inits(r, alpha, x, q, t, NULL);
  max = exp2(lg2r/2/(k - 1));
```

α table starting values

prime embedding degrees

flag for which algorithm to use

expect number of security bits for input

small to play is ok but not too small!

request embedding degree and check if in table

ignore input if not found

create file for found parameters

← compute largest x

The program asks for an embedding degree after listing the options that are acceptable. After scanning the `ktab[]` array for a match it will exit the program on an incorrect input. For an acceptable choice a new file is created to hold the prime values r, q and all other parameters for the chosen embedding degree.

The variable `nmrpm` is a count of the number of r prime values found. The GMP variables are named according to their use from the formulas.

Listing 14.7 shows the full sweep. The outer loop adds the multiple $20a$ to α. The variable `m` steps through the `atab[]` array. Between these two loops all possible values of α available from the Hilbert class polynomial list is attempted.

Listing 14.7 Pairing sweep loop

```
for(a=0; a<8; a++)
{
  for(m=0; m<5; m++)
  {
    alphabase = atab[m] + a*20;   ← integer version of α
    printf("processing alpha = %ld\n", alphabase);   ← long slow process; let user know program running
    mpz_set_ui(alpha, alphabase);
    for(j=1; j<max; j+=2)
    {                             ← only odd values of x are useful
      mpz_set_ui(x, j);
      phi4k(r, k, alpha, x);      ← both algorithms use same r(x)
      rsz = mpz_sizeinbase(r, 2);   ← is returned r value acceptable size?
      if(rsz > lg2r)
        break;
      rpm = mpz_probab_prime_p(r, 25);   ← is r value prime?
      if(rpm)                     ← yes so increment number r primes found
      {
        nmrpm++;
        if(algt[kdex])
        {
          qofz_2(q, k, alpha, x);
          tofz_2(t, k, alpha, x);
        }
        else                      ← use correct algorithm for this embedding degree
        {
          qofz_20(q, k, alpha, x);
          tofz_20(t, k, alpha, x);
        }
        qsz = mpz_sizeinbase(q, 2);   ← number of bits in q
        qpm = mpz_probab_prime_p(q, 25);   ← is q prime?
        if(qpm)
        {
          gmp_fprintf(pair, "k= %ld alpha = %Zd  x = %Zd\n", k, alpha, x);
          gmp_fprintf(pair, "r = %Zd numbits: %ld\n", r, rsz);
          gmp_fprintf(pair, "q = %Zd  numbits: %ld\n", q, qsz);
          fprintf(pair, "rho = %lf\n", (double)qsz/(double)rsz);
          gmp_fprintf(pair, "t = %Zd\n\n", t);
        }                         ← save all parameters including ρ to file
      }
    }
  }
}
fclose(pair);
```

```
  mpz_clears(r, alpha, x, q, t, NULL);
  printf("found %d r primes\n", nmrpm);
}
```

The variable `j` is converted to the x value used in all the formulas. Only odd values are useful, so the loop increment is 2. Since both algorithms use the same formula for $r(x)$ this is computed first. If the size of `r` exceeds the requested limit the `j` loop is terminated.

The routine `mpz_probab_prime()` checks to see if `r` is prime. The GMP manual says values between 15 and 50 are reasonable for the second argument, so I chose 25. Higher values take longer and lower values run the risk of being wrong since it is probabilistic.

If `r` is a prime then q and t are computed using the correct subroutines for this choice of embedding degree. The value of `q` is also checked to ensure it is in fact a prime. There is no output if q is not prime.

Once good values for r and q are found all the values are saved to the file. Here is an example for embedding degree 13. From table 14.1 the subgroup size of 224 can use embedding degree 13. But when used as input the only value found is much too small. When an input size of 260 is requested then one value is found as shown here:

```
$ ./pairing_sweep_alpha 260
choose embedding degree k from 5, 7, 11, 13, 17, 19, 23, 29, 31: 13
processing alpha = 7
processing alpha = 11
processing alpha = 15
⋮
processing alpha = 151
processing alpha = 155
processing alpha = 159
processing alpha = 163
found 182 r primes
```

Of the 182 r values which were prime, only three had corresponding prime q values. Listing 14.8 is the output from this particular run. Low embedding degrees have higher ρ values, in this case $\rho = 280/226 = 1.2$. For higher security levels the value of ρ drops below 1.1.

Listing 14.8 Output pairing_sweep.13

```
k= 13 alpha = 35  x = 1
r = 3285353271721733941 numbits: 62
q = 38320360561362304687789  numbits: 76
rho = 1.225806
t = -34

k= 13 alpha = 39  x = 3
r = 348698316460683694295470753710l numbits: 102
q = 380205022584152061630276872755726824O1  numbits: 125
rho = 1.225490
t = -350
```

```
k= 13 alpha = 55  x = 91
r = 79679158002503797122797469815576171221524228764283160439578042655821
numbits: 226
q = 18820191050656678907601352196207657317550640639210641977114976509412592
5989359586529  numbits: 280
rho = 1.238938
t = -455454
```

14.4.2 Finding the curve

In this section the code to find a curve using one set of parameters from the sweep program is described.

The prime q values in listing 14.8 are the modulus used for finding the roots of the Hilbert class polynomials listed in appendix B. Each curve will have different roots even though the equation is the same for the same α value. The `get_curve.c` program described in this section takes the α, q and t values from the `pairing_sweep.*` file as inputs. It scans the `Hilbert_Polynomials.list` file for the correct polynomial and then proceeds to find all the factors modulo q. Once we have those factors, we can compute the coefficients for pairing friendly curves.

Since each root is a j-invariant the values for a_4 and a_6 can be computed using formula 14.3. The cardinality is computed using formula 6.2 ($\#E = p + 1 - t$) with the input t value. Multiplying a random point on the curve by the cardinality should give the point at infinity. If not, I keep trying roots until one is found that works. If none work, then the twist of the first curve attempted is computed. This is again tested with a random point, and if that fails the program gives up. The program has never failed to find a curve (so far).

If the degree of the Hilbert class polynomial is 1, the root is just the negative of the zeroth degree coefficient because the Hilbert polynomial is monic. If the degree is two we can use the quadratic formula of equation 14.7 modulo q to find the two roots. For higher degrees we need to break the polynomial down into smaller chunks and peel off 1 and 2 degree factors at a time as shown in figure 14.2.

The program `get_curve.c` starts with a subroutine for computing the quadratic roots modulo q as shown in listing 14.9.

Listing 14.9 Finding curve quadratic roots subroutine

```
#include "poly.h"
#include "eliptic.h"
#include <string.h>

void tworoots(mpz_t j1, mpz_t j2, POLY hc)      ← input polynomial hc
{                                                 outputs roots j1, j2
  mpz_t d, e, f;

  if(hc.deg != 2)   ← just bail out if not degree 2
    return;
  if(mpz_cmp_ui(hc.coef[2], 1))
  {
```

```
    mdiv(hc.coef[1], hc.coef[1], hc.coef[2]);
    mdiv(hc.coef[0], hc.coef[0], hc.coef[2]);
  }
  mpz_inits(d, e, NULL);
  mmul(d, hc.coef[1], hc.coef[1]);
  mpz_init_set_ui(f, 4);
  mmul(e, hc.coef[0], f);
  msub(d, d, e);
  msqrt(e, d);
  mpz_neg(j1, hc.coef[1]);
  mpz_neg(j2, hc.coef[1]);
  madd(j1, j1, e);
  msub(j2, j2, e);
  mpz_set_ui(f, 2);
  mdiv(j1, j1, f);
  mdiv(j2, j2, f);
  mpz_clears(d, e, f, NULL);
}
```

force monic

$d = a^2 - 4b$

$e = \sqrt{d}$

$j_1 = (-a + e)/2$

$j_2 = (-a - e)/2$

If the input polynomial is not degree two the routine immediately bails out. The polynomial is then normalized. With the input polynomial being

$$x^2 + ax + b = 0$$

The subroutine then computes equations 14.7. Note that in formula 14.7 there is division by a_2. By forcing the equation to be monic $a_2 = 1$ so that is ignored in the code.

The `main()` routine of the `get_curve.c` program is in listing 14.10. This checks for proper input and looks for the `Hilbert_Polynomials.list` file. If not found the program exits. After opening the file it reads all of `Hilbert_Polynomials.list` into memory for easy random access.

Listing 14.10 Find curve initialize

```
int main(int argc, char *argv[])
{
  FILE *hilb;
  POLY hc, stack[16], x1, hofx, Aofx, Bofx, rem;
  int i, j, k, alpha, xs, xe, sign, stckp, done;
  mpz_t root[16], p, c, a4[16], a6[16], tp, b, j0;
  mpz_t two, tri, t, crde;
  char *hcdat;
  POINT R, P0;
  CURVE E;

  if(argc < 4)
  {
    printf("Use: ./get_curve <discriminant> <prime> <t>\n");
    printf("   values from output of parings_alpha.k\n");
    exit(-1);
  }
```

check input values are all on command line

```
                                    read in
                                    Hilbert Class Polynomials
hilb = fopen("Hilbert_Polynomials.list", "r");
if(!hilb)                                          ensure Hilbert
{                                                  polynomials file is there
  printf("can't find file Hilbert_Polynomials.list\n");
  exit(-2);
}
hcdat = (char*)malloc(6*1024);
k = 0;
while(!feof(hilb))
{                                                  read in
  hcdat[k] = fgetc(hilb);                          entire file
  k++;                                             to small
}                                                  buffer
fclose(hilb);
k -= 2;
```

Listing 14.11 shows the scan for the discriminant. If the input value is not one of the values used in creating the Hilbert polynomial list the program issues an error and exits.

Listing 14.11 Find curve check discriminant

```
see if discriminant in list
  alpha = atol(argv[1]);
  if(alpha < 0)
    alpha *= -1;          ← technically discriminant is negative
  i = 0;                  ← buffer index
  j = 0;                  ← α value from file
  while((j < alpha) && (i < k))
  {                                                use positive
    sscanf(&hcdat[i], "%d", &j);                   values to make
    j *= -1;                                       search and limit
    if(j == alpha)                                 easier
      break;              ← found correct polynomial
    while(hcdat[i] != '\n')
      i++;                                         skip to
    i++;                                           end of line
  }
  if(j != alpha)                  exit if
  {                               not found
    printf("invalid discriminant %d\n", alpha);
    exit(-3);
  }
```

The discriminant values are actually negative. The output value from program `pairing_sweep_alpha.c` is positive, so I chose to make the comparison with the assumption that the input is positive. If the input is actually negative I make it positive so the compare works. Obviously there are many different ways to do this.

The beginning of each line in the Hilbert_Polynomials.list file holds the α value. Scanning for the newline character and then skipping over the newline puts the index at the start of the next entry. If the index goes past the end of the buffer no match was found and the program issues an error and stops.

Listing 14.12 starts by checking inputs q and t are integers. If not the program exits. If they are, the field prime modulus is initialized. The Hilbert class polynomial is then converted from text to internal representation. The polynomial is monic so the first x has no coefficient in the text file. If there is a ^ after the first x entry the degree of the Hilbert polynomial is set. If not, the degree is set to 1.

The variable `j` then keeps track of the remaining coefficients. The first thing to look for is a + or − sign. After that the end of the coefficient is found with the asterisk which is changed to a zero. The GMP conversion routine then places the coefficient in the right place with binary format. The sign is applied, and the final result is reduced modulo the field prime.

Listing 14.12 Find curve set parameters

```
if(mpz_init_set_str(p, argv[2], 10) < 0)        Set up mod q operations
{                                                 make sure
  printf("invalid prime string\n");               prime modulus
  exit(-4);                                       is an integer
}
minit(p);              ← initialize base field prime

if(mpz_init_set_str(t, argv[3], 10) < 0)        input factor t = p + 1 - #E
{                                                 make sure
  printf("invalid t string\n");                   Frobenius trace
  exit(-5);                                       is an integer
}
read in Hilbert polynomial
poly_init(&hc);
xs = i;                ← xs is coefficient start index
while(hcdat[xs] != 'x') xs++;
if(hcdat[xs + 1] == '^')                        no first coefficient
{                                                 so check for
  xs += 2;                                        degree of
  sscanf(&hcdat[xs], "%ld", &hc.deg);             polynomial
}
else
  hc.deg = 1;      ← special case degree 1
mpz_set_ui(hc.coef[hc.deg], 1);   ← monic polynomial
j = hc.deg - 1;                   for all remaining
while(j >= 0)                     coefficients
{
  while((hcdat[xs] != '+') && (hcdat[xs] != '-'))
    xs++;
```

```
    if(hcdat[xs] == '-')
      sign = -1;
    else                                                    look for sign
      sign = 1;
    xs++;
    xe = xs+1;
    while((hcdat[xe] != '*') && (hcdat[xe] != '\n'))        * marks end
      xe++;                                                 of coefficient
    hcdat[xe] = 0;                                  convert text
    mpz_set_str(hc.coef[j], &hcdat[xs], 10);        to binary
    if(sign < 0)                                    apply sign
      mpz_neg(hc.coef[j], hc.coef[j]);              and ensure modulo field prime
    mpz_mod(hc.coef[j], hc.coef[j], p);
    j--;          continue with next coefficient
  }
```

If the polynomial degree is 1 or 2 we can directly solve for the j-invariant. This is shown in listing 14.13. Degree 1 is trivial, degree 2 calls the subroutine `tworoots()` from listing 14.9.

Listing 14.13 Find curve low degree Hilbert polynomial

```
/*  for order 1 and 2, output result directly  */

  stckp = 0;        stack pointer for polynomials
  if(hc.deg < 3)    degree 1 or 2
  {                 no stack
    if(hc.deg == 1)
    {
      mpz_init_set(root[0], hc.coef[0]);        degree 1 root
      mpz_neg(root[0], root[0]);                from only coefficient
    }
    else
    {
      mpz_inits(root[0], root[1], NULL);        degree 2
      tworoots(root[0], root[1], hc);           subroutine gets both roots
    }
  }
```

For Hilbert class polynomials of degree larger than two the section of code shown in listing 14.14 is executed. The polynomial multiplication table is prepared using the Hilbert class polynomial $\mathrm{hc}(x)$. The polynomial $x^p \bmod \mathrm{hc}(x)$ is computed and then x is subtracted to give $h(x) = x^p - x \bmod \mathrm{hc}(x)$. The polynomial $A(x)$ is then computed as the $\gcd(h(x), hc(x))$. If the result is a constant something is wrong and the program exits.

To create the flow of figure 14.2 the variable $B(x)$ is initialized along with a fixed size stack of polynomials. $A(x)$ is then pushed on the stack.

Listing 14.14 Find curve high degree initialize

```
else                          enter this when
{                             Hilbert polynomial > 3

    find h(x) = x^p mod hc(x)
  poly_mulprep(hc);     ← set up polynomial power table
  poly_init(&x1);
  x1.deg = 1;                          polynomial x1 = x^1
  mpz_set_ui(x1.coef[1], 1);
  poly_init(&hofx);
  poly_xp(&hofx, x1);                  hofx = (x^p mod hc(x)) − x
  poly_sub(&hofx, hofx, x1);

    A(x) = gcd(x^p − x, hc(x))
  poly_init(&Aofx);
  poly_gcd(&Aofx, hc, hofx);           A(x) = gcd(hofx, hc(x))
  if(!Aofx.deg)
  {
      printf("no roots found for this combination:\n");
      poly_print(hc);                  extreme badness
      gmp_printf("prime: %Zd\n", p);   should never happen
      exit(-5);
  }
  poly_init(&Bofx);          initialize stack
  poly_init(&rem);           polynomials

    push first A(x) on stack
  for(i=0; i<16; i++)                  create all
    poly_init(&stack[i]);              possible stack entries
  poly_copy(&stack[stckp], Aofx);  ← push first A(x) on stack

    initialize all roots to zero
  for(i=0; i<hc.deg; i++)              initialize all
    mpz_init(root[i]);                 possible roots
  j = 0;                  ← root index counter
  stckp++;          ← one item on stack
}
```

The meat of the root finding routine is shown in listing 14.15. This implements the algorithm shown in figure 14.2. By pushing and popping polynomials off a stack it is easy to break down each factor until the degree is 1 or 2 and a direct root can be added to the list.

As long as there are polynomials on the stack the loop continues to pull one off. The variable `done` flags when roots have been added to the list and no more polynomials will be added to the stack. A root of zero will continue processing unless $A(x)$ actually is zero. If $A(x)$ has one or two roots they are added to the list and the next item on the stack will be processed.

If the `done` flag is not set the polynomial multiplication table is prepared with $A(x)$.

A random value a is placed in the constant coefficient of `x1` to create $(x + a)$. The value $(x+a)^{(p-1)/2} \bmod A(x)$ is then computed using the `gpow_p2()` routine. The value $B(x) = \gcd((x+a)^{(p-1)/2} - 1, A(x))$ is then found. This is repeated until $B(x)$ has degree smaller than $A(x)$. Then the polynomials $B(x)$ and the quotient from $A(x)/B(x)$ are both pushed on the stack for further processing.

Listing 14.15 Find curve high degree root finding

```
while(stckp)     ← continue until all factors removed from stack
{
  | pop next A(x) off stack
  stckp--;
  poly_copy(&Aofx, stack[stckp]);
  done = 0;
  while(!done)
  {
    if(!Aofx.coef[0])          ← is A(0) = 0?
    {
      j++;  ←| array value zero already                    | if last coefficient
      for(i=1; i<=Aofx.deg; i++)                           | is zero
        mpz_set(Aofx.coef[i - 1], Aofx.coef[i]);           | divide by x
      Aofx.deg--;
      if(!Aofx.deg)
        done = 1;      ← go to next item on stack
    }
    if(Aofx.deg == 1)
    {
      if(!mpz_cmp_ui(Aofx.coef[1], 1))
        mpz_set(root[j], Aofx.coef[0]);   ←| monic single root
      else
        mdiv(root[j], Aofx.coef[0], Aofx.coef[1]);  ←| normal single root
      mneg(root[j], root[j]);
      j++;
      done = 1;      ← go to next item on stack
    }
    else if(Aofx.deg == 2)
    {
      tworoots(root[j+1], root[j], Aofx);  ← degree two, get roots directly
      j += 2;
      done = 1;      ← go to next item on stack
    }
    if(!done)
    {
      Bofx.deg = 0;
      poly_mulprep(Aofx);    ← set up multiplication table
      while(!Bofx.deg || (Bofx.deg == Aofx.deg))
      {
        mrand(x1.coef[0]);
```

```
        gpow_p2(&hofx, x1);                                  for random a
        mpz_sub_ui(hofx.coef[0], hofx.coef[0], 1);           compute
        poly_gcd(&Bofx, hofx, Aofx);                         (x + a)^((p-1)/2) - 1 mod A(x)
      }                                                      until deg(B) < deg(A)
      poly_copy(&stack[stckp], Bofx);      push B(x)
      stckp++;                             on stack
      poly_euclid(&stack[stckp], &rem, Aofx, Bofx);   push A(x)/B(x)
      stckp++;                                        on stack
      done = 1;
    }
  }
  if(stckp > 15) exit(0);   ← max degree 10, so this should never happen
}
```

Once all the roots are found they are printed out as shown in listing 14.16.

Listing 14.16 Find curve output roots

```
for(i=0; i<hc.deg; i++)                          output all
  gmp_printf("%d:  %Zd\n", i, root[i]);          found roots
```

Each root in the array is a j-invariant which can be used in equation 14.3. The values of a_4 and a_6 are then easy to compute from equation 14.2. I create an array of these for each root as shown in listing 14.17. The constants 1728, 2 and 3 are created outside the loop. Inside the loop each a_4 and a_6 coefficient is than added to the appropriate array and printed out.

Listing 14.17 Find curve a_4, a_6 coefficient array

```
mpz_inits(c, tp, b, j0, two, tri, NULL);
mpz_set_ui(j0, 1728);          set up
mpz_set_ui(two, 2);            constants
mpz_set_ui(tri, 3);            1728, 2 and 3
for(i=0; i<hc.deg; i++)        Compute a4 and a6
{                              for each root
  mpz_inits(a4[i], a6[i], NULL);
  mpz_set(tp, root[i]);
  msub(b, j0, tp);             compute c =
  mdiv(c, tp, b);              j/(j - 1728)
  mmul(a4[i], c, tri);         place a4, a6
  mmul(a6[i], c, two);         into respective arrays
  gmp_printf("%d: a4= %Zd  a6= %Zd\n", i, a4[i], a6[i]);
}
```

The next code segment determines which of the roots give the correct curve and which give the twist curve. Listing 14.18 starts by computing the curve cardinality from the input value of t. A curve for every table entry of a_4 and a_6 is created and a random point on that curve is created. If the order of the curve times the random point gives the point at infinity we have the right curve. Otherwise, we have the twist.

Listing 14.18 Find curve twist or correct for each coefficient

```
compute #E * random point
on each curve.
if we get point at infinity
this is the curve we want.
      mpz_init_set(crde, p);
      mpz_add_ui(crde, crde, 1);      compute cardinality
      mpz_sub(crde, crde, t);         #E = p + 1 − t
      gmp_printf("#E = %Zd\n", crde);  from input
      point_init(&R);
      curve_init(&E);                 initialize random point
      point_init(&P0);                curve and
      done = 0;                       test variables
      for(i=0; i<hc.deg; i++)
      {
        mpz_set(E.a4, a4[i]);
        mpz_set(E.a6, a6[i]);         pick random
        point_rand(&R, E);            point on
        elptic_mul(&P0, R, crde, E);  curve
        if(test_point(P0))            does multiply by cardinality
        {                             give point at infinity?
          printf("curve %d is right curve!\n", i);
          done = 1;  ← if yes then no need for twist calculation
        }
        else
          printf("curve %d is not right curve.\n", i);
      }
```

If any of the roots give a correct result from a test for the point at infinity, then the program is finished except for cleaning up the stack. If the done flag is not set after checking every possible curve then the coefficients for a twist curve is computed using the first root. This is shown in listing 14.19.

A random value is chosen which is a quadratic non-residue. This might take a few tries. Then new coefficients are computed using equation 14.4. The check for the point at infinity after multiplication by the curve cardinality is then checked again. Since all the roots are the same, if this fails something is wrong and there is no point in trying anything else.

Listing 14.19 Find curve compute twist coefficients

```
if(!done)                         if no curve is right
{                                 compute twist of
  k = 1;                          first one and try again.
  while(k >= 0)
  {
    mrand(c);                     find random
    k = msqr(c);                  quadratic
  }                               non-residue
  mmul(b, c, c);                  twist a4
  mmul(E.a4, a4[0], b);           is c^2 a4
```

```
    mmul(b, b, c);
    mmul(E.a6, a6[0], b);
    point_rand(&R, E);
    elptic_mul(&P0, R, crde, E);
    if(test_point(P0))
    {
      gmp_printf("a4 = %Zd  a6 = %Zd\n", E.a4, E.a6);
      printf("twist is right curve!\n");
    }
    else
      printf("twist is not right curve.\n");
  }
  mpz_clears(c, tp, b, j0, two, tri, NULL);
  mpz_clear(p);
  poly_clear(&x1);
  poly_clear(&Aofx);
  poly_clear(&Bofx);
  poly_clear(&hofx);
  for(i=0; i<=hc.deg; i++)
    mpz_clears(root[i], a4[i], a6[i], NULL);
  point_clear(&R);
  curve_clear(&E);
}
```

twist a_6 is $c^3 a_6$

does multiply by cardinality give point at infinity?

output result

clean up stack

After compiling and linking `get_curve` we can use the output from the last entry of listing 14.8 as inputs. The result is shown in listing 14.20.

Listing 14.20 Output get_curve

```
$ ./get_curve 55 188201910506566789076013521962076573175506406392106419771149765
094125925589359586529 -455454
0:  103819152823030692423078526202194651272381844248740088593429442093780725173O
662541635
1:  130228568175007853693563317488266515582973748803781228626776426192682850308O
11676860
2:  157648892626047296297274586026685872331546462459674549690428716926010515649 9
548298462
3:  101912918746554804062317599946445971188787131195420078395763728549192325342 7
359971476
0: a4= 323184758278707016800780289642567687579996821781745099591538691685393193O
67046598812  a6= 147013590889624993837394367284222227955670725713520619820202422
841776829937093745689 4
1: a4= 253209590405765595337802023168999975941443251105648298577934899990599822 9
98245456275  a6= 142348579698095565739862482852651047179767154335114166419295503
395457271932507002853 6
2: a4= 800567439490059568182015238102515494099278422299131397047615534879362470 6
29762792943  a6= 178839102970381830596143363848218748390289499081346372983940879
054708115107941491964 8
3: a4= 163082665437641004863099656557007075043229811967327368023985016587349315 9
40968015165  a6= 136340118033553926374882325078518187119886258392559437715698844
```

```
501907238128688506779 6
#E = 188201910506566789076013521962076573175506406392106419771149765094125925598
9360041984
curve 0 is right curve!
curve 1 is right curve!
curve 2 is not right curve.
curve 3 is not right curve.
```

From appendix B we see that the Hilbert class polynomial for discriminant 55 is 4^{th} order. The four roots are output first followed by the curve coefficients for each root. The cardinality is printed next followed by the random point check. In this case, the first two curves are the curve we want.

It does not matter which coefficient set we chose here. While the points on the two curves will be different, the total number of points on each curve is identical. By definition, they are isomorphic.

At this point we have found a base curve which has an extension to a degree 13 polynomial. Using the find irreducible polynomial routine `poly_irreducible()` and using that routines output in `poly_mulprep()` we are ready to work on a field extension curve. All the routines from chapter 13 allow us to work on the field extension curve.

14.5 Summary

- The ratio of the number of bits in the field prime to the number of bits in the largest prime factor determines the efficiency of a field extension. The algorithms chosen get this ratio as close to 1 as possible.
- Low embedding degree algorithms for all possible degrees are available. To maximize security only prime embedding degrees are described here.
- Curves with the same j-invariant are isomorphic. The correct curve (or its twist) will always be found by using the j-invariant to compute the curve coefficients.
- The method of complex multiplication (CM) uses the j-invariant to find the curve equation. We still have to test for the twist curve, but it is straightforward to change the coefficients to get the curve we want.
- The discriminant used to find a field prime and large prime order defines the Hilbert class polynomial used to find the j-invariant. While this is an active area of research to obtain exceptionally large discriminants, smaller values have not been found to be insecure.
- Factoring a k degree Hilbert class polynomial is a recursive process that finds k roots. Each root gives one j-invariant.
- The algorithms for low embedding degree have a low probability of success. Many values must be tested to find acceptable parameters.
- Direct calculation with the j-invariant can find the twist of the desired curve. It is necessary to verify the point at infinity is reached when a random point is multiplied by the curve cardinality. This proves we have the correct curve.

Chapter Bibliography

Barreto, Paulo S. L. M., Lynn, Ben, & Scott, Michael. 2003. Constructing Elliptic Curves with Prescribed Embedding Degrees. *Pages 257–267 of:* Cimato, Stelvio, Persiano, Giuseppe, & Galdi, Clemente (eds), *Security in Communication Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg. 164

Blake, I., Seroussi, G., & Smart, N. 1999. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press. 165, 167, 168

Cohen, Henri. 2000. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag. 165, 168, 169

Freeman, David, Scott, Michael, & Teske, Edlyn. 2006. *A taxonomy of pairing-friendly elliptic curves*. Cryptology ePrint Archive, Paper 2006/372. https://eprint.iacr.org/2006/372. 164, 165, 166, 167

Silverman, J.H. 2013. *Advanced Topics in the Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York. 165

14.6 Answers to exercises

- 14.1) $\#E$ is the cardinality of an elliptic curve, q is field prime the curve is defined over, t is the trace of Frobenius, h is the small cofactor of the cardinality, and r is the large prime factor which determines the security of the elliptic curve.

15 General rules of elliptic curve pairing explained

This chapter covers

- An introduction to elliptic curve point pairing.
- A geometric description of the pairing function.
- The essential rules of pairing mathematics.
- Routines to compute common pairing functions.

This chapter presents the point pairing mathematics over field extension elliptic curves.

With the background mathematics under our belts we can now begin to look at the mathematics of pairing points on an elliptic curve. Pairing points gives us a one way trap door function which is efficient and secure. It also gives us efficient ways to compute algorithms which would be horribly complicated otherwise. Point pairing is a very deep subject and I will do my best to ignore most of it. The fundamental reason for ignoring this depth is that the underlying mathematics works and fully understanding why it works is not critical to writing functional code.

In this chapter, I explain the general mathematical rules of pairings at a high level. The rules are mostly symbolic manipulation, so the meanings will be explained before we get into details.

There are two fundamental subroutines used in the following chapters which are the guts of computing elliptic curve pairings. In this chapter I give a geometric hand waving

argument that completely ignores the actual mathematical underpinnings of these routines. While the theory is extremely interesting, it is not necessary to understand everything to use the algorithms. To gain a deeper understanding of the theory, I recommend reading sections III.8 and XI.9 in (?).

15.1 Mathematical rules of elliptic curve pairings

In this section I explain an elliptic curve group structure required for pairings to work mathematically. The result of a pairing operation is not a point. I describe what the result actually is and use vectors on the complex plane to get the idea across.

The group structure of an elliptic curve over a finite field is either cyclic or the product of two cyclic groups. In fact, we can always say it is the product of two cyclic groups if one of the groups has the size of one element. It is just more convenient to think of a base curve having only one cyclic group.

From section III.3 in (?) we find that the structure of an elliptic curve over a finite field is written as

$$E_{F_q} \cong \frac{Z}{d_1 Z} \times \frac{Z}{d_2 Z} \tag{15.1}$$

where d_1 divides d_2. The meaning of the symbol $\frac{Z}{d_j Z}$ is "all integers divided by d_j times all integers". This boils down to integers modulo d_j. For cryptographic purposes we look for groups where d_1 is a large prime. Since d_1 is a factor of d_2 the order of the curve is at least d_1^2 and there are no points of order d_1^2 because the two groups are separate.

For base curves the field q is a prime number and for almost all cases $d_1 = 1$. That means there is just one cyclic group. In chapter 2 when I described the group sizes this is what I was referring to. In chapter 3 I showed the doughnut with two vectors which is repeated here in figure ??. The two vectors represent the two independent groups d_1 and d_2. We now look for curves over a field extension with two separate groups so we can create elliptic curve pairings.

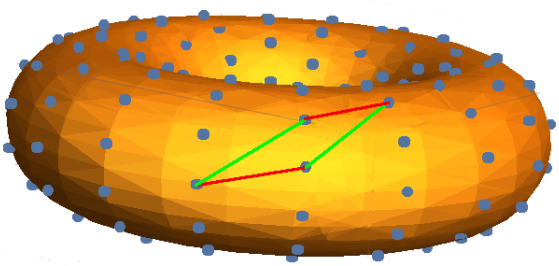

Figure 15.1 Elliptic curve over the complex plane

Figure ?? shows a circle to represent a cyclic group. For pairings to work a double cyclic structure is required. Pairings will not work if there is only one of cyclic group. The phrase "single cycle" means $d_1 = 1$, the group structure is not complex enough to allow pairings to work.

There must be a large prime that is squared in the order of the curve, and each factor must be part of the two groups described by formula ??. I am pretty sure this is what makes pairings so difficult to comprehend. It is not just that the rules are complicated. Finding curves with the correct group structure is challenging on top of the complexity.

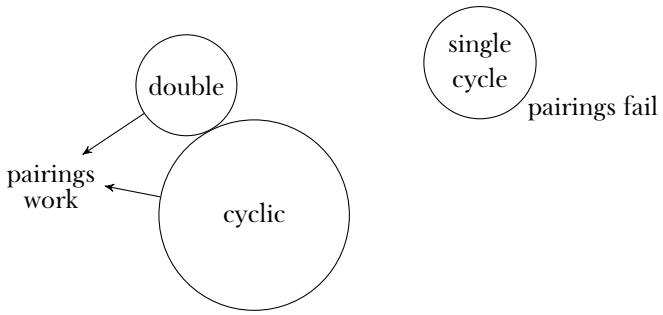

Figure 15.2 Elliptic curve group structure for pairings to work

Given two points P and Q on an elliptic curve of order m the pairing of those points is written as

$$e_m(P, Q) = \mu_m.$$

The form $e_m(\cdot, \cdot)$ is the pairing function and the symbol μ_m is an m^{th} root of unity. As an example from complex numbers, an m^{th} root of unity is

$$\mu_m = e^{2i\pi n/m}$$

where m and n are integers and $i = \sqrt{-1}$. Taking μ_m to the m^{th} power gives

$$\mu_m^m = e^{2i\pi n} = 1.$$

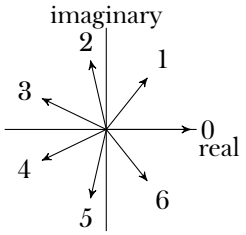

Figure 15.3 n^{th} root of unity for $n = 7$ using complex numbers. Multiply any of these vectors by 7 and the result is 1.

Figure ?? shows what μ_7 looks like on the complex plane. Each number in figure ?? is the value of n in $e^{2i\pi n/m}$. All the angles are $2\pi/7 \cdot n$, which clearly gets us back to 2π when $n = 7$. The same idea applies with pairings of points over elliptic curves.

For pairings, μ_m is a member of the field extension p^k that defines the points P and Q. That means there are elements of order m in the extension field and points of order m on the elliptic curve.

Note that the way we compute the order of a point is mP but the way we compute the order of a field element is μ_m^m (that is, μ_m to the m^{th} power). In the order of a point we get the identity element which is the point at infinity and in the order of an m^{th} root of unity we get the identity element 1. The process of pairing points takes us from multiplication of points (which is addition) to powers of elements (which is multiplication).

This is an important form of magic, so let's look at some of the rules. The rules I cover here include bilinearity and non-degeneracy. Different pairings will have additional capabilities. These two rules are fundamental to all pairing operations.

Exercise 15.1

Show that $32x + 34 \bmod 43$ is an 11^{th} root of 1 modulo $p = x^2 + x + 3 \bmod 43$. Hint: use PARI/gp.

15.1.1 Elliptic curve point pairing rule of bilinearity

In this section the mathematical rule of bilinearity is described. It is fundamental to how pairings work.

The primary rule of importance for pairings is bilinearity which means it has the same results on the right and left sides of the pairing. Take three points of order m as R, S and T. Then the rule is

$$\begin{aligned} e_m(R+S,T) &= e_m(R,T)e_m(S,T) \\ e_m(R,S+T) &= e_m(R,S)e_m(R,T) \end{aligned} \tag{15.2}$$

that is, the pairing of the sum of two points is the multiplication of the pairings of those points. The rule converts addition of points to multiplication of pairings.

To expand on this rule we can take a multiplication of a point with some integer n and pair it with another point. The result will be the power of the pairing of the two points. That is

$$e_m(nS,T) = e_m(S,T)^n.$$

Similarly by linearity we have

$$e_m(S,nT) = e_m(S,T)^n.$$

If we take integers a and b multiplied with points S and T then the same rule as ?? becomes

$$e_m(aS,bT) = e_m(S,T)^{ab}. \tag{15.3}$$

Another important and useful relationship is

$$e_m(-S,T) = e_m(S,-T) = e_m(S,T)^{-1} \tag{15.4}$$

Rule ?? is consistent with rules ?? and ??. It will allow us to create some interesting protocols.

This ability to combine points from an additive domain into a multiplicative domain is what makes pairings such a powerful tool for cryptographic algorithms. The result is no longer a point on the curve, it is a field element in p^k. Given the number of possible combinations of points that lead to a field element the use of pairings as a one way trap door function has a lot of appeal.

Exercise 15.2

Show that $e_m(aR + bS, cT) = [e_m(R, T)^a e_m(S, T)^b]^c$.

15.1.2 Non-degeneracy rule with point at infinity

This section describes how non-degeneracy works. This rule prevents division by zero.

The point at infinity is not actually on an elliptic curve. It is required to be the identity element on an elliptic curve for the arithmetic to work. Point pairing operations may hit the point at infinity as an input so a special rule of non-degeneracy is included to ensure pairings work under all possible conditions.

If one of the points of a pairing is the point at infinity the rule of non-degeneracy states we get 1 for an answer. The reason for this rule is to prevent division by zero. We can write this rule as

$$e_m(S, \mathbf{0}) = e_m(\mathbf{0}, S) = 1. \tag{15.5}$$

This is true for any point S of order m on the curve. For example, if we take point Q as the point at infinity and attempt to compute a pairing with T and $-T$ one of those results is the same as the inverse of the other. That is

$$e_m(\mathbf{0}, T) = e_m(\mathbf{0}, -T) = e_m(\mathbf{0}, T)^{-1} = 1.$$

The main advantage of this rule is that it forces our code to behave nicely. The algorithms presented in the next section take this rule into account as a special case.

15.2 Algorithms for pairing

In this section I introduce two algorithms that are used to compute pairings. For the first algorithm, I will use a geometric image to give an idea of how it works. The real mathematics is much deeper, but you don't need to know it all to use the algorithm correctly. The second is called Miller's algorithm which is named after the mathematician who developed it.

The first algorithm is actually a function used in Miller's algorithm. Miller's algorithm is a function used in the pairings to be presented in chapters 16 and 17. The details of both algorithms can be found in section XI.8 in (?). In the mathematical literature both of these functions have different symbols, so I will follow (?) and call the first function $h_{P,Q}(R)$ and the second function $f_P(R)$.

15.2.1 Function $h_{P,Q}(R)$

This section explains the core function of elliptic curve pairing operations.

The first algorithm is the heart of point pairing. It does not really have a name, so the symbol $h_{P,Q}(R)$ is the best reference. In this section I show how two points are paired using a third point as a reference. In the symbol, P and Q are the points being paired while R is the reference.

Figure ?? is an attempt to depict the relationships between three points P, Q and $R = (x, y)$. This is on the same curve shown in chapter 3 with

$$y^2 = x^3 - 5x + 5.$$

The sum of points P and Q results in the point $P+Q$. The line between P and Q intersects the curve at x coordinate x_{P+Q} shown with a dashed vertical line in figure ??.

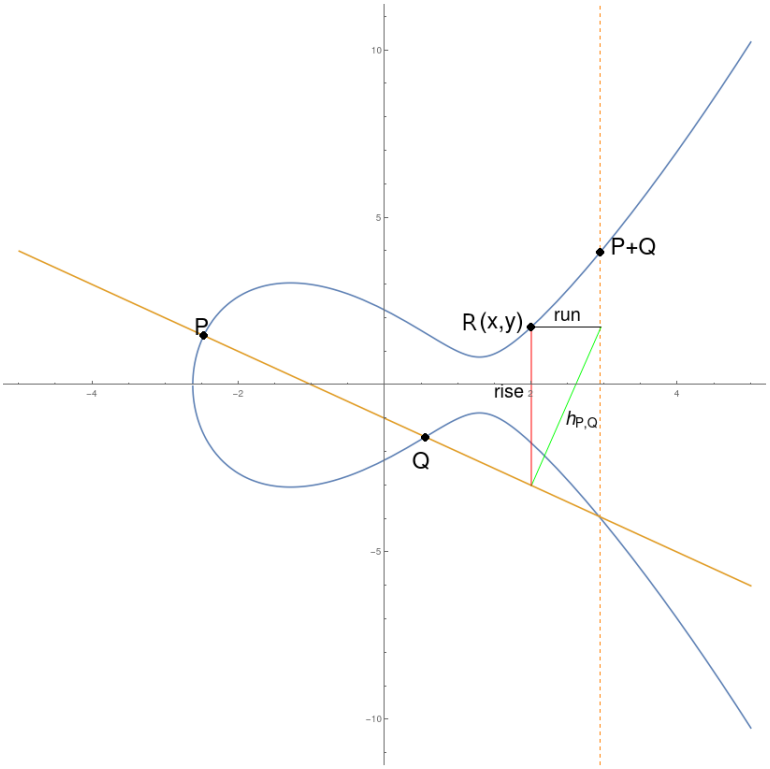

Figure 15.4 $h_{P,Q}(R)$ **function diagram: line between points** P **and** Q **defines rise and run to reference point** R

Taking an arbitrary point $R = (x, y)$ on the curve that is not related to P or Q, and not the point at infinity, we can define the distance from that point to the x coordinate of $P+Q$ as the "run". This is labeled in figure ??.

The vertical distance from the line intersecting P and Q to the point R is called the "rise" which is also labeled in figure ??. The function $h_{P,Q}$ is defined as the rise over the

run:

$$h_{P,Q}(R) = \frac{\text{rise}}{\text{run}}.$$

The horizontal distance defining the run is easy to see as $x - x_{P+Q}$. The vertical distance is easy to derive. We know that the slope of the line between P and Q is λ which we found in chapter 3. Let's take the line intersecting P and Q as $y' = \lambda x' + \nu$. The distance labeled rise in figure ?? is then $y - \lambda x - \nu$.

Combining the rise $= y - \lambda x - \nu$ with the run $= x - x_{P+Q}$ gives us the formula

$$h_{P,Q} = \frac{y - \lambda x - \nu}{x - x_{P+Q}}.$$

From chapter 3 equation 3.3 we have

$$x_{P+Q} = \lambda^2 - x_P - x_Q.$$

Since the point P is on the line we can take

$$\nu = y_P - \lambda x_P.$$

Plugging these two formulas into the formula for $h_{P,Q}$ gives

$$h_{P,Q}(R) = \frac{y - y_P - \lambda(x - x_P)}{x + x_P + x_Q - \lambda^2} \tag{15.6}$$

In the special case when $P = -Q$ the slope of the line between them is infinity. The function $h_{P,Q}(R)$ is then defined to be

$$h_{P,Q}(R) = x - x_P \qquad \lambda = \infty \tag{15.7}$$

As seen in figure ?? this is just the "run" between the vertical line and the point R.

The final special case is when either P or Q is the point at infinity. Then we have

$$h_{P,Q}(R) = 1 \qquad P = \mathbf{0} \text{ or } Q = \mathbf{0} \tag{15.8}$$

The software subroutines will first check if either input is the point at infinity, then check if the slope is infinite and then compute equation ?? if those tests are not applicable.

Exercise 15.3

Is there any problem with the $h_{P,Q}(R)$ when P and Q are the same point?

15.2.2 Miller's algorithm

In this section I cover Miller's algorithm as a sequence of steps using a flow chart. The inputs to Miller's algorithm are two points on the curve, with one of them being the reference point mentioned in the $h_{P,Q}(R)$ algorithm.

Miller's algorithm looks like the multiplication of a point by m, the order of the point. The double and add algorithm is followed for every bit in m as shown in figure ??. The

output of the algorithm is written as $f_P(R)$ with the subscript point being one of the points in the pairing and the point R being a reference point. In chapters 16 and 17 we see how Miller's algorithm is used.

The function $h_{T,T}(R)$ means the slope λ is a tangent, and we use equation 3.2 to compute the slope. Repeated here the equation is

$$\lambda = \frac{x_1^2 + x_1 x_2 + x_2^2 + a_4}{y_1 + y_2} \tag{15.9}$$

with $x_1 = x_2 = x_T$ and $y_1 = y_2 = y_T$. For the addition step we use equation ?? with $x_1 = x_T$ and $x_2 = x_P$. Similarly, $y_1 = y_P$ and $y_2 = y_P$.

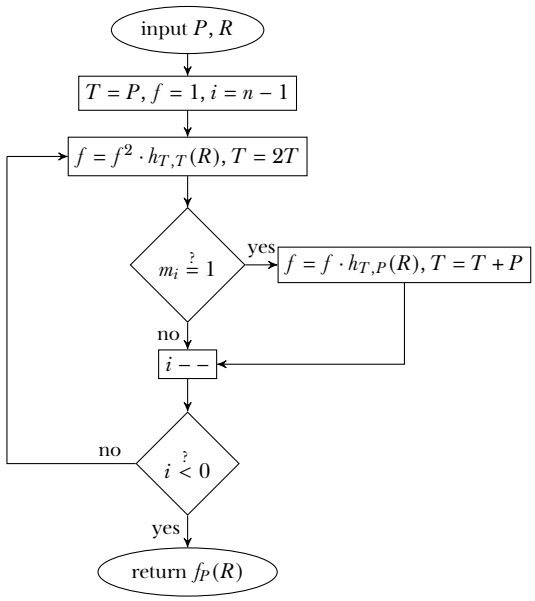

Figure 15.5 Miller's algorithm flow chart

Miller's algorithm takes two points as input and returns a field element. All the equations are computed on a field extension of a curve. The value of every variable is a polynomial modulo the prime polynomial that defines the field extension. So the equations look simple, but all the subroutines of the previous chapters are required to make them work.

Exercise 15.4

Is every variable in figure ?? a polynomial modulo a prime polynomial (or a point made from a pair of polynomials modulo a prime polynomial)?

15.3 Subroutine $h_{P,Q}$

In this section I describe code which implements the $h_{P,Q}(R)$ function. I first deal with the exceptional inputs which might cause problems, then describe the main algorithm which

does the calculations.

The pairing subroutines are in file `pairing.c`. In this chapter, I describe the primary routines used for computing pairings. In chapters 16 and 17, I explain both the pairing computation itself along with some of the utility functions.

This section covers the routine which computes the formulas ?? through ??. The routine `hpq()` is shown in listing ??. The inputs are three points and the elliptic curve the points are on. The output is a single polynomial value.

The first check on entry is for either input being the point at infinity. If either point does test positive as the point at infinity, the output of `hpq()` is set to 1. The function returns without creating any internal variables.

Listing 15.1 Computing $h_{P,Q}$ infinite slope check

```
void hpq(POLY *h, POLY_POINT P, POLY_POINT Q, POLY_POINT R, POLY_CURVE E)
{
  POLY t, lmbda, b, tx, t1;

  if((poly_test_point(P)) || poly_test_point(Q))   ← is either point 0?
  {
    h->deg = 0;
    mpz_set_ui(h->coef[0], 1);          | yes, return value of 1
    return;
  }
  poly_init(&t);
  poly_init(&b);
  poly_add(&b, P.y, Q.y);
  if(!b.deg && !mpz_cmp_ui(b.coef[0], 0))   ← is y_P + y_Q = 0?
  {
    poly_sub(&b, P.x, Q.x);          ← is x_P − x_Q = 0?
    if(!b.deg && !mpz_cmp_ui(b.coef[0], 0))   ← Really P == -Q?
    {
      poly_sub(h, R.x, P.x);
      poly_clear(&t);                | slope is infinite, return x_R − x_Q
      poly_clear(&b);
      return;                        | y sum was zero
    }                                | x difference is not
    poly_sub(&t, P.y, Q.y);          | so slope is computable
  }
```

The variables `t` and `b` are for top and bottom of the slope λ calculation. If the two y inputs sum to zero then a check that the x coordinates are not the same is used to determine if the slope is infinite. If the two x coordinates are the same then we return the value specified in equation ??. If the two x coordinates are different then we can compute the slope as the difference in y coordinates divided by the difference in x coordinates.

The conditions which make the first if statement true are rare, so most cases enter the else section as shown in listing ??. This computes the `t` and `b` variables directly from numerator and denominator of equation ??. On exit from the else section both `t` and `b` are

set and λ is found by their division.

Listing 15.2 Computing $h_{P,Q}$ slope

```
else                          ← not a special case
{                               compute lambda (slope between P and Q)
                                using secure form
  poly_init(&t1);
  poly_init(&tx);
  poly_mul(&t, P.x, P.x);
  poly_mul(&t1, P.x, Q.x);
  poly_mul(&tx, Q.x, Q.x);
  poly_add(&t, t, t1);
  poly_add(&t, t, tx);
  poly_add(&t, t, E.a4);
  poly_add(&b, Q.y, P.y);     ← y_Q + y_P
}
poly_init(&lmbda);
poly_div(&lmbda, t, b);       ← slope is top/bottom
```

$x_P^2 +$
$x_P x_Q +$
$x_Q^2 +$
a_4

Listing ?? shows the calculation of equation ??. The variables `t` and `b` are reused for the numerator and denominator. Once the value of $h_{P,Q}$ is placed in the designated output location the stack is cleaned up, and the routine is finished.

Listing 15.3 Computing $h_{P,Q}$

```
  poly_sub(&t, R.y, P.y);
  poly_sub(&tx, R.x, P.x);
  poly_mul(&tx, tx, lmbda);
  poly_sub(&t, t, tx);
  poly_mul(&tx, lmbda, lmbda);
  poly_sub(&b, R.x, tx);
  poly_add(&b, b, P.x);
  poly_add(&b, b, Q.x);
  poly_div(h, t, b);          finally, compute h
  poly_clear(&t);
  poly_clear(&b);
  poly_clear(&tx);            clean
  poly_clear(&t1);            up
  poly_clear(&lmbda);         stack
}
```

numerator =
$y_R - y_P - \lambda(x_R - x_P)$

denominator =
$x_R + x_P + x_Q - \lambda^2$

15.4 Miller's algorithm code

This section presents the code which computes point pairings at the lowest level. This subroutine will be called by routines in chapters 16 and 17.

The code to compute the algorithm shown in figure ?? is very similar to the routines described for point multiplication in chapters 3 and 13. The fundamental difference is that now there are three points involved as well as a field element to keep track of.

Listing 15.4 Miller's algorithm routine

```
void miller(POLY *f, POLY_POINT P, POLY_POINT R, mpz_t m, POLY_CURVE E)
{
  POLY_POINT T;
  POLY h;
  long mask;

  mask = mpz_sizeinbase(m, 2) - 2;  ← 1 less than number of bits
  poly_point_init(&T);
  poly_point_copy(&T, P);
  f->deg = 0;                       | initialize f = 1
  mpz_set_ui(f->coef[0], 1);        | and T = P
  poly_init(&h);
  while(mask>=0)   ← for every bit in m
  {
    hpq(&h, T, T, R, E);
    poly_mul(f, *f, *f);            | f ← f^2 · h_{T,T}
    poly_mul(f, *f, h);
    poly_elptic_sum(&T, T, T, E); ← T ← 2T
    if(mpz_tstbit(m, mask))   ← is this bit set?
    {
      hpq(&h, T, P, R, E);
      poly_mul(f, *f, h);           | f ← f · h_{T,P}
      poly_elptic_sum(&T, T, P, E); ← T ← T + P
    }
    mask--;   ← go to next bit
  }
  poly_point_clear(&T);
  poly_clear(&h);                   | clean up stack
}
```

The point T keeps track of the multiplication of input point P by the order of the point (which is m). The $h_{P,Q}$ function uses T and R in the doubling step and all three points T, P and R in the addition step.

At both the doubling step which happens for every bit and the addition step which happens for every set bit, the value of f is modified by the appropriate formula as shown in figure ??.

Since there is no chance of over writing an input point the output $f_P(R)$ is computed at each step directly. The variable `f` is assumed initialized before the call and what ever it was before is lost.

The `miller` routine is fundamental to pairing calculations. In the next two chapters we will use it to compute related but quite different pairings.

15.5 Summary

- To incorporate elliptic curve pairing algorithms, curves which allow pairings must have a dual group structure of the form

$$E_{F_q} \cong \frac{Z}{d_1 Z} \times \frac{Z}{d_2 Z}.$$

  Purely cyclic curves described in chapter 3 will not work.

- The pairing of points on an elliptic curve results in a field element which is a root of unity. If the order of the points is m then the polynomial result of pairing is an m^{th} root of unity. This gives us a one way trap door function for cryptographic use.

- The pairing function $e_m(R, T)$ is bilinear. The general form to show this is

$$e_m(aS, bT) = e_m(S, T)^{ab}.$$

  This gives us a way to convert point addition over an elliptic curve to multiplication over a finite field which is used in algorithms shown in chapters 18 and 19.

- The pairing function is non-degenerate which means it never blows up. If one of the input points is the point at infinity the value of the function is 1. This is essential to ensure algorithms work under any condition.

- The function $h_{P,Q}(R)$ can be viewed geometrically as a relation between the line adding P plus Q and a reference point R. $h_{P,Q}(R)$ is the core of Miller's algorithm.

- Miller's algorithm computes the function $f_P(R)$. The input is a point of order m and a reference point. The output is a field element based on multiplying the input point by m and using the $h_{P,Q}$ function for doubling and adding. Miller's algorithm is the function used in chapters 16 and 17 to compute different kinds of pairings.

Chapter Bibliography

Blake, I., Seroussi, G., & Smart, N. 1999. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press.

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York.

15.6 Answers to exercises

- 15.1) Create the irreducible polynomial p using the Mod() command:

```
? p = Mod(1,43)*x^2+Mod(1,43)*x+Mod(3,43)
%59 = Mod(1, 43)*x^2 + Mod(1, 43)*x + Mod(3, 43)
```

  Then make $32x + 34$ a polynomial modulo p:

```
? a = Mod(Mod(32,43)*x+Mod(34,43),p)
= Mod(Mod(32, 43)*x + Mod(34, 43), Mod(1, 43)*x^2 + Mod(1, 43)*x + Mod(3, 43))
```

  Finally take a to the 11^{th} power:

```
? a^11
%61 = Mod(Mod(1, 43), Mod(1, 43)*x^2 + Mod(1, 43)*x + Mod(3, 43))
```

  The result is 1, which shows that $32x + 34$ modulo $x^2 + x + 3$ is an 11^{th} root of unity.

- 15.2) From the point addition rule we have $e_m(aR+bS, cT) = e_m(aR, cT)e_m(bS, cT)$ From the rule of equation ?? this becomes $e_m(R, T)^{ac}e_m(S, T)^{bc}$. Factoring out c from the exponent gives the result $[e_m(R, T)^a e_m(S, T)^b]^c$.
- 15.3) No. This is similar to point doubling where the slope is tangent to the elliptic curve. Since λ is finite, equation ?? still applies.
- 15.4) No, i and n are integers and m_i is a bit. T and P are points which are pairs of polynomials and f (which is equal to $f_P(R)$) is a polynomial modulo a prime polynomial.

16 Weil pairing defined

This chapter covers

- Weil pairing properties
- Code to compute Weil pairing
- Example with tiny numbers

In this chapter I describe Weil pairings. These will be used with the example application shown in chapter 18.

In chapter 15 I covered the fundamental functions that compute all pairings. In this chapter I will explain the Weil pairing which has properties that make it unique. For some cryptographic protocols these properties make the Weil pairing more useful, and for other protocols the Weil pairing does not work. All the protocols explained in chapters 18 and 19 use different base points for the pairing algorithms, so the Weil pairing could work in principle. If you run across algorithms which have the same base point for both pairing input points, the Weil pairing will not work, use the Tate pairing from chapter 17 instead.

The formula that computes the Weil pairing uses the Miller $f_P(R)$ function from chapter 15. After showing the properties of the Weil pairing I describe formulas in gory detail so you can get a feel for why certain calculations should give specific results. This arms us for debugging our code because the math has to work.

There are several utility routines included in this chapter along with the Weil subroutine. After describing these routines I use the tiny example from chapter 13 to show how specific inputs from G_1 and G_2 points give useful and failing results. The purpose of the example

is to show how the Weil pairing works, not to explain any particular protocol.

As a reminder, G_1 are points on a base curve and G_2 are points on a field extension curve. The subroutines are written with the assumption that all points are on the field extension G_2.

16.1 Weil pairing formula

In this section I describe the Weil pairing formula.

In 1940 A. Weil introduced the concept of pairing points of order n as a general concept to solve a problem unrelated to elliptic curves. The method was then applied to elliptic curves as a way to uncover the secret key. Today we have algorithms which take advantage of the rules Weil developed. In this section I go over the formulas which compute the Weil pairing and describe the rules which result from those computations.

There are several definitions of the Weil pairing. I am going to use the version in section XI.8 of (?). The definition uses three points. Two points are being paired, and the third point is a reference point. The reference point must not be in the same subgroup as the points being paired. Figure ?? is a duplicate of figure ??.

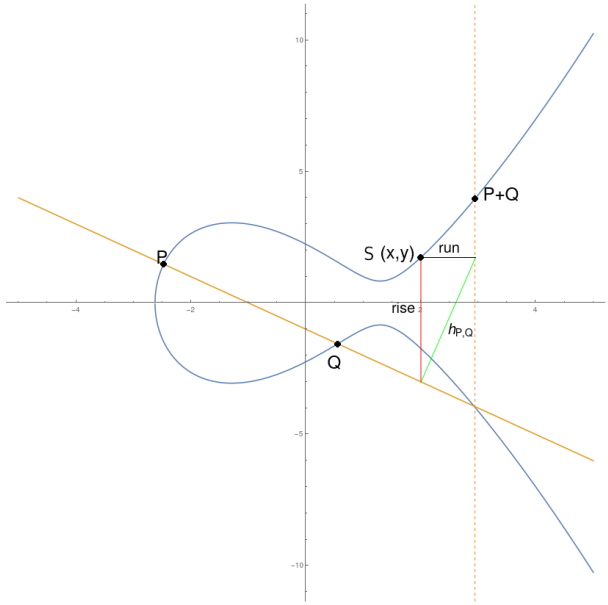

Figure 16.1 Relations between points P, Q being paired and reference point S

We take two points of prime order m, call them P and Q. The third point is S which is not in the same subgroup of P and Q. That is, it has order different from m. As pointed out in chapters 2 and 13, elliptic curves over finite fields have groups of points with the number of points in each group related to the factors making up the cardinality of the curve. For the point S, it is OK for m to be one of many factors in the order of S. Typically, most random points on a field extension will have that property simply because the size of the

group m is small compared to the extension field curve cardinality.

Using the function $f_P(R)$ from section ??, the Weil pairing is defined as

$$e_m(P,Q) = \frac{\frac{f_p(Q+S)}{f_p(S)}}{\frac{f_Q(P-S)}{f_Q(-S)}} \tag{16.1}$$

Equation ?? is written as a fraction of two fractions to show the relationship with the reference point S and the two input points. The first argument of the pairing is the input parameter to f on the top fraction and the second argument of the pairing is the input parameter to f on the bottom fraction. This symmetry leads to a new rule for the Weil pairing that is not available with other pairing definitions.

The Weil pairing has a property called alternating. If both input points are the same we have

$$e_m(T,T) = 1. \tag{16.2}$$

An additional part of this alternating property is that swapping the arguments inverts the result. That is

$$e_m(P,Q) = e_m(Q,P)^{-1}. \tag{16.3}$$

Equation ?? seems obvious from the definition ??. But the equation ?? implies that

$$\frac{f_T(-S)}{f_T(S)} = \frac{f_T(T+S)}{f_T(T-S)}.$$

While these values are equal, they don't have to be equal to 1. A hand waving argument why this should be true (other than the mathematicians have proved it) is that we only care about the distance from the reference point to the vertical line attached to the sum of the two points in figure ??. The choice of positive or negative point for the reference point does not matter. Clearly it actually does matter or the definition would be different. The negative points are in the denominator of equation ??. The math works, so we work with the formulas as defined.

Let's rewrite ?? in a more computational friendly manner

$$e_m(P,Q) = \frac{f_Q(-S)}{f_P(S)} \frac{f_P(Q+S)}{f_Q(P-S)} \tag{16.4}$$

and examine the rule

$$e_m(P,-Q) = e_m(P,Q)^{-1} = e_m(Q,P).$$

The expansion of this using ?? is

$$\frac{f_{-Q}(-S)}{f_P(S)} \frac{f_P(S-Q)}{f_{-Q}(P-S)} = \frac{f_P(S)}{f_Q(-S)} \frac{f_Q(P-S)}{f_P(Q+S)} = \frac{f_P(-S)}{f_Q(S)} \frac{f_Q(P+S)}{f_P(Q-S)} \tag{16.5}$$

A quick glance at this formula is rather head scratching. In the first term we have $f_{-Q}(-S)$ and $f_{-Q}(P-S)$ which do not appear in either of the other two terms. In the

second term we have $f_P(Q+S)$ in the denominator with $f_Q(P+S)$ in the numerator of the third term. According to the rules of pairings all these formulas are in fact identical. We can use formulas like ?? to ensure our code is correct. Computing the three different Weil pairings around an inverse should all give the same answer. If they don't all give the same result, we know we have a bug to find.

> **Exercise 16.1**
>
> The Diffie-Hellman protocol uses two public keys created from different private keys and the same base point. Why would the Weil pairing of those two public keys be useless? Hint: review pairing rule ??.

16.2 Pairing subroutines

This section provides the code to implement the Weil pairing of points on field extension elliptic curves.

The Weil pairing subroutine is simple with just four calls to the Miller algorithm from section ??. In addition to showing that listing, I also add the routines mentioned in chapter 13. Utilities include computing the cardinality recurrence formula 13.3 and finding the order of a point for both base and extension fields.

The Weil pairing calculation is shown in listing ??. There are three points derived from the input points shown in equation ??. These are the points $Q+S$, $-S$ and $P-S$. The first is computed directly from the input points. The value for $-S$ is created by first copying over the x component into the initialized point `mS` and then subtracting the y component of S from the zeroed out y component of `mS`. Adding this value to the point P gives the final point we need.

After computing the four Miller functions the top two are divided, then the bottom two. The final answer is the division of these two intermediate results. The last operation is cleaning up the stack.

Listing 16.1 Weil pairing routine

```
void weil(POLY *w, POLY_POINT P, POLY_POINT Q, POLY_POINT S, mpz_t m,
          POLY_CURVE E)
{
  POLY_POINT QpS, mS, PmS;
  POLY t1, t2, t3, t4, w1, w2;

  poly_point_init(&QpS);
  poly_elptic_sum(&QpS, Q, S, E);      | create Q + S point
  poly_point_init(&mS);
  poly_copy(&mS.x, S.x);               |
  poly_point_init(&PmS);               | create -S point
  poly_sub(&mS.y, PmS.y, S.y);         |
  poly_elptic_sum(&PmS, P, mS, E);     <- create P - S point
  poly_init(&t1);
  miller(&t1, P, QpS, m, E);           <- compute f_P(Q + S)
```

```
  poly_init(&t2);
  miller(&t2, P, S, m, E);      ← compute f_P(S)
  poly_init(&t3);
  miller(&t3, Q, PmS, m, E);    ← compute f_Q(P − S)
  poly_init(&t4);
  miller(&t4, Q, mS, m, E);     ← compute f_Q(−S)
  poly_init(&w1);
  poly_div(&w1, t1, t2);        | Weil pairing =
  poly_init(&w2);               | f_P(Q + S) / f_P(S)
  poly_div(&w2, t3, t4);        | over
  poly_div(w, w1, w2);          | f_Q(P − S) / f_Q(−S)

  poly_clear(&w1);
  poly_clear(&w2);
  poly_clear(&t1);              |
  poly_clear(&t2);              |
  poly_clear(&t3);              |
  poly_clear(&t4);              | clean up stack
  poly_point_clear(&QpS);       |
  poly_point_clear(&PmS);       |
  poly_point_clear(&mS);        |
}
```

The equation for cardinality of a curve over a field extension is given in chapter 13 along with the recurrence relation in equations 13.2 and 13.3. Copied here they are

$$\begin{aligned} \#E_k &= p^k + 1 - t_k \\ t_{n+2} &= t_1 t_{n+1} - p t_n . \end{aligned} \tag{16.6}$$

The routine to compute this is shown in listing ??. The inputs to this routine are the trace of Frobenius and the embedding degree. It is assumed that the field prime has already been set.

Because the input variable is `t` I used the array `v[]` to keep track of the t_n in equation ??. Since we start at zero and go to k there are $k + 1$ elements in the `v[]` array. After space is allocated the array elements are initialized to zero and then the first two are set to the correct values to start the recurrence.

Listing 16.2 Field extension cardinality

```
void cardinality(mpz_t crd, mpz_t t, long k)
{
  mpz_t *v, t1, t2, p, pk;
  int i;

  v = (mpz_t *)malloc(sizeof(mpz_t)*(k+1));   | allocate space
  for(i=0; i<=k; i++)                         | for k + 1
    mpz_init(v[i]);                           | variables
  mpz_set_ui(v[0], 2);                        | first two values are
  mpz_set(v[1], t);                           | 2 and t
  mpz_inits(t1, t2, NULL);
```

```
  mget(p);                                  ← recover field prime
  for(i=2; i<=k; i++)                       ← first two values already set
  {
    mpz_mul(t1, t, v[i - 1]);
    mpz_mul(t2, p, v[i - 2]);               recurrence is
    mpz_sub(v[i], t1, t2);                  t_{n+2} = t_1 t_{n+1} - p t_n
  }
  poly_q_get(pk);                           get p^k from
  mpz_set(crd, pk);                         global static storage
  mpz_add_ui(crd, crd, 1);                  compute
  mpz_sub(crd, crd, v[k]);                  #E_k = p^k + 1 - t_k
  for(i=0; i<=k; i++)
    mpz_clear(v[i]);
  mpz_clears(t1, t2, p, pk, NULL);          clean up stack
}
```

With the field prime collected from the `modulo.c` static global the recurrence relation is computed in the loop. The value of p^k is taken from global storage in file `poly.c` and equation ?? is computed for the cardinality.

16.3 Example code with tiny curves

In this section an example with printable points is shown which implements the Weil pairing.

In chapter 13 I introduced a tiny field prime and extension curve. I want to show what happens with several of the points from that example when used with the Weil pairing. First I describe the routines used in chapter 13 which checked the order of a point. Then I look at the remainder of the program introduced in chapter 13 to examine Weil pairings.

Two utility routines were used in the example code from chapter 13. These look for the order of a point. The assumption is that we have already found all the possible factors. These routines won't work on really large extension fields. In fact, it is the difficulty of factoring these very large numbers which helps with their security.

Figure ?? is a flow chart for the routine which finds the order of a point. The inputs include the point, curve, and list of factors which make up the order of the curve as well as the number of factors. If all the factors are correct no error should ever occur.

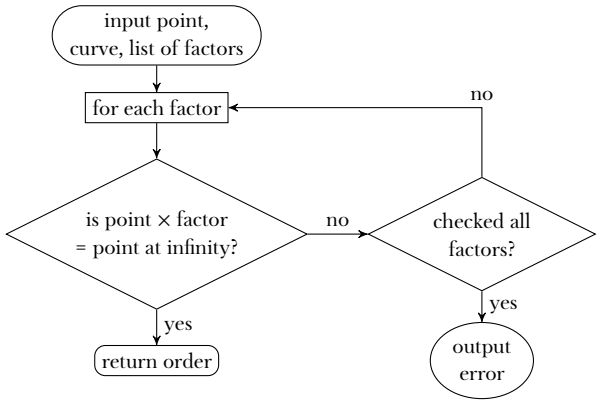

Figure 16.2 Subroutine to find order of a point with known factors

Listing ?? takes a point and a curve as input. It also expects an array of integers which are all the possible factors the point could have. The length of that array is the final input value. The routine brute force tests multiplication of each factor with the point until one of the multiplies hits the point at infinity. If none of the factors is correct the routine halts the program with an error that some factor is missing.

Listing 16.3 Order of point on base curve

```
int get_order(mpz_t order, POINT P, CURVE E, mpz_t *factors, int n)
{                                                    ↑ array of factors
  int i;                                             | and number of factors
  POINT R;                                           ↓

  point_init(&R);
  for(i=0; i<n; i++)
  {
    elptic_mul(&R, P, factors[i], E);   | when a factor
    if(test_point(R))                   | gives point at infinity
      break;                            | we have point order
  }
  if(i<n)
    mpz_set(order, factors[i]);     ← set output order
  else
  {
    printf("missing order in base!!\n");     | catastrophic error
    exit(-3);                                | kill program
  }
  point_clear(&R);
  return i;                  ← return index of factor as well
}
```

Listing ?? shows the same process for polynomial field extension points and curves. As we'll see in the example code, the factor list is the same array as in listing ?? but the list is

longer.

Listing 16.4 Order of point on field extension

```
int poly_get_order(mpz_t order, POLY_POINT P, POLY_CURVE E,
                   mpz_t *factors, int n)
{
  int i;                          array of factors
  POLY_POINT R;                   and number of factors

  poly_point_init(&R);
  for(i=0; i<n; i++)
  {
    poly_elptic_mul(&R, P, factors[i], E);   when a factor
    if(poly_test_point(R))                   gives point at infinity
    break;                                   we have point order
  }
  if(i<n)
    mpz_set(order, factors[i]);          ← set output order
  else
  {
    printf("missing order in xtended!!\n");     catastrophic error
    exit(-4);                                   kill program
  }
  poly_point_clear(&R);
  return i;                        ← return index of factor as well
}
```

In listing 13.13, I showed the start of a test program. The object of showing the code was to list the points for a tiny example. In the print-out I showed the order of each point using the routines ?? and ??. The input variable `factors[]` is set up as shown in listing ??.

Listing 16.5 Factors setup for tiny example

```
  for(i=0; i<8; i++)                  initialize each element
    mpz_init(factors[i]);             in the array
  mpz_set_ui(factors[0], 5);
  mpz_set_ui(factors[1], 11);         first 3 numbers
  mpz_set_ui(factors[2], 55);         are for base curve
  mpz_set_ui(factors[3], 3);
  mpz_set_ui(factors[4], 33);         last set
  mpz_set_ui(factors[5], 15);         includes all
  mpz_set_ui(factors[6], 165);        remaining possibilities
  mpz_set_ui(factors[7], 1815); ← no points of this order!
```

The order of the G_1 curve is 55 which has factors 5 and 11. So the first three entries are used to define all the possible orders for the base curve. The extension curve has 1815 points with factors 3, 5 and 11^2. Notice that there is no group of order 121 points. As

described in chapter 15, the structure of the field extension curve is

$$E_{F_q} \cong \frac{Z}{d_1 Z} \times \frac{Z}{d_2 Z}$$

with $d_1 = 11$ and $d_2 = 165$ for the tiny example. The two groups are independent, so there are no points of order 121.

The tiny example base curve all by itself will not work with the Weil pairing. It does not have a squared prime value that splits between two groups. The tiny example extension curve does have a squared prime value. So we can use all the points of order 11 on the curve as inputs to the Weil pairing routine.

In the listings of chapter 13 I explained how the points were generated. In listing ?? I show how they are saved in a table. I also indexed the order of each point along with the group "type", which again does not mean anything. It was useful for picking out G_1 and G_2 points.

Listing 16.6 Tiny example point saving

```
while(j < 1814)
{
  poly_elptic_embed(&Px1, &Px2, xtnd, Ex);
  poly_get_order(ordr, Px1, Ex, factors, 8);   ← find the order of the point
  :
  :
  poly_point_init(&table[j]);          | put first point
  poly_point_copy(&table[j], Px1);     | into table
  grp[2*j] = g1g2(Px1);
  grp[2*j + 1] = mpz_get_ui(ordr);   ← convert GMP value to integer
  j++;
```

The vertical dots in listing ?? are the print statements in listing 13.13. With every point placed in `table[]` and every order placed in `grp[]` it is now possible to see how the Weil pairing works.

The main usefulness of the `g1g2()` routine is to pick out base points vs extension field points. Values of 1 are all on the base curve and values of 2, 3 and 4 are on the extension curve. Points of order 11 were chosen because that is a "large prime" in the base curve and because the structure of the extension field curve is [11 x 165].

To test the Weil pairing I chose four points. Three of the points are order 11 and the fourth point is order 55. The fourth point is used as the reference point, and the other 3 points are combined to test equation ??.

The entire test program and all the output are in the code repository. The repository is found at URL https://github.com/drmike8888/Elliptic-curve-pairings. The points for both the base and extension fields are found in directory Chapter13 while the Weil pairing output can be found under directory Chapter16. Some of the output is just debugging data.

Here I want to point out some of the test results because failure shows up. If you run across these kinds of problems while testing your programs, this might be a useful clue on what to fix.

For the G_1 x G_1 test three points are on the base curve. The chosen points were $P = (3, 40)$, $Q = (11, 11)$ and $T = (23, 24)$. The reference point was chosen to be $S = (x, 22x + 32)$. All these numbers are modulo 43, the value of the field prime. The irreducible polynomial for the extension field is $x^2 + x + 3$.

The Weil pairing of the points $e_m(P, Q)$ and $e_m(P, T)$ were both 1. From formula ?? these act like the same point. The powering equation ?? says

$$e_m(aS, bT) = e_m(S, T)^{ab}$$

so the points are directly related to each other by some factor with $Q = aP$ and $T = bP$. Because the numbers are small we could figure out what the multiplier is. The idea behind the exercise is G_1 x G_1 fails to be useful.

For the G_1 x G_2 test I left P as the same point and chose $Q = (x + 4, 15x + 18)$ and $T = (3x + 1, 41x + 32)$. The reference point S was also left to be the same. This time the pairings came out to be

$$\begin{aligned} e_m(P, Q) &= 36x + 25 \\ e_m(P, T) &= 11x + 2 \end{aligned} \tag{16.7}$$

The point $T + Q = 24x + 36$ and the Weil pairing of P with $T + Q$ came out as

$$e_m(P, T + Q) = 37x + 23.$$

Multiplication of $e_m(P, Q)$ with $e_m(P, T)$ gives

$$e_m(P, Q)e_m(P, T) = 37x + 23$$

which is what we expect from the rules of pairings.

Remember that all these operations are done modulo the irreducible polynomial which defines the field extension. Taking these last two results to the 11^{th} power because the order of the points being used is 11 the expected result of 1 is output. That is, two points of order 11 fed into the Weil pairing algorithm result in an 11^{th} root of unity.

In the test for a G_2 x G_2 example I changed the value of P to $(x + 39, 39x + 4)$. The value for Q was $(3x + 1, 2x + 11)$, T was chosen to be $(3x + 16, 41x + 11)$, and S was left as $(x, 22x + 32)$. The results came out to be

$$\begin{aligned} e_m(P, Q) &= 34x + 19 \\ e_m(P, T) &= 11x + 2 \\ e_m(P, Q)e_m(P, T) &= 32x + 34 \\ e_m(P, Q + T) &= 32x + 34 \\ e_m(P, Q + T)^{11} &= 1 \end{aligned} \tag{16.8}$$

I did not do a G_2 x G_1 test. We expect that it would work, and we know from formula ?? that using the same values from the G_1 x G_2 test we should get the inverse result. For the Weil pairing the order of the arguments matters. The G_1 x G_1 test shows the non-degeneracy property, but otherwise it is not useful.

16.4 Summary

- The Weil pairing is computed using four calls to Miller's algorithm. Two points of the same order m are paired and a third reference point of different order are inputs to the operation. The input values are field extension points and the output is a field extension value which is an m^{th} root of unity.
- A Weil pairing of a point with itself gives a result of 1. Most other pairings do not have this property. Algorithms with multiple base points work well with Weil pairing.
- A Weil pairing will compute an inverse result if the input points are exchanged. That is

$$e_m(P, Q) = e_m(Q, P)^{-1}.$$

  This is a very useful way to test code is working correctly.
- The cardinality of an extension curve is easy to compute. It may be exceptionally difficult to factor. For very large field primes this inability to completely factor the cardinality increases the security of pairing friendly curves.

Chapter Bibliography

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York.

16.5 Answer to exercise

- 16.1) With base point T and two private keys a and b the Weil pairing of the two public keys is

$$e_m(aT, bT) = e_m(T, T)^{ab} = 1^{ab} = 1.$$

The Weil pairing of two directly related points is always 1.

17 Tate pairing defined

This chapter covers

- Mathematical description of Tate pairing
- Implementation of Tate pairing
- Test with tiny example to see how Tate pairings work

In this chapter the Tate pairing is described. This will be used with the example in chapter 19.

The Tate pairing has the properties of bilinearity and non-degeneracy but not the alternating property which the Weil pairing of chapter 16 possesses. In this chapter I will first go over the mathematical description of the Tate pairing. In chapter 16 pairing test code utility routines were described, so this chapter will just show how the Tate pairing is computed with one listing.

The tiny example introduced in chapter 13 will again be used to show how the number of points available for the Tate pairing is enlarged over the Weil pairing. Explaining the math with words is great, but seeing an example should help to get an intuitive feel for what the mathematics actually means. Using the same tiny example, the Tate pairing will be explored in detail.

17.1 Tate pairing mathematics

In this section the mathematics of the Tate pairing is explained.

Similar to the general pairing description of a curve having two cyclic groups, the Tate pairing is described in reference (?) section XI.9 with the mathematical statement

$$\tau : E[m] \times \frac{E}{mE} \longrightarrow \mu_m \tag{17.1}$$

The symbol τ is the Tate pairing operation. To distinguish it from the Weil pairing I'll use $\tau(\cdot, \cdot)$ for the Tate pairing of two points. The symbol $E[m]$ means all the points of order m on the curve E. The symbol μ_m is again the m^{th} root of unity.

That leaves the symbol $\frac{E}{mE}$ which to be honest, threw me for a loop the first time I ran across it. It's actually pretty simple. It means the points on curve E with all the points m times every point on E removed. The reason this confused me is that the points of order m go to the point at infinity. Those points are *not* removed. Points which have an order with a factor of m are also not removed. The points which are not the point at infinity after multiplication by m are removed, so we are left with points of order m and all points which have m as one of the factors of their order.

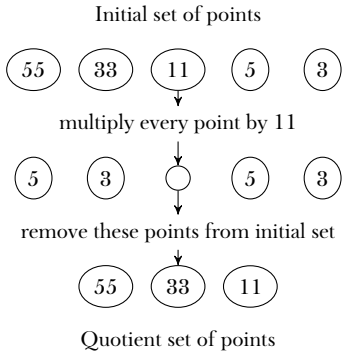

Figure 17.1 Example of quotient set $\frac{E}{mE}$ for $m = 11$. The first row shows all possible point orders, the second row shows the resulting orders of points after multiplying all points by 11, the third row shows the first row minus the second row.

Figure ?? shows a concrete example taken from the tiny field extension curve. Each circle represent all the points with the order given by the number in the circle. When points of order 55 and 33 are multiplied by 11 the result is a point of order 5 and 3 respectively. Points of order 11 go to infinity, so that circle is empty. Points of order 5 and 3 are removed from the possible points in a Tate pairing, but the points of order 55 and 33 can be used. That is really cool.

The Tate pairing calculation formula is found in section 5.1 of (?). In terms of Miller's formula $f_p(R)$ it is

$$\tau(P, Q) = \left(\frac{f_P(Q + S)}{f_P(S)} \right)^{(p^k - 1)/m} \tag{17.2}$$

where m is the order of the points we are interested in. Equation 13.1 ($p^k \cong 1 \bmod m$) says that m divides $p^k - 1$. That means the exponent in equation ?? is an integer. For high security situations it is a very large integer as seen in table 14.1. For a 256 bit security level that exponent is over 15,000 bits. So while the Weil pairing requires four calls to the Miller function, the Tate pairing requires two calls to the Miller function plus a final power operation. In the literature I have seen arguments that one is faster than the other, but the reality is "it depends" on the case.

If the two points of input are the same, we do not get 1 as an answer. This is different from the Weil pairing. In addition, if we swap the inputs we do not get an inverse, we get a completely different result. Most protocols which use the Tate pairing are very specific about which points go into each slot especially if there is a requirement to use a G_1 x G_2 pairing.

Exercise 17.1

An elliptic curve with field prime 41 and large prime $m = 29$ has embedding degree 4. What is the exponent value for a Tate pairing of this field extension curve?

17.2 The Tate pairing subroutine described

In this section the code to compute the Tate pairing of points on an elliptic curve over a field extension is described.

The execution of equation ?? is straightforward. The use of the Tate subroutine is illustrated in figure ?? and the code is shown in listing ??. As with the Weil pairing the first two points are being paired and the third point is the reference for the Miller calculation. Unlike the Weil pairing, the second point only requires that it have the factor m in the order of the point. The first point must be of order m. I verified this with brute force testing as we'll see in a bit.

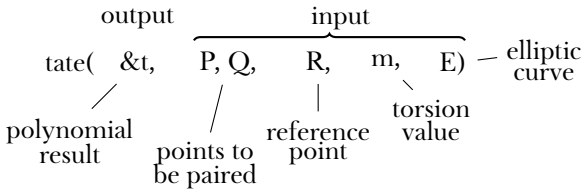

Figure 17.2 Tate pairing subroutine calling parameters

Listing 17.1 Tate pairing routine

```
void tate(POLY *t, POLY_POINT P, POLY_POINT Q, POLY_POINT S, mpz_t m,
          POLY_CURVE E)
{
  POLY_POINT QpS;
```

```
  POLY t1, t2;
  mpz_t pw;

  poly_point_init(&QpS);
  poly_init(&t1);
  poly_init(&t2);                          | create point
  poly_elptic_sum(&QpS, Q, S, E);          | Q + S
  miller(&t1, P, QpS, m, E);     ← numerator = f_P(Q + S)
  miller(&t2, P, S, m, E);       ← denominator = f_P(S)
  poly_div(t, t1, t2);       ← f_P(Q + s)/f_P(S)
  poly_q_get(pw);                          | prepare result
  mpz_sub_ui(pw, pw, 1);                   | to power (p^k − 1)/m
  mpz_divexact(pw, pw, m);                 |
  poly_pow(t, *t, pw);       ← full result
  poly_point_clear(&QpS);
  poly_clear(&t1);                         | clean
  poly_clear(&t2);                         | up
  mpz_clear(pw);                           | stack
}
```

The reference point S cannot be related to either input point. One suggestion I found in the literature was to choose a point S with x component equal to zero. Unfortunately that might be a point of order m so that choice would not work in every case.

Listing ?? follows the equation ?? directly. The point $Q + S$ is found, then the Miller algorithm is computed for that point. The Miller algorithm is computed for the point S and then the two results are divided. The value $(p^k - 1)/m$ is computed. This is used as the power to the result from division of Miller functions and the final result is placed in the requested location. Now that we have all the machinery of modulo polynomial operations available, it looks easy.

17.3 Testing the Tate pairing using a tiny example

In this section a printable point code example of the Tate pairing is explained.

The Weil tiny example program discussed in chapter 16 was copied and modified to use the Tate pairing. The program is in the repository under directory Chapter17 as is the output. Saving the points to a file was removed. But the table of points was kept, so any point could be tested. Listing ?? shows the essential setup copied from the Weil tiny example into the Tate tiny example. The group size is 11 and the extension curve has the same coefficients.

Listing 17.2 Tate tiny setup

```
  mpz_init_set_ui(prm, M);
  minit(prm);                  | set up field prime to 43
  poly_init(&irrd);
     :
  poly_irrd_set(irrd);         | set up irreducible polynomial
  poly_mulprep(irrd);          | and multiplication table
```

```
    :
  mpz_set_ui(factors[2], 11);      ← m = 11 is prime group order
    :
  mpz_init_set(tor, factors[2]);
    :
  poly_curve_init(&Ex);
  mpz_set_ui(Ex.a4.coef[0], 23);   | same curve
  mpz_set_ui(Ex.a6.coef[0], 42);   | for tiny example
    :
```

Rather than pick specific points to test the algorithm, I decided to use the built-in random number generator of gcc initialized with the nanosecond clock as a seed. The setup for this is shown in listing ??.

Listing 17.3 Tate random setup

```
#include <time.h>
    :                              | use internal
  struct timespec ts;              | clock to change
    :                              | rand seed

  clock_gettime(CLOCK_MONOTONIC_RAW, &ts);  | use nanoseconds
  srand(ts.tv_nsec);                        | to stir things up
```

Since the order of the point matters for the algorithm, I set up an array with each index into the list of points separated by order. The first step was to determine how many points were in each order. I modified listing 13.15 to increment a counter with the same index as the factor found from the `poly_get_order()` routine. This is shown in listing ??. There are two points for every embedded value, so the counter is bumped by two each time.

Listing 17.4 Tate order counting

```
    k = poly_get_order(xtndordr[j], xtndpnt[j], Ex, factors, 7);
    numxtdpnts[k] += 2;          | increment correct order
     :
     :
  for(i=0; i<7; i++)             | output number of points
  {                              | in each order
    gmp_printf("order %Zd has %d points\n", factors[i], numxtdpnts[i]);
    pdex[i] = 0;
  }
```

Listing ?? shows the output from listing ??.

Listing 17.5 Tate orders found

```
order 3 has 2 points
order 5 has 4 points
order 11 has 120 points
```

```
order 15 has 8 points
order 33 has 240 points
order 55 has 480 points
order 165 has 960 points
```

The largest order has 960 points and there are 7 different orders, so the index list is 7 groups of 1000 index values. Table ?? is a schematic layout of the point list array. The point list array holds an index into the list of points created by brute force embedding, not the (x, y) values for each point.

Table 17.1 Point array for Tate pairing test

order	3	5	11	15	33	55	165
Points	P_3^1	P_5^1	P_{11}^1	P_{15}^1	P_{33}^1	P_{55}^1	P_{165}^1
	P_3^2	P_5^2	P_{11}^2	P_{15}^2	P_{33}^2	P_{55}^2	P_{165}^2
		P_5^3	P_{11}^3	P_{15}^3	P_{33}^3	P_{55}^3	P_{165}^3
		P_5^4	P_{11}^4	P_{15}^4	P_{33}^4	P_{55}^4	P_{165}^4
			$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

The creation of the `point_list[]` array was by brute force as shown in listing ??. Every point and every possible order was checked. This allowed me to take the points found from the embedding order and create the array as shown in table ??.

Listing 17.6 Tate point_list[] array creation

```
point_list = (int*)malloc(sizeof(int)*7*1000);   <- one column for each factor
                                                    one entry for each point
for(j=0; j<XTEND; j++)
{                                                 | loop over each point
  for(i=0; i<7; i++)
  {                                               | loop over each factor
    if(!mpz_cmp_ui(factors[i], grp[2*j + 1]))     <- look for which column this point belongs
    {
      k = i*1000 + pdex[i];                       | 2D index into point_list
      point_list[k] = j;                          | saves this point's place
      pdex[i]++;                                  <- each column has an index counter
    }
  }
}
```

A random point was selected from a particular column as shown in listing ??. The input to the routine includes the `grp[]` array which holds the order and group type, a pointer to one of the columns in the `point_list[]` array, the length of that array and 0 for a G_1 or 1 for a G_2 point. If there is no G_1 point for a selected order it is an infinite loop that never exits. I realized this mistake a few times. The purpose of this test is education, so I learned a lot!

Listing 17.7 Tate random selection

```
int rndselect(long *grp, int *point_list, int nmpnt, int type)
{
  int j, k, r;

  r = -1;
  while(r < 0)
  {
    k = rand() % nmpnt;
    j = point_list[k];
    if(!type && (grp[2*j] == 1))
      r = j;
    else if(type && (grp[2*j] > 1))
      r = j;
  }
  return j;
}
```

↑ G_1=0, G_2=1
number of points to pick from
pointer to column with correct order
pseudo random modulo length of array
← index of point in grp[] table
choose index if correct group
← repeat forever if no match!

For the tests of orders 11x11 and 11x55, the point S was found using the criteria that it be in $G2$ and have order 3. Since there are only two points of order 3 there is not much choice. For tests with orders 11x33 I changed S to be order 5 in G_1. I left S as order 5 in G_1 for 11x165 tests as well.

I also tried reversing the orders using 33x11, 55x11, 165x11 and 55x55. Every single one of these failed to compute a correct pairing such that

$$\tau(P, Q+T) = \tau(P, Q)\tau(P, T). \tag{17.3}$$

On occasion one of these backward tests would give a matching result, but running the random selection 20 times showed these were accidents of luck.

Listing ?? shows one of the test programs performed where the second group is order 165 which is only possible for a G_2 group. The two tests are orders 11x165 with G_1 x G_2 and G_2 x G_2. The point S was reset to order 5 at the beginning of this test.

Listing 17.8 Tate 11x165 test

```
k = rndselect(grp, &point_list[1000], numxtdpnts[1], 0);
poly_point_copy(&S, xtndpnt[k]);
poly_point_printf("order 5 S:\n", S);
printf("=========================================\n");
printf("Tate G1 x G2* (order 11x165)\n\n");
k = rndselect(grp, &point_list[2000], numxtdpnts[2], 0);
poly_point_copy(&P, xtndpnt[k]);
poly_point_printf("P:\n", P);
k = rndselect(grp, &point_list[6000], numxtdpnts[6], 1);
poly_point_copy(&Q, xtndpnt[k]);
poly_point_printf("Q:\n", Q);
k = rndselect(grp, &point_list[6000], numxtdpnts[6], 1);
poly_point_copy(&T, xtndpnt[k]);
```

set S to order 5, G_1
set P to order 11, G_1
set Q to order 165, G_2

```
                                                          set T to
                                                          order 165, G2
poly_point_printf("T:\n", T);
tate(&t1, P, Q, S, tor, Ex);       ← compute τ(P,Q)
poly_printf("tate(P, Q): ", t1);
tate(&t2, P, T, S, tor, Ex);       ← compute τ(P,T)
poly_printf("tate(P, T): ", t2);
poly_mul(&t3, t1, t2);             ← product τ(P,Q)τ(P,T)
poly_printf("(P, Q)*(P, T): ", t3);    ←
poly_elptic_sum(&TpQ, T, Q, Ex);   ← compute Q+T          compare
tate(&t4, P, TpQ, S, tor, Ex);     ← compute τ(P,Q+T)     output
poly_printf("(P, T+Q): ", t4);          ←

printf("=========================================\n");
printf("Tate G2 x G2* (order 11x165)\n\n");
k = rndselect(grp, &point_list[2000], numxtdpnts[2], 1);
poly_point_copy(&P, xtndpnt[k]);                          set P to
poly_point_printf("P:\n", P);                             order 11, G2
k = rndselect(grp, &point_list[6000], numxtdpnts[6], 1);
poly_point_copy(&Q, xtndpnt[k]);                          set Q to
poly_point_printf("Q:\n", Q);                             order 165, G2
k = rndselect(grp, &point_list[6000], numxtdpnts[6], 1);
poly_point_copy(&T, xtndpnt[k]);                          set T to
poly_point_printf("T:\n", T);                             order 165, G2
tate(&t1, P, Q, S, tor, Ex);       ← compute τ(P,Q)
poly_printf("tate(P, Q): ", t1);
tate(&t2, P, T, S, tor, Ex);       ← compute τ(P,T)
poly_printf("tate(P, T): ", t2);
poly_mul(&t3, t1, t2);             ← product τ(P,Q)τ(P,T)
poly_printf("(P, Q)*(P, T): ", t3);    ←
poly_elptic_sum(&TpQ, T, Q, Ex);   ← compute Q+T          compare
tate(&t4, P, TpQ, S, tor, Ex);     ← compute τ(P,Q+T)     output
poly_printf("(P, T+Q): ", t4);          ←
```

I ran this test many times and was happy to see equation ?? matched every time. Table ?? shows the summary of two of these runs. Each run has a G_1 x G_2 column with point P being the G_1 value and points Q and T being order 165 in G_2. The second column of each run has P being order 11 in G_2. There are enough random points that we see no duplicates. Each point entry is an x, y pair and all values are modulo 43.

Table 17.2 Tate pairing tiny test random points 11x165

	run 1		run 2	
	G_1 x G_2	G_2 x G_2	G_1 x G_2	G_2 x G_2
S	39, 12		39, 31	
P	30, 30	$13x + 17, 10x + 21$	24, 20	$19x + 12, 37x + 30$
Q	$27x + 35, 2x + 16$	$7x + 36, 18x + 5$	$13x + 21, 10x + 1$	$23x + 35, 42x + 35$
T	$15x + 32, 5x + 4$	$25x + 25, 42x + 29$	$40x + 35, 18x + 38$	$33x, 41x + 8$
$\tau(P, Q)$	$6x + 29$	$12x + 17$	$31x + 5$	$7x + 32$
$\tau(P, T)$	$36x + 25$	$11x + 2$	$31x + 5$	$9x + 28$
$\tau(P, Q)\tau(P, T)$	$32x + 34$	$36x + 25$	$37x + 23$	$34x + 19$
$\tau(P, T + Q)$	$32x + 34$	$36x + 25$	$37x + 23$	$34x + 19$

At the bottom of the table we see that the multiply of Tate pairings equals the Tate pairing of the sum of points. We also see duplicate values for pairings. This makes sense because there are only 11 possible values for pairing results. They must be 11^{th} roots of unity, and only 11 of the $43^2 = 1849$ values have this property.

17.4 Summary

- The Tate pairing is a bilinear pairing between two points on an elliptic curve defined over a finite field.
- The Tate pairing works with points which have a factor m of large prime order for the second point. The first point must only be of large prime order m. Mathematically this is written as

$$E[m] \times \frac{E}{mE} \longrightarrow \mu_m$$

- The Tate pairing uses two calls to the Miller algorithm followed by being taken to the power of $(p^k - 1)/m$. The formula is

$$\tau(P, Q) = \left(\frac{f_P(Q + S)}{f_P(S)} \right)^{(p^k - 1)/m}$$

Chapter Bibliography

Menezes, Alfred. 2009. An introduction to pairing-based cryptography. *Contemporary Mathematics*, **477**, 47–65. Providence, RI.

Silverman, J.H. 2013. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York.

17.5 Answer to exercise

- 17.1) From formula ?? we get

$$\frac{41^4 - 1}{29} = 97440.$$